FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Generation of Custom Run-time Reconfigurable Hardware for Transparent Binary Acceleration

Nuno Miguel Cardanha Paulino

Programa Doutoral em Engenharia Electrotécnica e de Computadores (PDEEC)

Supervisor: João Canas Ferreira (Assistant Professor)

Co-supervisor: João M. P. Cardoso (Associate Professor)

June 2016

Abstract

With the increase of application complexity and amount of data, the required computational power increases in tandem. Technology improvements have allowed for the increase in clock frequencies of all kinds of processing architectures. But exploration of new architecture and computing paradigms over the simple single-issue in-order processor are equally important towards increasing performance, by properly exploiting the data-parallelism of demanding tasks. For instance: superscalar processors, which discover instruction parallelism at runtime; Very Long Instruction Word processors, which rely on compile-time parallelism discovery and multiple issue-units; and multi-core approaches, which are based on thread-level parallelism exploited at the software level.

For embedded applications, depending on the performance requirements or resource constraints, and if the application is composed of well-defined tasks, then a more application-specific system may be appropriate, i.e., developing an Application Specific Integrated Circuit (ASIC). Custom logic delivers the best power/performance ratio, but this solution requires advanced hardware expertise, implies long development time, and especially very high manufacture costs.

This work designed and evaluated a transparent binary acceleration approach, targeting Field Programmable Gate Array (FPGA) devices, which relies on instruction traces to automatically generate specialized accelerator instances. A custom accelerator, capable of executing a set of previously detected loop traces, is coupled to a host MicroBlaze processor. The traces are detected via simulation of the target binary. The approach does not require the application source code to be modified, which ensures the transparency for the application developer. No custom compilers are necessary, and the binary code does not need to be altered either offline or during runtime.

The accelerators contain per-instance reconfiguration capabilities, which allow for the reuse of computing units between accelerated loops, without sacrificing the benefits of circuit specialization. To increase the achievable performance, the accelerator is capable of performing two concurrent memory accesses to the MicroBlaze's data memory. The repetitive nature of the loop traces is exploited via loop-pipelining, which maximizes the achievable acceleration. By supporting single-precision floating-point operations via fully-pipelined units, the accelerator is capable of executing realistic data-oriented loops. Finally, the use of Dynamic Partial Reconfiguration (DPR) allows for significant area savings when instantiating accelerators with numerous configurations, and also ensures circuit specialization per-configuration.

Several fully functional systems were implemented, using commercial FPGAs, to validate the design iterations of the accelerator. An initial design relied on translating Control and Dataflow Graph representations of the traces into a multi-row array of Functional Units. For 15 benchmarks, the geometric mean speedup was 2.08×. A second implementation augmented the accelerator with shared memory access to the the entire local data memory of the MicroBlaze. Arbitrary addresses can be accessed without need for address generation hardware. Exploiting data-parallelism allows for targeting of larger, more realistic traces. The mean geometric speedup for 37 benchmarks was 2.35×. The most efficient implementation supports floating-point operations and relies on loop pipelining. The developed tools generate an accelerator instance by modulo-scheduling each trace at the minimum possible Initiation Interval. The geometric mean speedup for a set of 24 benchmarks is 5.61×, and the accelerator requires only 1.12× the FPGA slices required by the MicroBlaze. Finally, resorting to DPR, an accelerator with 10 configurations requires only a third of the Lookup Tables relative to an equivalent accelerator without this capability.

To summarize, the approach is capable of expediently generating accelerator-augmented embedded systems which achieve considerable performance increases whilst incurring a low resource cost, and without requiring manual hardware design.

Sumário

Com o aumento da complexidade das aplicações, aumenta também a capacidade computacional necessária. Melhorias tecnológicas têm permitido o aumento da frequência de relógio para todo o tipo de arquitecturas computacionais. Contudo, a exploração de novas arquitecturas é igualmente importante para melhorias de desempenho, explorando eficientemente o paralelismo de dados de tarefas exigentes. Por exemplo: processadores *superscalar*, que descobrem paralelismo ao nível da instrução durante a execução; processadores VLIW (Very Large Instruction Word), que dependem de paralelismo explorado durante a compilação e de várias unidades em paralelo; e tecnologias multi-core, que se baseiam na exploração de paralelismo ao nível da *thread* através de software.

Para aplicações embebidas, dependendo dos requisitos de desempenho ou restrições de recursos, e se a aplicação for composta por tarefas bem definidas, um sistema mais especifico poderá ser mais apropriado, i.e., desenhar um ASIC (Application Specific Integrated Circuit). Lógica dedicada beneficia do melhor desempenho por *watt*, mas esta solução requer experiência de desenho de *hardware*, sofre de tempo de desenvolvimento longo, e custos de fabricação elevados.

Este trabalho desenvolveu e avaliou uma abordagem de aceleração transparente de código binário, orientada especificamente para FPGAs. A abordagem baseia-se em sequências frequentes de instruções executadas (i.e, *traces*) para gerar automaticamente aceleradores especializados. Um acelerador, capaz de executar um conjunto de *traces* previamente detectados, complementa um MicroBlaze, que age como processador principal. Os *traces* são detectados por simulação da aplicação, e representam ciclos de execução (i.e., *loops*). Não é necessário modificações ao código-fonte, o que aumenta a transparência da abordagem para o programador. Um compilador especializado não é necessário, e o código binário não sofre modificações pós-compilação.

O acelerador contém lógica de reconfiguração especializada, o que permite a reutilização de unidades de cálculo entre os *loops* acelerados, sem sacrificar a especialização. Para maximizar o desempenho, o acelerador é capaz de efectuar dois acessos paralelos à memória de dados do processador. O uso de *loop-pipelining* maximiza a aceleração, e o suporte para operações de vírgula flutuante permite a execução de tarefas embebidas realistas. Finalmente, o uso de Reconfiguraçao Parcial Dinâmica (DPR), reduz significativamente a área necessária para suportar várias configurações, e assegura a especialização do hardware respectivo a cada configuração.

Vários sistemas totalmente funcionais foram implementados para validar os aceleradores, usando FPGAs comerciais. Um primeira implementação traduz representações de grafo de dados (i.e., Control and Dataflow Graph) dos *loops* para várias linhas de unidades funcionais interligadas. Para 15 *benchmarks*, a média geométrica da aceleração foi de 2.08×. Uma segunda implementação adiciona ao acelerador suporte para acesso à memória. O mesmo é capaz de aceder directamente a toda a memória local de dados do MicroBlaze, sendo suportados acessos endereços arbitrários. Explorar o paralelismo de dados permite acelerar *loops* mais realísticos. Para 37 benchmarks, a média geométrica da aceleração foi de 2.35×. A implementação mais eficiente suporta operações de vírgula flutuante e utiliza *loop-pipelining*. Um escalonador gera uma instância do acelerador efectuando *modulo-scheduling* para cada *trace* ao intervalo de iniciação (i.e., Initiation Interval) mínimo. A média geométrica da aceleração é de 5.61× em média para 24 aplicações, e os aceleradores requerem 1.12× o número de *slices* de FPGA de um MicroBlaze. Finalmente, com o uso de DPR, um acelerador com 10 configurações necessita apenas de um terço das *Lookup Tables* relativamente a um acelerador equivalente sem esta capacidade.

Concluindo, a abordagem permite gerar rapidamente sistemas com aceleradores especializados que aumentam consideravelmente o desempenho, com um custo reduzido em termos de recursos, evitando também a necessidade de desenho de *hardware* manual.

Acknowledgments

This thesis is the result of four years of work that I was fortunate enough to be able to carry out with focus and nearly undivided attention due mostly, if not totally, to the support of my parents, who helped keep my mind of other, more time consuming, and infinitely less productive matters.

The individual moments where someone lent me their support, and particular people who took the time to care and ask about my work, are too numerous to list. A thank you to the friends that shared a seat in the lab where I sit as I type this. It would go without saying, but I'd also like to thank my supervisor, João Canas Ferreira, for his guidance and insight which helped make sure I didn't stray off into the distance on a random direction. Also, a thank you to my co-supervisor, João Manuel Paiva Cardoso, for many fruitful discussions, and another to João Bispo, both for suggestions and for providing his own set of tools, which were the starting point for what I have developed. I would also like to thank Michael Hübner for his interest in my work, as well as Max Ferger for some helpful suggestions.

Finally, a special thanks goes out to my friend Henrique Martins, with the hope that he comes to his senses and comes back home.

On a completely different note, I would also like to acknowledge the support through PhD grant SFRH/BD/80225/2011, provided by FCT (*Fundação para a Ciência e a Tecnologia* - Portuguese Foundation for Science and Technology). Finally, thank you to Stephen Wong from the Delft University of Technology in the Netherlands, for the ρ-VEX processor release and tools.

Nuno Paulino

"Terry took Death's arm and followed him through
the doors and on to the black desert under the endless night."

Terry Pratchett

Contents

List of Figures

Listings

List of Tables

Acronyms

ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CCA	Configurable Compute Accelerator
CCS	Compiled Code Simulator
CDFG	Control and Dataflow Graph
CGRA	Coarse Grained Reconfigurable Array
CPL	Critical Path Length
CR	Communication Routine
DIM	Dynamic Instruction Merging
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor
FF	Flip Flop
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSL	Fast Simplex Link
FU	Functional Unit
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
HW/SW	Hardware/Software
ICAP	Internal Configuration Access Port
IC	Integrated Circuit
II	Initiation Interval
ILP	Instruction Level Parallelism
IPC	Instructions per Clock Cycle
IP	Intellectual Property
LMB	Local Memory Bus
LUT	Lookup Table
MAC	Multiply Accumulate
MAM	Memory Access Manager
MIMD	Multiple Instruction Multiple Data
MRT	Modulo Reservation Table
PLB	Processor Local Bus
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SoC	System-on-a-chip
VEX	VLIW Example
VLIW	Very Long Instruction Word

Chapter 1

Introduction

With the constant increase of application complexity and data volume, the required computational power increases in tandem. This is especially true for embedded systems, where the performance must be maximized while also incurring the least cost, both in terms of chip area and power consumption. To meet these requirements, a great development effort has always been placed on designing faster, smaller and more power-efficient circuits to tackle the increasing demands of applications that range from industrial scenarios to consumer electronics. Even more importantly, the capabilities of the computing architectures, especially from an application developer point of view, are essential to allow efficient implementations of applications.

Specifically, it is generally possible to improve application performance by either 1) enhancing General Purpose Processor (GPP) architectures, or otherwise deploying processor designs more sophisticated than simple single-issue, in-order, processors, or 2) relying on heterogeneous architectures, where the demanding portions of an application execute in dedicated computing elements, whilst the remainder of the application executes in a main GPP. The following paragraphs briefly contrast these paradigms. Afterwards, the reason as to why this work adopted the later type of approach is explained, specifically focusing on the importance of heterogeneous system architectures to efficiently implement embedded applications.

Improving Processor Performance and Architecture In order to improve GPP performance, the most straightforward method is the increase of clock frequency to increase throughput. This clearly has technological limitations, as well as power consumption implications; it is not an architectural improvement. In contrast, other approaches alter the single instruction sequential pipeline architecture, such as vector or superscalar processors, first introduced in 1960s [BDM+72, Tho80]. The former are capable of working on several data at once, while the latter employ hardware detection of data dependencies to execute more than one instruction per clock cycle. Very Long Instruction Word (VLIW) processors are another approach, in which parallel instructions are detected at compile time and merged into a single instruction. Yet another approach is based on assigning tasks to multiple processors or cores (in multi-core architectures). This allows for thread-level parallelism, managed by application programmers at a software level. Naturally,

each of the approaches has limitations. For instance, runtime exploitation of Instruction Level Parallelism (ILP) with superscalar processors is limited to a small execution scope and requires additional power-hungry hardware to determine the instruction dependencies. As designs incorporate more resources at increasingly smaller scales, the device transistor increases, becoming difficult to dissipate heat and meet power constraints. These technological limitations mean that these architectures may eventually reach a performance wall [EBSA+12, Nat11].

Heterogeneous Architectures An alternative approach is based on implementing applications on systems containing one (or more) GPPs alongside dedicated Intellectual Property (IP) blocks (if considering a single chip) or Integrated Circuits (ICs), specialized for certain tasks. These specialized units execute the portions of the application(s) which represent the bulk of computation, and have potential parallelism to exploit. This follows Amdahl's law, which states that an application's potential speedup is limited by the amount of its computation that can be made parallel. The remainder of the application can be executed in its shortest possible time in a sequential fashion. Dividing the application into tasks to assign to the computing devices on the target system, or designing the heterogeneous system itself based on the application requirements, is typically referred to as Hardware/Software (HW/SW) partitioning [Wol03, Tei12] (or HW/SW co-design),

The notion of heterogeneous computing is far from new, and is present at many scales, from High-Performance Computing machines to embedded devices. An example is the use of Graphics Processing Units (GPUs) for personal-computers. Another are Digital Signal Processors (DSPs) for embedded applications such as audio/video or encryption. Another commonplace example are Systems-on-a-chip (SoCs) for mobile devices, containing different types of computing elements.

When targeting these devices, the developer must partition the application: determine from the application specifications which portions are adequate for each device on the system, (re-)write the software with the envisioned partition in mind, and (obligatorily) rely on software libraries/frameworks targeting the dedicated devices in the system, such as OpenCL or CUDA. These types of solutions are very programmable, and cover a large range of applications within their own domains. The use of OpenCL especially increases the applicability of devices such as GPUs [MV15].

However, it may be the case that a more specific and application-dependant system is desired. This is true for cases where the application(s) contains more specific tasks or processes. That is, yet another alternative is to develop the specialized hardware itself, according to an given application partition. In other words, some applications justify the design of ASICs, i.e., an application specific heterogeneous platform. The flexibility of the system is assured by the GPP, and the performance is assured by custom circuit(s). This solution however requires very advanced hardware expertise, and the development is lengthy and error-prone. Also, this type of design is only economically viable for large production volumes, as the fabrication costs of ASICs are very high.

To summarize, designing per-application circuits to target demanding data-oriented tasks is usually not economically viable. Instead, solutions such as GPUs or DSPs are satisfactory in terms of balancing cost and performance, and are also more approachable by developers. However, it

is also true that these solutions can be either to excessive for the performance requirements of an application, or too expensive. Also, designing an ASIC (even for small defined tasks in embedded applications) implies a manual HW/SW partitioning effort which must be very well guided in order to avoid several design iterations, which require lengthy design and validation time.

These aspects lead to the notion of not only automated circuit specification, but also of automated HW/SW partitioning. The purpose is to achieve the desired performance increase without requiring a manual partitioning effort, hardware design expertise, or suffering lengthy development time. That is, applications could benefit from improved performance without resorting to expensive off-the-shelf chips or laborious hardware design.

The following section explains how FPGAs are ideal platforms for development flows of this nature, especially when iterative design is required. Section 1.2 summarizes existing automated HW/SW partitioning approaches.

1.1 FPGAs as a Platform for HW/SW Partitioning Design

There are several systems which can be classified as heterogeneous. For example SoC is essentially a heterogeneous system, since it contains many different types of computing elements, and a recent family of devices from AMD integrate a conventional CPU and GPU into a single chip, supported by a heterogeneous programming paradigm [AMD]. This work however focuses on embedded applications for which solutions such as SoCs, DSPs or manual hardware design are excessive. Instead, FPGAs are the target device utilized to implement the heterogenous systems.

When SRAM-based FPGAs first appeared, their primary purpose was fast prototyping of hardware circuits. That alone makes them attractive for hardware design, as the circuits may easily go through several revisions. They were not first seen as deployment devices, due to the small number of logic gates, high static power consumption, and the unfamiliar tools and languages which supported them [Tri15]. With technology improvements FPGAs can now contain up to 50 million ASIC equivalent logic gates [Xild], and operate at frequencies around 700 MHz [Xilb]. Also, some FPGA families are geared towards low power consumption [Xil15b].

So, an FPGA is essentially a heterogeneous device in itself, given that it is composed of several different types of hardware components. Due to its programmability, its possible to implement a SoC on an FPGA, by instantiating any logic the target application requires: essentially a *soft-SoC*, in the same way that Xilinx's MicroBlaze is a soft-core processor. The recent Xilinx Zynq devices are an example of the opposite: hardcore SoC logic coupled to configurable FPGA fabric [Xile]. So both types of chip, processors/SoCs and FPGAs, are converging to a single type of device.

Even outside the embedded domain, FPGAs are presenting themselves as interesting devices for the future of heterogeneous computing. Notable examples include big data and data-center applications [ORK+15, PCC+14]. This demonstrates that FPGAs are not only the prototyping device of choice, but are increasingly interesting as deployment devices, accompanying the trend towards new heterogeneous programming and design paradigms.

1.2 Automated HW/SW Partitioning

Given the effort associated with designing custom co-processor circuits when relying on manual application partitioning, the concept of automated HW/SW partitioning has been the subject of research for several decades [Tei12]. Two types of approaches which generally rely on autonomously migrating execution of software to dedicated hardware can be outlined: those that rely on high-level source code, and those that rely on binary-level information.

1.2.1 High-Level Synthesis

As early as the 1980's there has been work on specialized compilers which generate custom circuits from high-level code [MS09], complementing traditional software compilers. Examples of current sophisticated vendor tools include Calypto's Catapult C [Cal], Xilinx's Vivado HLS [Xilc], Synopsys' Synphony Model Compiler [Syn], and recently Xilinx's SDSoC [Xila]. In relying on high-level code, typically *C*, these approaches fall into the category of High Level Synthesis (HLS). They rely on a designer-guided step to identify candidate functions to translate into circuits, typically those which implement demanding tasks and which are amenable to acceleration. The HLS tools then generate custom circuits by analysis of the source code.

However, they impose syntactic and/or functional restrictions on the source code, such as lack of support for floating-point, not supporting the complete source language syntax, and dealing poorly with control oriented functions and loops [MS09]. More importantly, from an ease of adoption perspective, they still require considerable designer effort. Specifically, although the hardware is automatically generated, there is still a need for a back-and-forth iterative design between hardware and software, just as manual HW/SW partitioning effort at source code level would require. The developer must write code while considering the effects of the HLS tool, e.g., how the computation is implemented, and how data is accessed and organized. Also, the partitioned software needs to be re-written to interface with the hardware, and the system must be re-designed to integrate it. This typically entails developing interfaces or equipping the function accelerator with local memories or memory access capability via Direct Memory Access (DMA). Finally, the iterative nature of the flow also comes from the need to ensure that it is the most demanding portions of the application that are the ones targeted for partitioning. A solution is to profile each resulting attempt at a partition, as is done by Xilinx's SDSoC.

To avoid these issues, approaches can instead rely on compiler-driven partitioning steps or on an analysis of post-compilation information, i.e., binary code.

1.2.2 Binary-level HW/SW Partitioning

In contrast to the high-level approaches to partitioning, binary-based approaches rely only on post-compile information. That is, they process application binaries resulting from a standard compilation flow. This is often referred to as binary-level HW/SW partitioning, or dynamic HW/SW

partitioning, if the approach relies on binary runtime information. There are a number of consequences to this type of approach: the partitioning step is no longer (usually) developer-guided, the application is not necessarily partitioned along function boundaries, any syntactical complexity (or otherwise unsupported syntactical constructs) of the source code is no longer an issue, and the source code does not need to be modified to interface with the generated accelerator hardware.

Since it is up to either a compiler or post-compilation step to manipulate or analyse the binary for partitioning, there is less interference with the traditional software development flow. The portions of the application which are automatically detected for acceleration may be short acyclic sequences of instructions [CBC+05, NMIM12], instruction sequences delimited by backwards branches (i.e., basic blocks) [LV09], or even sequences of basic blocks [BRGC08, LCDW15]. Consequently, the partitioned code may represent only a portion of a function body, or may include an inlined function call to a nested function. In order to correctly target the most demanding portions of the application, the partitioning may rely on execution traces to determine frequently executing instruction sequences, as opposed to selecting sequences solely by a static binary analysis.

This type of approach has some disadvantages. Firstly, the binary lacks information which is present at high-level source code. For instance, these approaches suffer when the compiler introduces built-in subroutines for data-type casting or floating-point emulation. The context of the operation being performed is lost at the binary level. Secondly, a binary level analysis cannot usually rely on optimizations such as loop fissioning or fusion. Some approaches attempt to recover some information via decompilation [SV05]. Thirdly, the performance improvements depend on how the computations are expressed by the target GPPs's instruction set. This type of approach has been applied to single-issue [LV09, BRGC08] and VLIW instruction sets alike [FDP+14].

Finally, these approaches typically do not rely in a complete circuit synthesis as-such, as HLS approaches do. Instead, they are based on *customization* or *reconfiguration* of a pre-designed accelerator peripheral, such as a Coarse Grained Reconfigurable Array (CGRA) or other array of processing elements. On one hand, the developer does not need to manually integrate the custom hardware into the remainder of the system. On the other hand, the use of a modified processor and/or binary modification may be required, and the hardware adaptability is limited by the capabilities of the configurable accelerator(s).

1.3 Motivation and Problem Statement

As the previous paragraphs have shown, there are many ways to devise, and support development for heterogeneous computing systems for acceleration. Depending on the application, powerful devices such as GPUs or DSPs might be the appropriate solution. However, this work focused on embedded applications struggling to meet performance requirements or under resource or cost constraints. That is, the target scenarios are those where one (or few) custom circuit(s), coupled to a host processor, would suffice to achieve the desired performance increase. However, the

manual design of these systems is a difficult task for non-experienced software designers, and is also lengthy even for a joint effort by a hardware team and a software team.

Given this, and considering the type of target application, some form of HW/SW partitioning approach seems to be the ideal solution, but despite the existing approaches there are several issues with HW/SW partitioning that hinder its widespread adoption. Specifically, consider that: 1) software developers generally have no hardware design experience, even those familiar with embedded systems, so any partitioning flow must be as non-intrusive and require the least amount of hardware expertise as possible; and 2) the automatically generated hardware must be as specialized as possible, in order to improve performance while also being resource-efficient.

Some tools have contributed significantly towards these aspects, creating high-level IDE-based partitioning flows [Cal, Xilc, Syn]. These tools rely on powerful optimization steps to generate very specialized circuits, maximizing the resulting performance. Other approaches rely on binary code, and target a pre-designed configurable accelerator.

In other words, the first type of approach is geared towards circuit *specialization*, relying on generating hardware descriptions which are later processed by vendor synthesis tools; while the second relies on circuit *reconfiguration*, by mapping partitions onto a configurable peripheral. This has implications on a particular aspect: *accelerating multiple partitions* (either complete functions bodies or instruction sequences, depending on the approach).

The HLS approaches typically generate one custom circuit per function when targeting multiple functions; there is no re-utilization of resources between custom circuits. This might imply a considerable resource cost, which is especially wasteful if the custom circuits do not operate in parallel. Binary-level approaches have disadvantages as well: the co-processor hardware might be excessive for the partition(s) being accelerated; inversely, it might be impossible to accelerate a partition if the hardware resources are insufficient; the operating frequency might decrease with circuit complexity and reconfiguration capabilities, and the amount of required runtime reconfiguration information increases; finally, the performance of the accelerated partition(s) might be sub-par, since the selected partition(s) has (have) to adapt to the existing accelerator hardware, as opposed to generating custom hardware from the selected partition(s).

There is another aspect at play when relying on a reconfigurable circuit to execute multiple software partitions: the resource re-utilization, at a processing element level, is largely dependant on the similarity of the partitions. That is, a custom circuit with reconfiguration capabilities does not necessarily imply a good re-utilization of all processing elements it contains. A good analogy is the under-utilization of resources in VLIW processors [Liu08]. So the issue is not only one of resource re-utilization, at the processing element level, but one of total area savings.

That is, the trade-off in question is on how to provide resource re-utilization and area savings via *reconfiguration*, without compromising circuit *specialization*.

To summarize, this work considered: 1) automatic accelerator generation for (relatively) simple embedded applications for which one (or few) relatively small custom circuit(s), coupled to a

host processor, are enough to achieve the desired performance increase; 2) doing so without programmer intervention or tool chain interference; and 3) maximizing both the resource utilization and area savings of said accelerators by exploring reconfiguration mechanisms. Or, in a single sentence, this work developed a HW/SW partitioning approach whose central theme was:

How to automate the transparent generation of area-efficient specialized reconfigurable accelerators for embedded applications?

To address the issue, this work relied on an existing methodology to extract repetitive instruction traces [Bis12], and implemented the automated generation of accelerator hardware, its integration with the host system, and transparent migration of execution from processor to accelerator. Additionally, in order to make better use of hardware, the system has the capacity to reconfigure the accelerator resources at runtime in several ways. The following sections detail the specific objectives and the approach leading to this end result.

1.4 Objectives

The problem statement in the previous section already summarizes the major objective of this work, where two aspects are essential to the motivation. One is the transparency of partitioning to developers, but the most important is the generation of tailored reconfigurable circuits which efficiently accelerate critical portions of an application.

The following features contribute to the first aspect: avoiding modification of source or binary code (manual or automated), no manual hardware design, and no modifications to the host processor or otherwise manual integration of the accelerator hardware. Relative to the second aspect, to accelerate the demanding portions of the application, a selection must be performed, but in a way that does not compromise the first requirement.

The need for transparency led to relying only on binary information, specifically, *binary instruction traces*. Additionally, a requirement was set where the binary was to remain unmodified, pre- and post-deployment onto the target. It is obvious how avoiding the modification of source code contributes to the transparency, but modifying binary either during or post-compile would not interfere much from the point of view of the programmer. However, in order to capture the critical portions of an application it is important to rely on *traces* rather than static binary analysis. That is, the application must be monitored while it executes on the host processor, meaning the binary must remain unmodified. Also, not interfering with the binary avoids modified compilers or post-compilation tools. Finally, a modified binary would be bound to the accelerator-augmented system, making it impossible to deploy the same application to a software-only platform.

As for the generation of efficient accelerators, this relates to the explained trade-off between *specialization* and *reconfiguration*. The reconfiguration capabilities typically explored by binary approaches rely on configuration registers or similar to determine the functions performed by the accelerator circuit. Instead, efficient *reconfiguration* would be possible if the same hardware components, and therefore chip area, could be harnessed to execute multiple partitions, without

requiring additional control logic which is detrimental to *specialization*. That is, the issue would be resolved if the the reconfiguration mechanisms (or at least some portion thereof) were not themselves components of the accelerator circuit.

In short, the objectives for the developed work were too:

1. Design an accelerator architecture capable of:
   (a) Accelerating execution of traces by exploiting ILP and loop pipelining
   (b) Exploiting data parallelism by supporting concurrent accesses to data memory
2. Transform sets of frequent instruction traces into accelerator instances/configurations.
3. Design mechanisms capable of:
   (a) Transparently migrating execution from software to a custom accelerator instance
   (b) Reconfiguring the accelerator
   (c) Handling the data transfers between GPP and accelerator
4. Allow for accelerator reconfiguration, without compromising area savings or circuit specialization, by enhancing it with fine- and coarse-grain reconfiguration mechanisms.

In other words: the main objective was to accelerate the execution of one or more frequent *binary instruction traces*, using dedicated circuits with efficient reconfiguration capabilities.

To do this, a tool flow was developed which generates a tailored instance of a reconfigurable accelerator, without manual hardware design effort. The accelerator is coupled in a non-intrusive fashion to a host processor. According to the execution flow of the application at runtime, the accelerator is invoked to accelerate any one of the supported instruction traces. By relying on a runtime mechanism, the execution is transparently migrated from the processor the the accelerator, without modifying the executing binary. Finally, several fully-functional prototype systems were validated, using commercial FPGAs. Several sets of embedded benchmarks were used to determine the acceleration gains achieved via the automatically generated accelerators.

1.5 Approach

The developed approach to HW/SW partitioning and binary acceleration relies on four steps: 1) identification of candidate instruction traces, 2) translation of these traces into a custom accelerator instance, 3) detection of imminent execution of translated traces at runtime and reconfiguration of the accelerator, and 4) transparent migration of execution to the accelerator. The first two steps are performed by an offline tool chain, and the last two occur at runtime, using auxiliary hardware.

Figure 1.1 summarizes this concept. The identification and translation steps are based on a simulated execution step which detects frequently executing instruction traces called Megablocks

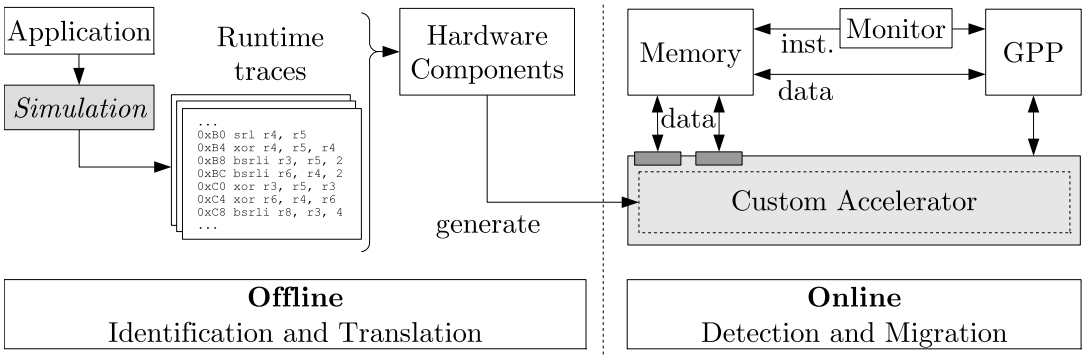


Figure 1.1: Proposed transparent binary acceleration approach

[Bis12], using an existing tool [Bis15]. The developed set of tools generates a specific tailored instance of the proposed accelerator architectures. Further details on the tool flow and detection steps can be found in Section 3.3, while the following section briefly explains the type of instruction trace this approach detects, translates and accelerates.

The right-hand side of Fig. 1.1 shows a generic architectural view of the accelerator-augmented system. The main idea is that a lightweight mechanism is capable of migrating execution of the translated instruction traces to the custom accelerator instance. The accelerator and GPP are capable of exchanging data and, to increase applicability, the accelerator supports memory accesses. Support for memory access by the accelerator and its advantages are discussed in Chapter 5.

The detection step refers to determining when the processor is attempting to execute any one of the translated instruction traces. This is done by a monitoring module which observes the instruction address. The migration stage is the process of preventing the execution of the trace from occurring via software, and instead invoking the accelerator, as well as handling the return to software execution afterwards. The migration step also involves reconfiguring the accelerator to execute the migrated trace. This is done in several ways, depending on the accelerator design. As per the objectives, this fulfils the task of transparently migrating execution between processor and accelerator, in a fashion that is explained in detail in Section 3.4.

1.5.1 Megablock Trace

Like the previous section explained, an important characteristic of the developed approach is that it relies on binary *traces* to capture the actual application workload. One type of trace that fulfils this is the Megablock [BC10, BC10, Bis12]. Megablocks are binary instruction traces which have one entry point and several exit points. Whereas basic blocks are delimited by control instructions (e.g., backward branches), Megablocks may incorporate multiple *branch* instructions, including backwards *branches*, and sub-routine return instructions. A formal definition of the Megablock can be found in [Bis12]:

"Let P be a static program formed by a sequence of instructions $[i_1, i_2, \ldots, i_m]$*; a trace*

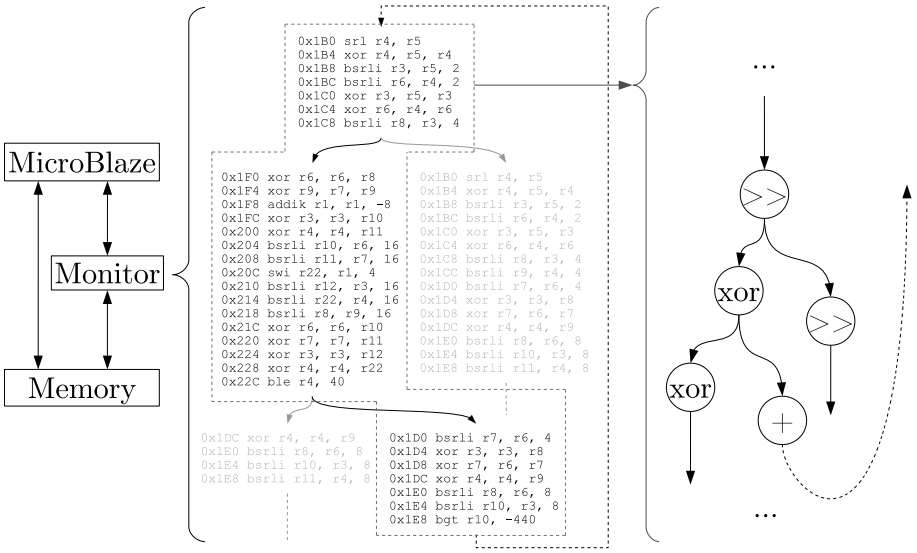


Figure 1.2: Representative synthetic example of a Megablock trace, and resulting CDFG

> *T is generated by executing the program, and is composed of possibly repeating instructions from P; let S be a sequence of m instructions where* $m > 1$*; a Megablock is a contiguous subsequence of T formed by repeated occurrences of S, and is represented as* $S\{n\}$ *where n is the number of times S repeats. E.g. let* $[i_4, i_5, i_6, i_4, i_5, i_6, i_4, i_5, i_6]$ *be a contiguous subsequence of T;* $S = [i_4, i_5, i_6]$ *and* $S\{3\}$ *is the corresponding Megablock."*

In short, a Megablock is a sequence of trace elements of smaller granularities (e.g., basic blocks). It may cross control flow boundaries, and represents a single repetitive path of execution, i.e., a *loop* path. Figure 1.2 shows a synthetic example of this, along with a Control and Dataflow Graph (CDFG) representation of the trace instructions. According to *branch* instructions, execution follows a particular path. Execution of this path repeats any given number of times, due to a backwards *branch* that leads execution back to the start of the trace. Execution of this particular path terminates when any *branch* executes in such a way as to follow a different path.

This type of trace provides information not available to a static binary analysis, or to a higher level execution profiler such as *gprof*. Also, in tracing the execution across several basic blocks, the size of the detected traces may increase, which means that a greater workload stands to be migrated to custom hardware. The focus of this work is on the automated generation of accelerators, and on exploring architectures to support the execution of these traces. So, although the trace profiling and CDFG extraction and optimization steps are vital, this work will not focus on runtime implementation of these features. To develop and validate the accelerator generation process, Megablock detection is performed offline, as the previous section stated. Section 3.3 explains the tool used to retrieve Megablocks from target applications, how it integrates with the developed accelerator generation tool flow, as well as the advantages and limitations of this type of trace.

1.5.2 Generating a Reconfigurable Customized Accelerator Instance

The generation of a specialized accelerator comprises the first two steps of the approach. Several accelerator architectures are presented in this document, but the overall methodology is the same: the accelerator is a hardware template whose features, such as Functional Units (FUs) and interconnections, are automatically customized by tools which translate one or more CDFGs into a set of Verilog parameters. Each generated accelerator instance is thus tailored for a specific set of CDFGs, and is capable of accelerating each one in a time-multiplexed fashion.

The details of the translation of Megablock CDFGs into accelerator instances vary with the accelerator architecture, but the general flow is discussed in Chapter 3. Keeping in line with the design rationale of the approach, the generation of custom accelerator instances happens without developer effort, and at a post-compile stage.

Like the objectives proposed, the balance between *specialization* and *reconfiguration* hinges on the ability to control the accelerator circuitry at several levels. So this work was based on the notion of an accelerator architecture which could be reconfigured at two different granularities.

1.5.2.1 Reconfiguration at the Functional Unit Level

As mentioned before, the kind of accelerators targeted by binary acceleration approaches usually contain visible configuration registers to control the interconnections between computing elements. Instead of targeting a pre-designed accelerator whose reconfiguration capabilities are fixed, the developed approach generates the entire accelerator structure, including reconfiguration logic.

The reconfiguration of the accelerators is essentially the control of data between FUs via specialized multiplexers. That is, the accelerators are configurable data paths. The implementations in Chapters 4 and 5 are reconfigured prior to trace execution. The implementation in Chapter 6 is more complex, and requires per-cycle control of its FUs and storage elements.

Generating custom interconnectivity is already a solution towards not compromising specialization. However, as the number of CDFGs to support increases, the amount of accelerator resources required increases, as well as the FU interconnection complexity. This leads to an increase in required area an potentially a decrease in operating frequency. Also, the efficient re-utilization of computing resources is not assured, despite the tailored interconnect, since it depends on the similarity between CDFGs. The second reconfiguration mechanism addresses these issues.

1.5.2.2 Module-Level Reconfiguration

The main issue with supporting too many configurations, even in a specialized accelerator instance, is that the reconfiguration logic becomes excessive. So the solution is to reconfigure the accelerator without using in-module logic to do so. This work relied on the use of Dynamic Partial Reconfiguration (DPR) to implement this functionality [Xil12b]. The capacity for DPR is an architectural feature of some SRAM-based FPGAs which allows for modification of a predefined area of the chip's circuity without powering down the device.

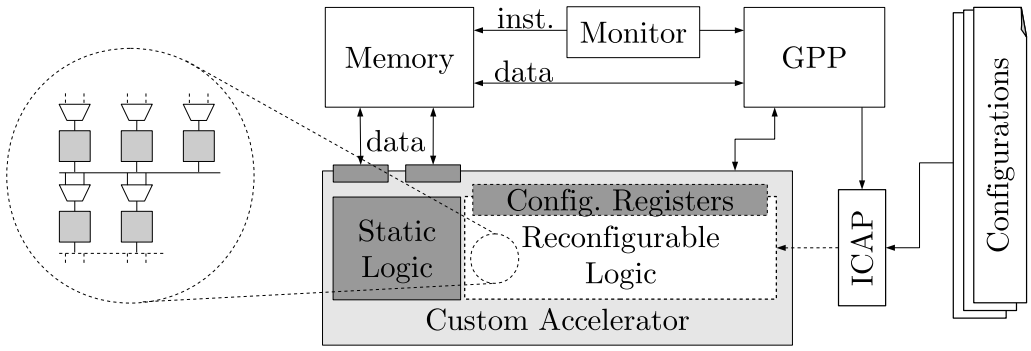


Figure 1.3: DPR Oriented system design

Essentially, given a user-defined area of the device, it is possible to define several circuit configurations for it, and at runtime switch between them according to design or application criteria. Although relatively old (some examples include the Xilinx XC6200 and the Virtex-II Pro), DPR is a yet under-explored feature of FPGAs. However, some academic implementations exist which rely on DPR, targeting very distinct application domains. For instance, self-configuring filters [LPV10], fault tolerance via DPR [ESSA00], image applications [LP13, SKK15], or communications [LFy09, Dun13]. In order to modify the circuit connections at runtime, Xilinx FPGAs rely on an Internal Configuration Access Port (ICAP), which can be accessed through software or from user-level logic. Each configuration of a dynamically reconfigurable area is stored in a *partial bitstream* file which is written to the FPGA's configuration memory via the ICAP.

Figure 1.3 shows the system architecture again, this time detailing the two reconfiguration mechanisms. The accelerator is segmented into instance independent-static logic and a reconfigurable portion. The former includes memory ports and interfaces with the processor. The later includes all FUs and configuration registers which control the interconnection logic. The circuity in this region is switched in and out as a whole via DPR.

Two use cases can be met by relying on this feature. A possible approach is to take a set of candidate Megablock traces and for each one generate an accelerator instance. Each instance is stored in a *partial bitstream* file. Fully dedicated instances like this could potentially reduce interconnectivity complexity relative to a multi-configuration accelerator. However, this does represent an additional cost in terms of memory to hold the several bitstream files and, more importantly, an additional ICAP reconfiguration overhead for every loop call to accelerate. So, to alleviate this time and storage overhead, several CDFGs can be used to generate a single accelerator instance which is controllable via context registers and multiplexers.

The more appropriate strategy depends on the workload of the target application, and to some extent on the target devive. In a device with less resources, partial configurations supporting only one CDFG could be more appropriate. If the device is capable, then a single larger instance capable of executing several Megablocks could be used. Alternatively, several multi-loop accelerators could be switched according to the application. Essentially, the same application can be accelerated in different ways without need for recompiling.

1.6 Contributions

Basing the automated generation of accelerator hardware/configurations on binary information allows for embedded applications to be accelerated without the need for manual hardware design, and without need to re-target the application to execute on the resulting heterogeneous system. This facilitates future revisions to the application, since accelerator hardware can easily be re-generated to target new versions of the binary, which increases adaptability and applicability. Because execution times are shortened through hardware execution, and because of the specialization of the accelerator hardware, power savings may also be achieved.

However, as is later summarized in Chapter 2, existing binary acceleration approaches suffer from some limitations. Many approaches accelerate only small portions of code, detected by either a static or runtime analysis. Binary modification is in some cases used to aid either the runtime migration of execution, or the supporting tools. The target accelerators are usually fixed arrays of FUs, so the resulting performance depends on the computing capabilities, which are not precisely tailored for the detected partitions. Support for memory access is especially important to exploit data parallelism, but is usually non-existent or limited. Finally, accelerators are either intrusively integrated with the host or suffer from large communication overhead if loosely coupled.

This work relies on instruction traces, over a static binary analysis, to target the portions of the application which represent the bulk of the workload. Megablock traces are not delimited by a single *branch* instruction, and necessarily represent frequent *loop* paths. To properly exploit potential parallelism in loops, loop-pipelining is implemented. Support for concurrent memory accesses is addressed also, being necessary in itself to support data-parallelism within one iteration, but becoming more important when the access contention increases due to pipelining iterations. Also, the developed approach allows for the binary to remain unmodified, both during partitioning and during runtime, by relying on a transparent migration mechanism. Generating an accelerator with dedicated reconfiguration capabilities allows for acceleration of multiple partitions without completely sacrificing circuit specialization. Tailoring the accelerator hardware to the detected trace helps towards optimizing the trade-off between resource requirements and performance, and potentially power consumption as well. Also, since the accelerator is customized, a candidate trace will not have to suffer a performance decrease or be discarded due to lack of FUs.

In summary, by completing the objectives through implementation of the proposed methodology and architecture, this work extends the state of the art in the following aspects:

1. Transparent acceleration of unmodified binary
2. Generation of *specialized* multi-loop accelerators
3. Joint exploitation of *data-parallelism* and *loop-pipelining*
4. Accelerator *reconfiguration* mechanisms for resource re-utilization and area savings

This work is a continuation of preliminary research done for an M.Sc. thesis [Pau11]. In the referenced publication, work was conducted regarding the transparent runtime migration of

execution between software and hardware. Preliminary results were attained by using custom tools to translate a set of CDFGs into Verilog specifications of a runtime reconfigurable accelerator. The system was capable of transparently migrating execution between software and hardware, utilized external memories, and did not support accelerator memory access. The target platform was a commercial FPGA, and the system was tested with 7 benchmarks, 2 of which generated accelerators with 6 runtime configurations. Achieved speedups ranged from 0.2× to 65×.

The work presented here expanded the existing implementation regarding support for memory access, more efficient accelerator architectures, exploiting loop pipelining, and supporting floating-point operations. Extensive on-chip validations of the systems were performed, using vendor FPGA boards and several sets of benchmarks. Chapter 3 summarizes the work by providing an overview of the accelerator designs and supporting tool flows. The following section summarizes the publications regarding this work and focuses on their individual contributions.

1.7 Summary of Published Work

In the context of this work the following publications, by type and order of submission, have been produced:

1.7.1 International Journals

1. *Transparent Trace-Based Binary Acceleration for Reconfigurable HW/SW Systems* [BPFC13] - This paper, published in the *IEEE Transactions on Industrial Informatics*, further expands on the system by presenting results for a set of 16 benchmarks and speedup estimations for a point-to-point connection based architecture for a set of 62 benchmarks which include memory operations. Speedup estimations average 2.7×.

2. *Transparent Runtime Migration of Loop-Based Traces of Processor Instructions to Reconfigurable Processing Units* [BPCF13] - An extension of [BPCF11], published in the *International Journal of Reconfigurable Computing*, compiles results of 17 benchmarks for three different system architectures which utilize the same accelerator architecture and tool chain. The three architectures combine two interfaces between GPP and main memory (local or external memories) and GPP and accelerator (bus or point-to-point). The architecture using local memory and point-to-point is an implementation of the architecture conceptualized in [BPFC13]. In this architecture, the accelerator is reconfigured in parallel with data communication. Optimizations were performed to the accelerator to handle values of up to one previous iteration. The tool chain has many optimizations and new features. A method to detect CDFGs online via dedicated pattern detection hardware is proposed. Speedups for the architecture suffering from least overhead range from 1.26× to 3.69×, which comes close to the best possible achievable speedup for the implemented parallelism. Speedup estimations using the previously mentioned formula incur an average error of only 1.75 %.

3. *A Reconfigurable Architecture for Binary Acceleration of Loops with Memory Accesses* [PFC14a] - Published in the *ACM Transactions on Reconfigurable Technology and Systems* as an extension of the work published in [PFC13]. A more efficient interface between accelerator and GPP is employed. The placement of load/store FUs is optimized in order to decrease the number of cycles during which execution is waiting for completion of memory accesses. A static memory access scheduling mechanism further decreases the waiting time. A mean geometric speedup of 1.71× is achieved for 37 integer benchmarks.

Additionally, a paper entitled *Loop Pipelining in Customized Accelerators for Transparent Binary Acceleration* has been submitted to the *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. This work vastly expands the results attained with a single-row modulo-scheduled accelerator first introduced in [PFBC15]. It is evaluated with a larger set of benchmarks, including floating-point kernels, using a more sophisticated software test harness. Additionally, the full custom accelerator instances are compared with ALU based accelerators in terms of performance/resource trade-off. Finally, several VLIW models are simulated and compared against the proposed accelerator design, including the ρ-VEX reconfigurable VLIW. Versus a single MicroBlaze, the geometric mean speedup for 13 floating-point kernels is of 6.61×, and for 11 integer kernels, it is of 1.78×. Compared to a 4-issue VLIW, this value is of 1.78×. It is also shown that this accelerator design is more efficient in terms of resources, versus previous implementations.

1.7.2 International Conferences

1. *From Instruction Traces to Specialized Reconfigurable Arrays* [BPCF11] - Published in the *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*. This work builds on the work presented in [BPFC13] by presenting estimations of speedups attained by using the same tool chain but with an improved hardware architecture, using a local memory interface for the GPP's program code. Speedup estimations range from 1.0× to 2.0×. This publication includes work relative to the Megablock trace, detailing how a Megablock is composed and detected.

2. *Architecture for Transparent Binary Acceleration of Loops with Memory Accesses* [PFC13] - Published in the *Proceedings of the 9th International Symposium on Applied Reconfigurable Computing*. This work explores the issue of supporting an accelerator with arbitrary memory accesses. The memory access enabled system is built on the local memory variant of the hardware/software partitioning system in previous work. The Megablock detection and accelerator generation tool chain was reworked to produce a tailored accelerator that supports up to two concurrent memory accesses to the local dual-port data memory. By using a memory sharing mechanism, both GPP and accelerator can access the program data, without incurring any data transfer overhead or introducing any synchronization issues. For 17 benchmarks, a maximum potential speedup of 2.04× is possible.

3. *Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support* [PFC14b] - Published in the *Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing*. In this paper, the accelerator architecture is re-designed to allow for simultaneous activation of multiple computing stages, thereby implementing loop pipelining. It was also adapted to deal more efficiently with higher memory access latencies. The design was evaluated using a system in which data resides completely in external memory, so as to test the applicability of the approach on larger applications. To cope with this, a configurable dual-port cache was designed to augment the accelerator. For 12 benchmarks, a geometric mean speedup of 1.91× was achieved.

4. *Transparent Acceleration of Program Execution Using Reconfigurable Hardware* [PFBC15] - Published in the *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. This publication summarizes results obtained with the multi-row accelerator architectures, and presents early results using a single-row modulo scheduled accelerator design. Additionally, the proposed approach is compared to the performance of Xilinx's Vivado HLS flow. For 12 benchmarks, the geometric mean speedup for the most efficient proposed architecture was of 5.44×.

1.7.3 National Conferences

1. *Generation of Coarse-Grained Reconfigurable Processing Units for Binary Acceleration* [BPCF12] - Published in the *Proceedings of the 8th Portuguese Meeting on Reconfigurable Systems (VIII Jornadas sobre Sistemas Reconfiguráveis)*, compares the speedup results of 17 benchmarks for the architecture developed in [Pau11] and an actual implementation of the proposed architecture introduced in [BPCF11]. The tool chain is explained in detail and an accurate speedup estimation formula is presented. Speedups attained for the local memory based architecture deviate slightly from the estimations presented in [BPCF11], but are within expected values.

2. *Transparent Binary Acceleration via Automatically Generated Reconfigurable Processing Units* [PCF15] - Published in the *Proceedings of the 11th Portuguese Meeting on Reconfigurable Systems (XI Jornadas sobre Sistemas Reconfiguráveis)*. This paper goes into further detail on the 1D accelerator design and the modulo scheduling of Megablock traces. The comparison with previous approaches is this time more thorough, as up to 37 benchmarks are used to evaluate the systems.

1.8 Structure of this document

The remainder of this document is organized as follows: Chapter 2 details the general characteristics of co-processors, summarizes related binary-level partitioning approaches as well as other similar works; Chapter 3 gives an overview of the developed approach, detailing the overall partitioning approach in terms of tool flow and architecture of all implemented systems; it also contains

more detailed summaries of the following three chapters. Chapters 4 to 6 each present a different accelerator design iteration, as well as the experimental evaluation performed. Chapter 7 explains how DPR augments the accelerator architecture of Chapter 6 with reconfiguration capabilities which help to maintain circuit specialization in multi-configuration accelerators without greatly increasing resource requirements. Each chapter also presents the translation tools targeting the particular accelerator design, as well as the system-level architecture. Finally, Chapter 8 presents the conclusions, future work and possible approaches to address it. Additionally, in Appendix A, a proof-of-concept for external memory access support by the accelerator is presented.

Chapter 2

Revision of Related Work

Research regarding runtime reconfigurable systems and hardware/software co-design spans over two decades [CH00, Nag01, Har01, Wol03, SML09, SBFC10]. Although the first notion of a runtime adaptable machine has existed since the 60's [Est02], an architecture capable of autonomously generating or modifying hardware at runtime to suit execution needs has yet to be developed. The difficulty rests on finding a consistent, scalable and flexible methodology or runtime algorithm that could, potentially, generate hardware as efficient as a custom design.

To provide for transparent runtime acceleration, approaches have been proposed that avoid the need for per-application basis hardware accelerator design by automatically generating hardware specifications of, or configurations for, dedicated accelerator hardware. Such peripherals are used to execute demanding portions of applications, typically data-oriented loops. Speedups are achieved by exploiting the Instruction Level Parallelism which is expressed by CDFG representations of said application partitions.

2.1 Overview

Existing approaches employ accelerators that are capable of varying degrees of reconfigurability and target the execution of different types of CDFGs. Usually, an accelerator is an array of FUs, either in a mesh type or row-based arrangement, with a configurable interconnection structure. The following are the main characteristics of such co-processors:

1. Type of accelerated application partition and method for its detection and translation;
2. Interface between accelerator and host system;
3. Granularity, arrangement an connections between Functional Unit (FU) of the accelerator;
4. Execution model of the accelerator;
5. Support for accelerator memory access;
6. Runtime control of the accelerator.

The following sections detail these properties and refer to implementations which exemplify different design choices. Most approaches implement these systems in FPGAs, although ASIC approaches have also been proposed [SLL+00, But07, LV04]. Approaches usually assume that accelerator and General Purpose Processor (GPP) reside on the same chip. FPGAs however seem to be the ideal target platform in which extensive design space exploration can be performed, both for purposes of research into such systems, and later flexibility of deployment of efficient HW/SW partitioning systems [Wol03].

2.1.1 Partitioning

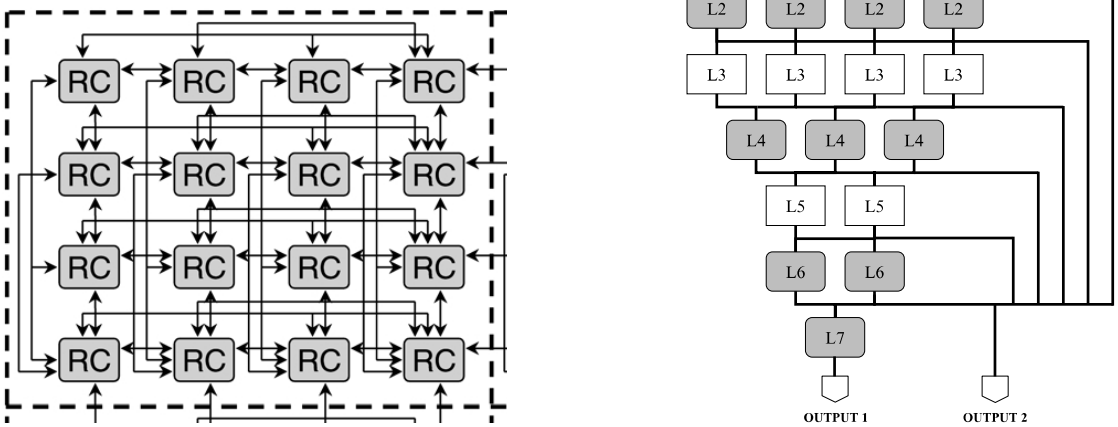
Performance improvement of an application through accelerators is achieved by first choosing or detecting demanding computation to be accelerated. Some approaches rely on forms of High Level Synthesis (HLS), i.e., offline stages involving manual source code analysis or use of tools to identify candidate functions [HW97, HFHK04, WH95, BFM+07]. A translation of high-level code is performed, to automatically generate Hardware Description Language (HDL) code, followed by standard logic synthesis. This may require modifying source code to allow compatibility with HLS tools, or later modification to allow for integration with the accelerator hardware.

In contrast, the approaches addressed throughout this chapter rely on binary information. Approaches of this kind analyse the compiled application, and provide a transparent use of custom hardware for the application programmer. Binary acceleration focuses on exploring Instruction Level Parallelism (ILP), by representing the target binary instruction sequences as CDFGs.

Binary analysis can be performed over the static binary or over the executing instruction trace. The latter makes more sense as an offline strategy. The binary can be analysed with custom compilers and modified either during or after compilation [CBC+05]. The former is appropriate for runtime implementations, as the instruction trace can be directly observed [LV09, BRGC08].

Observing the executing instruction stream provides additional information which is not found by static binary analysis. Due to this, approaches which perform CDFG detection offline also resort to simulated execution to extract trace information [NMIM12, BPFC13], but it is not guaranteed that simulated execution matches the behaviour of the application in-system (e.g., different volumes of incoming data, or event-based execution). This is also true for systems which perform discovery at runtime for only a limited time after program startup. If the application behaves sporadically, it is incorrect to assume that a predetermined amount of profiling after the start of execution will be representative of the execution demands for later periods. A better approach is to employ periodic or constant profiling [LV09].

Runtime binary translation provides even greater transparency, because use of any custom tools prior to deployment is avoided. Unlike HLS, binary translation it is more suited for runtime implementations because binary is more easily parsed, making it more tractable for embedded tools. Despite the greater transparency, runtime implementations imply additional overheads. Hardware is required to retrieve traces and perform analysis, and depending on the architecture, temporal overheads may be introduced during the execution of the application [LV09, RBM+11]. Because accelerator configurations are generated in-system, the embedded translation steps must



(a) Mesh arrangement for Functional Units used by the MorphoSys array (adapted from [SLL+00])

(b) Row based Functional Unit topology employed in the CCA approach [CKP+04]

Figure 2.1: Two different Functional Unit arrangements for reconfigurable arrays

be robust in order to ensure functional correctness of the accelerated regions. Also, the tasks of binary analysis, CDFG detection and translation are themselves computationally demanding. As such, runtime binary translation will be less powerful, detecting smaller kernels and performing fewer optimizations. Even the minimalist implementation of runtime synthesis used in [LV09] requires over 5 MB of data memory.

Trace elements targeted for acceleration include basic blocks, sequences of forward-jumping basic blocks [CKP+04], traces which support backwards jumps and multiple exits [BPFC13, BC10] or traces with multiple paths [NMIM12]. These elements represent frequent sequences of instructions (which some approaches compile into custom instruction), iterations of frequently executing loops, or common paths of execution throughout the iteration of a loop.

2.1.2 Accelerator Structure

An accelerator is typically composed of interfaces to the host system, and a region in which operations are actually performed. Operations can be implemented through fine-grained bit-level logic, or via coarse-grained FUs such as adders or multipliers, or even more specific functions such as square root modules or Multiply Accumulate (MAC) blocks [LV04, ECF96, GSB+99]. Commonly, accelerators contain an array of such FUs. In the later case, the term Coarse Grained Reconfigurable Array (CGRA) is often used. Three major features characterize an array: number and type of FUs, their arrangement and the their interconnection capability. Figure 2.1 provides two examples of FUs arranged into different array topologies. A comprehensive look at array architectures is given in [Har01].

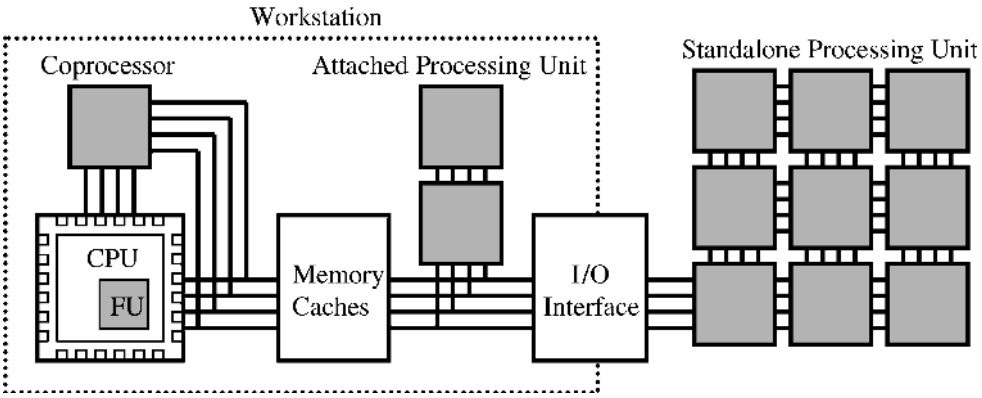


Figure 2.2: Different types of interfaces between host processor and co-processor [CH02].

Using a co-processor implies an exchange of information between it, the GPP and in some cases a shared or main memory. If an inefficient interface is used, the introduced overhead may negate the speedups obtained by acceleration through hardware. Figure 2.2 shows common possible methods of attaching a co-processor to a host system.

Some approaches couple the accelerator directly into a processor's pipeline, which makes it easier to operate on values directly in the processor's register file [BRGC08, NMM+06, CKP+04]. This allows for a very low-overhead interface between the accelerator and the host processor. However, such approaches are limited by the reduced portability and potentially require a synchronized co-design effort as GPP architectures themselves evolve. Placing new logic in the processor's pipeline could also introduce critical path delays. Additionally, memory accesses by the co-processor become more difficult to implement, because it is embedded into the pipeline, providing no obvious manner to interface with data memories or allow concurrent accesses.

Alternatively, co-processors may be loosely coupled as peripherals [SLL+00, BPCF13, LV04], using interfaces such as buses, dedicated links or shared memory schemes [PPM09]. Although loose coupling may introduce larger overheads in the communication with the GPP, development of the accelerator does not require intrusive modifications to the host processor. Because the accelerator is an external peripheral, design becomes less constricted. Shared memories or DMA can be used to allow for more sophisticated memory access by the accelerator.

2.1.3 Accelerator Functional Units

FU Capabilities The operations supported by the FUs depend on the targeted CDFGs. For instance, the approach shown in [NMIM12] uses an array with no support for multiplication, prohibiting the acceleration of CDFGs containing even one multiplication. Tightly coupled arrays are usually smaller, containing simpler FUs supporting integer arithmetics and logic, but usually not memory operations. Larger arrays are typically loosely coupled, employing more sophisticated FUs, which support more complex operations, potentially allowing for a wider scope of acceleration. This however, also depends on the type and size of region of code to accelerate [CH00].

Although arrays containing larger numbers of FUs with greater capabilities are more flexible, significant hardware overhead could be introduced depending on the operations to support. Some arrays are composed of numerous Arithmetic and Logical Units (ALUs) with dedicated register files, but even these cases do not include support for division operations and floating point arithmetic [RBM+11]. Some implementations find a compromise with heterogeneous arrays, using larger numbers of simple FU and a small number of more dedicated resources, arguing that complex operations are less frequent [NMIM12].

FU Arrangement The manner in which data flows to/from the array is a function of the layout of its FUs. The layout is based on how we wish to implement the latent parallelism of the accelerated code. The most representative approaches relative to the proposed work employ either row or mesh based topologies of word-level FUs.

A mesh type arrangement is shown in Fig. 2.1a. This type of arrangement is characterized by typically homogeneous FUs layed out in matrices which are richly interconnected. FUs can send and receive operands and results from the four nearest neighbours, and some FU might contain their own register files. This arrangement and connectivity does not enforce any data directionality; data can usually travel towards any direction in the array. Connectivity can be increased by also connecting FUs diagonally or with long lines for distant connections. In [BGDN03], a performance analysis is done of mesh arrays in function of connectivity and FUs capabilities.

Figure 2.1b shows a row arrangement for two types of FUs. FUs in one row execute concurrently and propagate data downwards to adjacent rows. This type of arrangement closely mimics the layout of CDFGs themselves, enforcing a feed-forward connectivity. Data can then be fed back to the first row, if the array is meant to be used as a loop accelerator [CHM08, BPCF13], or the connections may be strictly feed-forward. Between rows, there can be a crossbar (all-to-all) connection, or simpler multiplexers. Some arrays also allow connections spanning multiple rows [BPCF13, CBC+05, NMM+08]. This allows for greater applicability as typical CDFGs are not composed solely of nodes whose connecting edges have a length of 1, i.e., connect to adjacent nodes. There are also row arrangements with a single-row, whose configuration is changed every cycle, thus repeating a sequence of configurations composing an iteration [GSB+99, CFF+99].

Mesh arrangements have an easily scalable structure, and because of the homogeneous structure and capabilities of the FUs, are more generic than row arrangements. The latter tend to be more heterogeneous, distributing processing capabilities through stages where the designers determine to be necessary. Meshes are usually employed for loosely coupled arrays, while the data directionality of row-based arrays seems to be more appropriate for a tight integration into processor pipelines, although they can also be used as loosely coupled loop accelerators [CHM08, BPFC13].

One could argue that mesh arrays appear more flexible than row topologies. But better scalability and larger FU complexity does not ensure that more code will be successfully mapped to the hardware. Speedups depend on the maximum ILP that can be found in the traces, on the type of accelerated trace, the trace detection method and on the efficiency of the translation of traces into array configurations. Smaller row-based architectures geared towards streaming execution

can be suficcient to sucessfully accelerate the target CDFGs, depending on the number and type of operations, ILP, graph depth and especially complexity of the connections.

There are also arrays that employ a mesh arrangement of much more sophisticated processing units like small processors, which is a different category of heterogeneous systems. Acceleration can be achieved by having each core execute independent iterations of a loop, or multiple unrelated loops can be executed in tandem. For these cases, the notion of interconnection is intrinsically different, as data is exchanged at a higher level. In [Ima12], many processors communicate by local handshaking, and each is controlled by a dedicated instruction stream.

FU Interconnections As mentioned, it is the possibility of controlling data exchange that grants great flexibility to CGRAs, more so than the total number of resources or their complexity. Interconnection capability is especially influential for the performance of an array.

A richer interconnection scheme typically allows for a better use of the available computing resources. However, the richer the interconnect, the more configuration information will have to be generated and, depending on the approach, stored in memory to be loaded prior to execution, leading to increased memory requirements. Also, complex connection grids like crossbars and multiple buses can introduce critical path delays and require considerable resources. Meshes are more efficient if they are precisely tailored for the required operations and connectivity of the CDFGs to execute. Quantitative analysis of a reference set of CDFGs can be employed to determine these requirements [SNS+13, YM04, NMM+06, NMM+08], but might lead to an over-fit, making it hard to predict performance for other CDFGs.

Existing approaches experiment with different interconnections, often introducing small design changes whose effect can be difficult to quantify because the mapping sucess rate or achieved speedup also depends on the type of CDFG being mapped, and on other aspects of the array. Different interconnection topologies are evaluated in [WKMV04, VEWC+09, MSB+07].

Because mesh arrangements are homogeneous, the same interconnection capabilities will be found throughout the mesh. This may prove to be excessive because not all nodes of a CDFGs require the same amount of connectivity. As such, designing a mesh interconnect to satisfy the maximum possible connectivity can lead to under-utilization of available connections. Also, node connectivity on a CDFG tends to diminish with depth, i.e., operands are consumed as the graph executes, meaning there is a larger number of nodes and connections in the earlier levels.

Row topologies mitigate this due to the directionality of the connections. Consider a square mesh array, and a square row based array. FUs of the mesh array are connected to their 8 neighbours bidirectionally. FUs of the row array are connected through inter-row crossbars. It is evident that as the arrays scale, all things being equal, the amount of connections increases more rapidly for the mesh case. Also, for a row based topology, connection capabilities can be tailored on a per-row basis, further decreasing resource usage at the cost of decreased applicability.

2.1.4 Accelerator Memory Access

Support for memory access is a requirement to achieve large speedups. Potential speedups are higher for frequently executing kernels that contain numerous memory accesses, since there might be large amounts of latent data parallelism. Ideally the array should be able to access all the application data and perform as many concurrent accesses as possible. However, support for memory accesses in these scenarios is usually an issue. The data needs to be shared efficiently between host processor and accelerator, which might imply the use of shared caches or data-transfer steps to synchronize data. Also, the array needs mechanisms to perform memory accesses, ideally several in parallel, which means a sophisticated memory layout is required.

If local memories are used for the array [KLSP11, PPM09, SLL+00], the issue of synchronization with the GPP arises, as large amounts of data may need to be moved between the local memories and the main data memory, and the overhead would negate parallelism gains. To avoid this, DMA access can also be used, keeping in mind coherency between accelerator memory, main memory and processor data cache. Despite the overhead, using local memories remains the most straightforward method. It is employed, for instance, by commercial HLS tools [Cal], which instantiate local memories for the automatically generated hardware modules. The size of the memories is determined by source code analysis of explicitly declared static array dimensions.

Shared memories can be accessed by multiple master devices. One approach is to populate the memory at runtime prior to array execution. Thus it behaves as a local memory which the processor accesses directly [SLL+00], unlike local memories built into the accelerator, which would instead fetch data through DMA, for instance. This avoids the need to include more complicated memory access logic within the array so it populates its own local memories.

An alternative is to create a system which maps accesses to a given address range to a shared memory, avoiding lengthy transfers steps. The shared memory can be used to accomodate the entire address space of the application [PFC13, KHC11, LV09] or just a defined range [KHC11]. Data produced by the array is placed back into the shared space and then accessed by the processor. However, this involves determining appropriate memory ranges. Also, if the data accesses have a very small locality the choosen range will be insufficient, as the array will frequently try to access data outside the range. It is not straightforward to provide support for a shared memory onto which several non-sequential ranges are mapped, as this might imply compiler or linker modifications.

In [KHC11] a shared cache is used, supporting any one runtime defined range. The shared memory is placed at cache level and shadows the processor cache or main memory. The processor transparently accesses either the cache, or the shadow memory, depending on which has the up-to-date data. Writing produced data back to main memory is not required. However, if data required by the array prior to execution are not in the shadow memory, a transfer step is needed. Also, this approach employs a custom cache architecture for the processor, and does not support a closed third-party processor with integrated caches.

Regardless of memory layout, concurrent accesses are required to fully realize parallelism potential. If not, waiting for sequential accesses will introduce additional delays that negates

speedups. To avoid this, the chosen memory architecture can be further specialized.

For instance, in [LV09], a dual-port memory is used. One of the ports is reserved for the processor, while the second is used by the array, allowing for one access per cycle at most when executing on the accelerator. In [KLSP11, DGG05] a modulo scheduling approach is optimized so that conflicting memory accesses are not issued simultaneously. Additionally, the work in [KLSP11] studies the distribution of data through several local single-ported memory banks accessed by a generic mesh array. Data access is first profiled and the arrays are interleaved throughout several local memory banks. Similar to the shared memory scenario, this approach requires a compile-time analysis to determine an adequate mapping of arrays into the several memory banks, based on observing array sizes and access frequency. Despite this, it is an efficient way to provide multiple concurrent accesses to data, as more memory banks can be added. Alternatively, several single-port memories can be used, where each corresponds to a sub-range of adresses. However this means concurrent access to addresses in the same range still collide. Data could be copied into several memories, but this would introduce synchronization issues. By distributing out data into different banks, greater concurrency is possible, but this does not guarantee that conflicting accesses will not occur, i.e., the array may have to access several data in the same memory.

Another issue is the supported access pattern. The simplest approach is the use of address generator hardware which outputs addresses with a specific stride [LV09, AD11]. Also, the range of addresses is typically determined at compile time. This precludes accessing heap allocated data or random access patterns. Some HLS approaches only support analysis of memory accesses using explicit array based syntax with a stride that can be determined statically. In [BRGC08] access to random addresses is supported giving the array's load/store FUs ability to receive results from any other FUs to use as addresses. A source-to-source transformation is presented in [AMD13], to disambiguate memory accesses for HLS approaches thus better predicting accesses.

Approaches, with local shared memories, or other schemes of distributed memory, are appropriate for loosely coupled arrays. It is more difficult to support memory access in tightly coupled arrays. However, in [NMIM12], up to 1 store operation is supported by using the writeback stage of the processor pipeline. In [BRGC08], concurrent load accesses are supported, but implementation details are not given.

2.1.5 Accelerator Execution Model

How accelerator execution is controlled depends on the arrangement of FUs and how they are interconnected. This determines how data is fed and moved between FUs. A typical processor follows a Single Instruction Single Data (SISD) model while these types of accelerators exploit Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) paradigms.

The distinction is blurred, as a single stream of instruction words which configure several or all FUs in the array are in fact providing multiple instructions, one per FU, as in a VLIW processor. But, from the point of view of the GPP, which uses the array as a peripheral or custom pipeline FU, a single instruction stream is issued. If the FUs themselves are active components, autonomously fetching instructions from memory, this can be considered a MIMD implementation.

Accelerators containing ALUs can be considered MIMD, since ALUs in the array will perform a different functions. Even if the entire array is configured by a single word prior to execution, or even if a command is given per cycle to configure the ALUs, unless all ALUs are performing the same operation over multiple data streams, it cannot be considered a single instruction stream. We can instead look at how CDFGs operations and their data flow are implemented in the arrays.

Loop Acceleration vs. Sub-graph Acceleration Array execution can target cyclical or acyclical CDFGs. Cyclical CDFGs refer to graphs with backwards edges to earlier nodes. That is, a graph represents a complete loop iteration, which loosely coupled arrays, mesh or row based, usually target. The backwards edges determine the Initiation Interval (II) of the graph. An array can execute an iteration of a cyclical graph in a number of cycles equal to the II. That is, one graph iteration is completed every II cycles. Arrangements with multiple rows rely on mimicking the data directionality of the CDFGs, and mesh arrangements on scheduling loop iterations, under the existing resource and connectivity restrictions. Acyclical execution means that data is not sent back to previous nodes, which applies when considering smaller sub-graphs which represent a sequence of instructions in the binary code (or trace). They may be part of a frequent loop, but do not constitute the entire loop body. A tightly coupled accelerator serving as a custom GPP pipeline unit is the most appropriate for these cases.

Iterator Control For loosely coupled loop accelerators, the number of iterations to perform on the array can be a compile time constant, be determined at runtime by input data. Typically, HLS methods are incapable of inferring complex dependencies between data and control, i.e., dynamic data-control dependencies. As such, they usually support translation of loops with statically defined iterators. Binary translation methods suffer from the same problem, static binary analysis contains even less information about loop iteration, and trace profiling does not ensure the loop behaviour will be constant *ad eternum*. Arrays with more sophisticated control keep track of iterator values or evaluate termination conditions for iterative execution.

Tightly coupled arrays which are activated explicitly by the processor on a per-iteration basis do not suffer from this issue. Since they are not explicitly loop accelerators, some of the sequential code responsible for iteration control will still be executed on the processor. This is partially related to support for conditional execution on the array. CDFGs with conditional paths have to be broken down, leading to smaller accelerated CDFG. In [NMIM12] however, the array has support conditional execution, accelerating frequent sequences of basic blocks.

Execution in Mesh Based Arrays In a mesh arrangement with homogeneous FUs, a single configuration command could suffice to select an operation to be performed per FU as well as defining the interconnects of the entire array. Execution then starts, and the same calculations are repeated, which corresponds to iterative execution of a CDFG. This assumes the mesh has sufficient resources onto which every CDFG operation can be mapped. In constrast, ALUs in a smaller mesh can perform a different operation every cycle. A CDFG is split into stages which

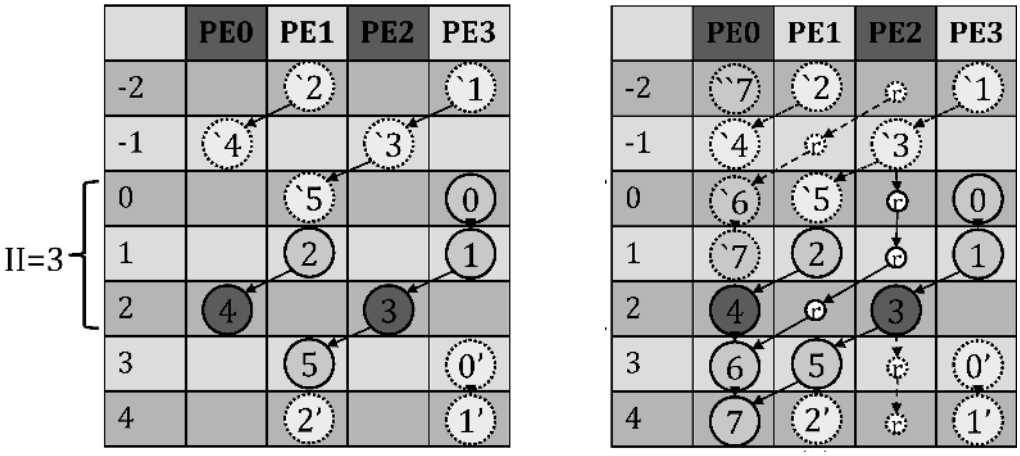


Figure 2.3: In [KLSP11], a memory-aware modulo scheduling is performed on mesh arrays. On the left, a partially complete mapping for a graph of 8 nodes onto 4 FUs. On the right, several superpositions of the complete schedule.

execute cyclically on the array. This requires instructions, and possibly data, to be fed to the array at every stage. In [SLL+00] it is the host processor that performs this task, while in [PPM09, But07] there are local instruction memories residing in the array. By folding the schedule of operations on mesh arrays, we overlap execution of parts of the CDFG belonging to different iterations, as shown in Figure 2.3. That is, modulo scheduling is commonly employed on mesh arrays to achieve greater speedups and better resource utilization per cycle.

The systems proposed in [PPM09, Ima12] are examples of mesh-based MIMD approaches. In [PPM09], the array contains FUs with dedicated register files which share data amongst themselves. Multiples of 4 FUs can be grouped into a sub-array controlled by a single instruction stream. Sub-arrays communicate amongst themselves at known intervals, and columns of the array can access individual scratch-pad memories, allowing for each sub-array to access multiple data. The multi-processor array of [Ima12] contains small processors each with an instruction memory, accessing shared data memories. In [SLL+00], entire rows or columns of FUs in a mesh are configured by the same configuration word, but each FU operates on a different input datum. In [MLM+05], a mesh array contains four enhanced units which together form a VLIW. Sequential portions of code are executed in the VLIW core, which can feed data directly to the rest of the mesh. This is a case of tight integration of a mesh array.

Execution in Row Based Arrays In row arrays, data produced by the last row can be fed back to the first row as mentioned previously. Execution thus repeats until a termination condition is met, and the array functions as a loop accelerator. In general, loop accelerators tend to be loosely coupled, as they execute several iterations without intervention of the host processor. On the other hand, since row-based arrays tend to be more heterogeneous and imply data directionality, they basically behave as a custom ALU. As such, smaller arrays can be tightly coupled to augment a processor pipeline, and are then controlled by instructions fetched by the processor. Note that

both can accelerate the execution of loops. If the tightly coupled array is fed the same instruction numerous times in a row, it is essentially quickly executing iterations of a loop whose instructions were transformed into a single-instruction configuration for the array.

If the rows buffer their output stages, this creates a multi-cycle array. If the interconnection scheme is sophisticated enough so that outputs of any row can be fed back to any row, this arrangement can easily be used to implement pipelined execution. The amount of required backwards connectivity depends on the IIs of the loops. In contrast, the array can be completely combinational, creating a single-cycle array. If the design is efficient so that no critical path delays are introduced, large speedups can be attained by executing up to tens of instructions per cycle.

In [BPCF13, PFC13], a loosely coupled row-based array contains single operation FUs. A configuration word is given prior to execution, and the array then executes iteratively for a runtime defined number of cycles. Some cases of tightly coupled row-based arrays are also an example of SIMD execution: the array is used a a custom FU in the processor pipeline, as such, a single instruction in the data stream configures the array which fetches multiple operands from the register file [BRGC08, CBC+05].

2.1.6 Accelerator Programmability and Compilation

While the execution model of the array relates to the method of data flow, programmability is related to the manner in which control is provided to the array. This includes control signals of the array, and how this control is generated, i.e., how is application information compiled into array configurations. The type of control that needs to be provided is directly related to the arrangement of FUs and interconnections, type of interface to the GPP and their execution model.

Considering the structure of CGRAs, there are two types of information required to control execution: defining the operations of the FU and establishing interconnections between FUs. The exact format of control information depends on the capabilities of the FUs and the interconnections, as well as the execution model of the design.

Controlling accelerators which are tightly coupled is done in some approaches by embedding custom control instruction in the application binary, i.e., through instruction set extension. The accelerator is thus controlled like any other FUs in the GPP's pipeline [BRGC08]. Custom instructions can results from either offline [CBC+05] or online binary translation [RBM+11, CHM08].

Mesh type arrays require several configuration stages per iteration, each FU performs different operations through an iteration. Datapath type arrays for pipelined execution require very little control. Entire CDFGs are mapped to the array and produce data at given intervals after activation. For instance, in [PFC13], additional code which the processor executes to communicate with the array is automatically generated offline. The array contains single-operation FUs and only the interconnects are configured prior to execution. In [LV09], the binary is modified offline so that the processor initiates communication with the reconfigurable area. The accelerator may also be controlled directly by the GPP by issuing control instructions [SLL+00], or the accelerator itself may actively fetch instructions from local memories [MO98].

Generating control information is unlike compiling a program for a sequential processor. Lack of a straightforward compilation methodology for these heterogeneous architectures greatly hinders their applicability and deployment. Existing binary-acceleration-based architectures still require the use of custom tools, after or during compilation [NMIM12]. Alternatively, on-chip translation methods are employed [LV09, RBM+11], which hide this process from the developer.

Translating CDFGs into these arrays entails finding a free and compatible FU for each operation, such that all operations are mapped without conflict [APTD11, ATPD12]. Pre-processing steps can be performed over the CDFG, such as constant propagation, analysis of WAR and RAW dependencies and elimination of store-load operations over data that is only used locally within the array. In [SGNV05], decompilation techniques are used to recover memory and loop related information from application binaries, providing more efficient binary translation.

2.2 Representative Approaches

The following sections summarize representative state-of-the-art approaches to constructing systems capable of autonomous runtime acceleration. The most relevant approaches are the Warp processor [LV09], the AMBER approach [NMIM12], the CCA [CBC+05] and the DIM array [BRGC08], as these consist of complete binary acceleration systems. In later sections of this section, some additional works are also summarized.

In [Wol03], a general view of hardware/software co-design is presented, a comprehensive summary of CGRA architecture can be found in [Har01], an extensive look at several aspects of reconfigurable architectures is shown in [CH02] and [Cho11] is a survey of CGRA architectures, some of which are also presented here. Approaches presented here stand out as the most recent publications similar to the proposed approach. Table 2.1 summarizes architectural and methodological information for the following approaches which more closely resemble this work.

2.2.1 Warp Processor

The Warp processor shown in Fig. 2.4 is an FPGA-based run-time reconfigurable system based on binary decompilation [LV09]. Cycles are first detected during execution [GRV05]. Once profiled, the running binary is decompiled into high level structures which are mapped into a custom FPGA fabric by custom tools running on an additional processor. The FPGA is modeled with a simpler interconnect structure and resource layout, to facilitate runtime P&R. Once the automatically generated hardware is ready, software execution is migrated for the identified sections by modification of the program binary, and operands are fetched from memory. The system is fine-grained and loosely-coupled. Only small loops in the running program are detected. Targeting more complex loops would greatly increase the effort of on-chip CAD and mapping time. Floating-point operations, pointer operations or dynamic memory allocation are not supported.

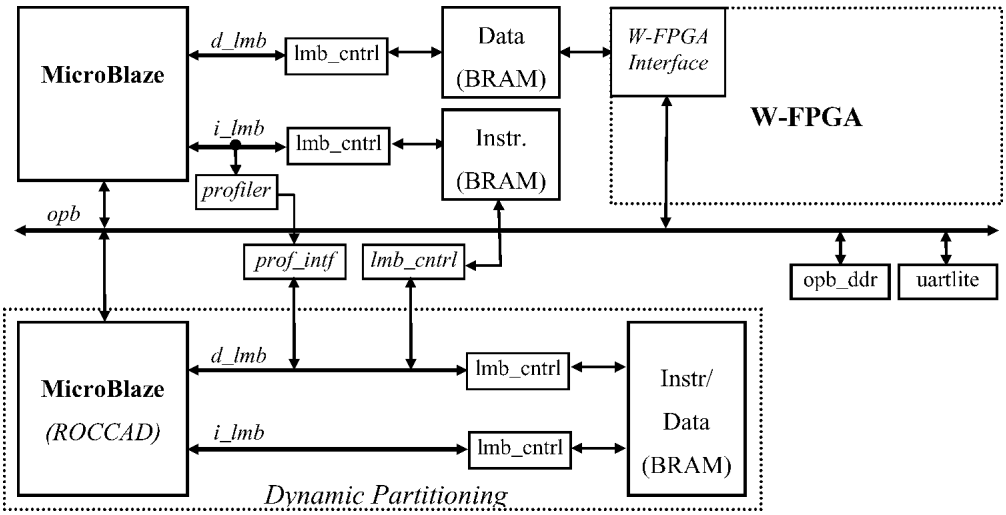


Figure 2.4: The Microblaze based Warp processor system. An additional processor performs runtime synthesis of accelerator hardware through profiling [LV09].

2.2.2 ADEXOR

In [NMM+06, NMM+08], a profiler is used together with a sequencer that stores microcode for the developed accelerator, which is coupled to a MIPS processor pipeline. Its execution is initiated by comparing the current PC with stored information. The accelerator consists of a reconfigurable unit, controlled by configuration bits to perform a given operation. This unit, shown in Fig. 2.5 contains a fixed number of FUs which support integer arithmetic. Up to 1 store operation can be performed. Loads, division and multiplication are not supported. It is configured whenever a basic block, detected by the runtime profiler, executes over a given number of times. Further work produced a heterogeneous reconfigurable unit [MGZ+07]. The flexible interconnection scheme introduced a large multiplexer delay, but many of the configurations were similar. For this reason, the architecture evolved to a more coarse-grained alternative with less configuration overhead.

In [NMIM12] support for conditional execution was implemented. Sequences of forward-jumping hot basic blocks are used to compose single-entry multiple-exit traces. Basic blocks cannot be linked accross function returns, indirect branches or branch and link operations. When constructing a trace, if execution is equally frequent through both the taken and non-taken directions of a branch delimiting a basic, both paths are included into the trace. Only short forward jumps are considered, as long jumps would imply support for executing a large number of instructions on the accelerator. If one of the branch directions is heavily biased, only it is included in the trace. The traces are then transformed into configurations for the accelerator. One configuration requires an average of 615 bits. The total amount of configuration memory required is reduced by performing similarity analysis on the generated configurations. During an offline phase, the application binary is profiled through execution in a SimpleScalar simulator and then modified with custom instruction which load accelerator configurations at runtime. An average speedup of 1.87× was achieved for the MiBench suite, versus a single-issue MIPS processor.

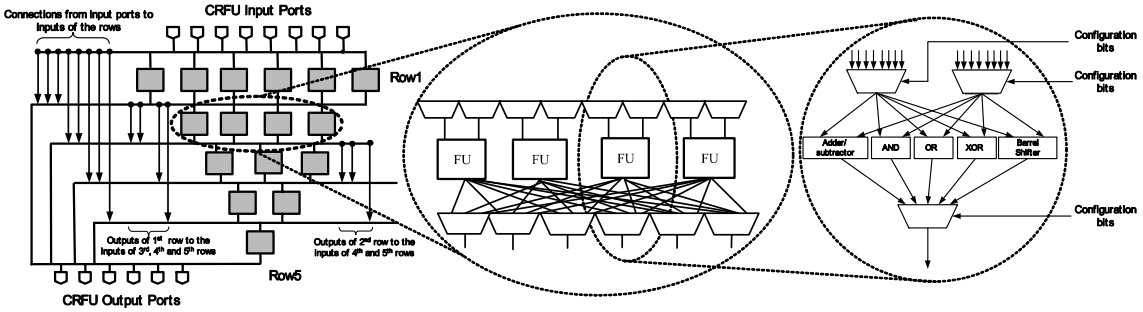


Figure 2.5: An array of FUs which is coupled to the processor pipeline in [NMM+08]. In [NMIM12] it is enhanced with conditional execution.

2.2.3 Configurable Compute Accelerator (CCA)

In [CKP+04, CBC+05] a method for transparent instruction set extension is presented. An ARM pipeline is augmented with a CCA as shown in Fig. 2.6. The CCA is composed of an array containing two types of ALUs. One is capable of arithmetic and logical operations, the other can only execute logical operations. The array is composed of alternating rows of each type of ALU. The number of rows and their width was determined by profiling data extracted by simulation of 29 applications. Candidate traces were identified and frequently executing ones had more weight in deciding the CCA structure. The CCA has 4 inputs and 2 outputs, and several combinations of depth and widths were tested. The width determines the maximum supported critical path, and the row widths the maximum ILP. The CCA does not support memory, barrel-shift, multiplication, division or branch operations. Crossbars connect adjacent rows. Configuration is controlled by setting ALU connections and 4-bit *opcodes*. Despite having rows of ALUs, execution on the CCA is not pipelined, and interrupts are not supported. The presented approach does not require modification of source code or executing binaries. CDFG discovery can be performed online or offline. Online discovery is based on the runtime trace. This method allows for detection of traces crossing control flow boundaries (as performed by the trace cache units of superscalar processors). Offline discovery is performed at compile time, and marks the binary with special instructions delimiting the region of code to translate into CCA instructions. At runtime, the processors instruction stream is altered to use the CCA. This can be done in the trace cache, if one is being used, or in the pipeline's decode stage. Results for 3 variants on these design options are presented. Evaluation was performed using the SimpleScalar simulator with 29 benchmarks. A CCA of depth 4 was shown to be able to execute 82 % of the candidate graphs, and achieved an average speedup of 1.26× for 11 benchmarks versus an ARM 4-issue processor [CKP+04].

In [CHM08], the same authors address binary portability for hardware accelerated systems. Applications developed to utilize a given accelerator might not be compatible with future hardware implementations. The authors propose a virtualization module that monitors the instruction stream, and generates configurations and control for a given accelerator to address this issue. This way, standard binary can transparently utilize any hardware accelerator. To validate this, a loop accelerator architecture which executes modulo-scheduled loops is employed. The accelerator in-

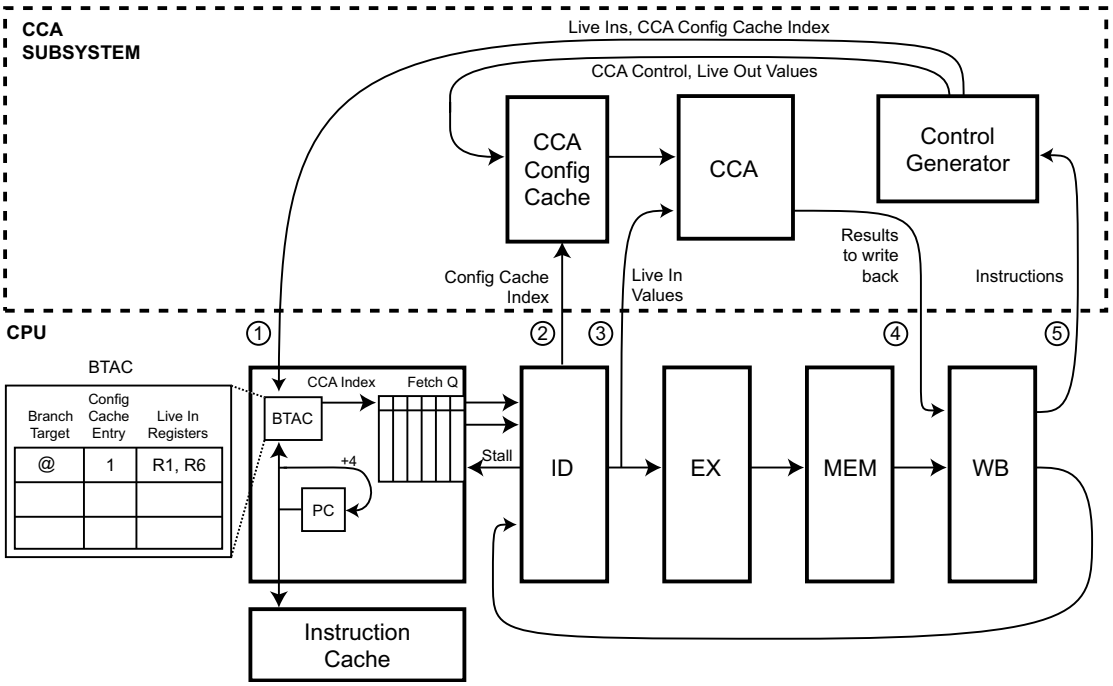


Figure 2.6: The CCA array is coupled into the processor pipeline. Execution is shifted towards the CCA after generating configurations for offline delimited regions of code [CBC+05].

tegrates the CCA as a functional unit. It was designed based on profiling data, by determining the required resources to achieve the best possible speedup for the candidate loops. The accelerator supports integer, single- and double-precision floating-point operations as well as 16 address generators for load operations and is composed of reconfigurable heterogeneous FUs. Configurations are generated by binary translation of the loops to accelerate. A thorough explanation of the steps of the translation mechanism is given. Scenarios in which one or more steps are either performed online (by the virtual machine) or offline (through static compilation steps) are analyzed. A mean speedup of 2.66× is achieved for 38 benchmarks versus a single-issue ARM, using the chosen hybrid combination of online/offline binary translation without compromising binary portability.

2.2.4 Dynamic Instruction Merging (DIM)

The DIM Reconfigurable System [BRGC08] proposes a reconfigurable array of FUs with a row-based topology supported by a dynamic binary translation mechanism. The array is tightly coupled to the processor, with direct access to the processor's register file as shown in Fig. 2.7. The DIM array is composed of equal rows of heterogeneous FUs, some of which are ALUs supporting arithmetic and logic operations. There can be as many concurrent memory operations as the number of available memory ports. Floating-point and division operations are not supported. Sequences of instructions are transparently mapped from a MIPS processor to the array, which executes the custom instructions in a single-cycle. A speculation mechanism enables the mapping of units composed of up to 3 basic blocks. The instruction stream is monitored concurrently with

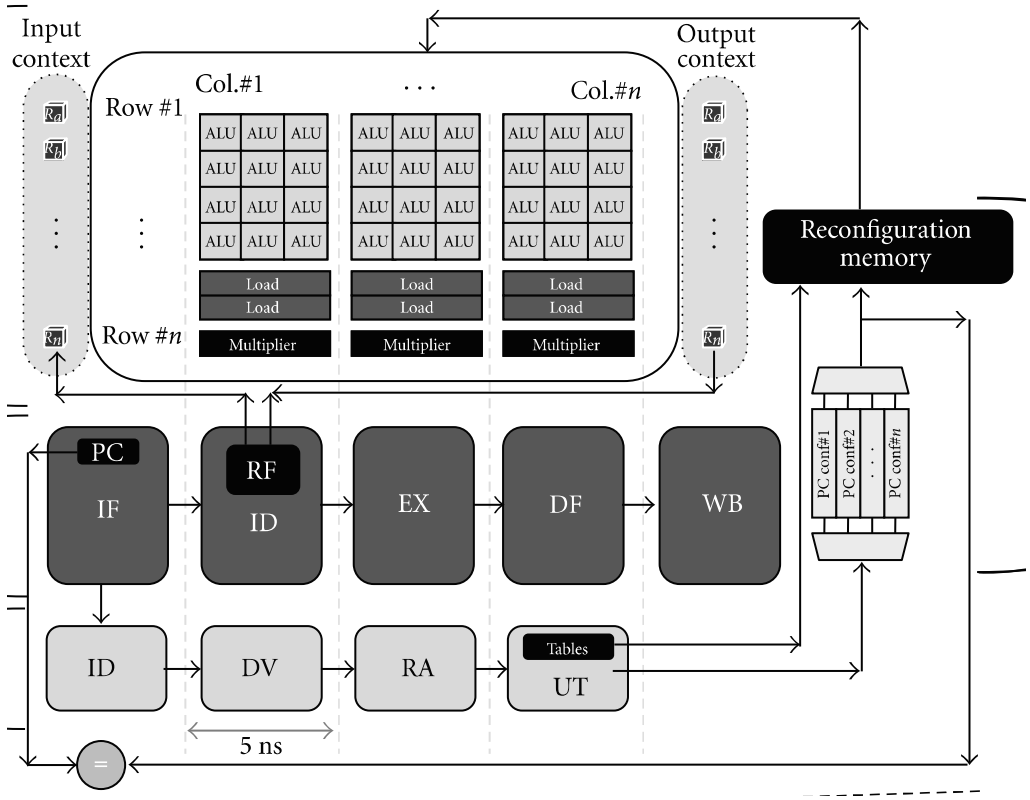


Figure 2.7: Like the CCA, the DIM array is placed in the processor pipeline. A separate pipeline performs binary translation during execution (adapted from [RBM+11]).

execution for candidate basic blocks to translated into array configurations. Configurations are stored for later use. An average speedup of 2.5× is achieved for 18 benchmarks of the MiBench suite. In [RBM+11], DIM arrays were coupled to SparcV8 processors to exploit thread parallelism in tandem with ILP. Using multiple DIM enhanced processors, a mixed approach of acceleration through ILP and thread parallelism was evaluated through simulation.

2.2.5 ASTRO

The ASTRO approach is presented in [LCDW15]. It is based on detection of MicroBlaze instruction sequences by execution profiling on a simulator, followed by synthesis of one loop accelerator per candidate instruction sequence. The accelerators are one-to-one translations of CDFGs into pipelined data-paths, which execute pipelined loop iterations, and are coupled to the MicroBlaze via a peripheral bus, and migration of execution is performed by monitoring the MicroBlaze's instruction address at runtime. ASTRO focuses on maximizing memory access parallelism. Dynamic memory access analysis is performed to determine disjoint regions of access. Memory accesses within the hot regions are grouped into partitions based on access dependencies. Each partition is assigned a customized cache. The analysis also deals with data hazards. With this information, a tailored Block RAM (BRAM) based multi-cache system is created per-accelerator, allowing for efficient exploration of data parallelism. The cache system for the accelerators may

require, at least, a partial cache invalidation after accelerator execution, leading to overheads. An average ASTRO accelerator with a multi-cache network requires 20400 Lookup Tables (LUTs) and 5900 Flip Flops (FFs) on a Virtex-5 device. For 10 benchmarks from MiBench and SPEC2006, the geometric mean speedup achieved was 7.06× versus software-only execution. The authors also conclude that maximizing concurrent accesses leads to approximately a speedup of 2× over single-access accelerators.

2.2.6 Work of Ferreira *et al.*

In [FDP+14] an approach is presented for acceleration of inner loops detected by runtime binary profiling. Auxiliary hardware monitors the execution stream for frequent backwards branches. Detected loops are modulo scheduled by on-chip translation tools onto the target accelerator. Either RISC or VLIW binaries can be translated into modulo schedules for the CGRA, with support for RAW/WAR dependency detection. The accelerated traces contain 40 up to 120 instructions. The target accelerator is a CGRA architecture with a programmable interconnect, containing 16 FUs (ALUs, multipliers, and up to two memory units). Floating point operations are not supported but there is support for conditional value assignments. A crossbar connects the FUs and each FU has its own register file. The accelerator is tightly coupled to the main processor and fetches operands from its register file. The proposed approach is compared to several VLIW models. For evaluation, the approach and accelerator targeted a Virtex-6 device. A speedup of 2.0× over a 8-issue VLIW is achieved for 5 benchmarks. The proposed accelerator requires 13000 LUTs and 23 BRAMs.

2.2.7 Morphosys

MorphoSys [SLL+00] consists of an array of Reconfigurable Cells (RCs), loosely coupled to a custom RISC processor, a memory to hold array configurations, a shared memory to exchange data with the array and external memories with program code. The array can access the external memory via DMA, and has direct access to the configuration and shared memories. The layout is a 8x8 mesh, and each row or column is configured by the same 32-bit word. The array is divided intro 4x4 quadrants which can exchange data. Data can be routed to all RCs within the same quadrant. The RCs are homogeneous coarse-grained ALUs augmented with a multiplier, a shift unit and 4 registers, and they operate on 8 or 16 bit values.

The processor controls array execution and configuration by issuing special control instructions added to its instruction set. Use of a custom compiler and modification to source code are necessary. These instructions can command the array to execute a row/column or move data between the processor, the array and shared memories. A DMA controller is also used by the processor to load new configurations onto the array. Since each row/column performs the same instruction for different input data, this implementation is an example of the SIMD paradigm. Computations along a row/column may continue while another row/column is being reconfigured.

Table 2.1: Characteristics of related approaches

Approach	Base Processor	FU Arrangement	Interconnection Scheme	Supported Operations	Memory Access	Accelerated Trace	Methodology	Speedup
Warp [LV09]	MicroB-laze	Fine grained custom reconfigurable fabric with bit-level logic	FPGA-like switch-boxes connect logic blocks	Fixed-point arithmetic and logic	One port for regular patterns	Most frequent innermost loops	Runtime trace profiling, binary disassembly and circuit synthesis	3.20× (geometric)
DIM [BRGC08]	MIPS	Matrix of homogeneous rows of heterogeneous FUs	FU outputs drive inputs of FUs in any row after their own	Logic and arithmetic (no division or floating point)	Concurrent accesses to random addresses	Sequences of basic blocks	Runtime binary profiling and generation of accelerator configurations	2.17× (arithmetic)
ADEXOR [NMIM12]	MIPS	Rows of heterogeneous FUs in an inverted pyramid shape	Crossbars between neighbour rows Additional lines connect distant FUs	Logic and fixed point arithmetic (no division or multiply)	One store operation	Single-entry multiple-exit trace with multi-path	Offline profiling, binary modification and configuration generation	1.87× (arithmetic)
CCA [CBC$^+$05]	ARM	Two types of FUs compose homogeneous alternating rows	Crossbar connections for neighbour rows	Logic and integer arithmetic excluding multiply and divide	Not supported	Sequences forming graphs with 4 inputs and 2 outputs max.	Compile time subgraph detection; Runtime configuration generation	2.21× (arithmetic)
ASTRO [LCDW15]	MicroB-laze	Rows of single-function units	Dedicated inter-row connections	Integer arithmetic and logic	Tailored cache for multiple parallel accesses	Atomic sequences of basic blocks (w/ hazard analysis)	Offline trace detection via simulation; offline synthesis accelerators	7.06× (geometric)
Ferreira *et al.* [FDP$^+$14]	RISC/ VLIW	16 FUs w/local register files	Global crossbar	Integer arithmetic and logic; conditional assignments	2 Memory units	Loop traces	Modulo-scheduling of binary traces	2.0× (arithmetic)

2.2.8 Additional Related Works

This section briefly presents approaches regarding other aspects of binary acceleration and accelerator architectures, such as memory access optimizations, and especially loop scheduling.

Kim et al. [KLSP11] address the issue of handling memory accesses in mesh type accelerator architectures. They present a scheduling algorithm and framework to map operations, temporally and spatially, to a generic type of mesh, such that memory access conflicts are reduced and IIs are minimized. A CDFG optimization algorithm is also used to reduce redundant memory accesses. A technique for distributing data through several single-ported local memories allows for parallel access to elements of the same data array without conflict. To determine the optimal distribution a static code analysis step is required, which considers factors such as access rates and array sizes.

Inner loops are detected and mapped onto a heterogeneous CGRA at runtime via binary translation in [FDP+14]. Auxiliary hardware monitors the execution stream and the translation of loops is performed by on-chip software. Either RISC or VLIW binaries can be translated into modulo schedules for the CGRA, with support for RAW/WAR dependency detection. The CGRA has 16 FUs of three different types: *load/store* units, ALUs or multipliers. A crossbar connects the FUs and each FU has its own register file. The CGRA is tightly coupled to the main processor and fetches operands from its register file. For the five reported benchmarks, the mean speedup is 2× versus a 4-issue VLIW. The proposed CGRA requires 13,000 LUTs and 23 BRAMs.

In [OEPM09], a heuristic is applied to improve solutions provided by edge-centric modulo scheduling [PFM+08], in which speculative scheduling is required when CDFG nodes form a closed circuit. To address this, closed circuits are clustered into a single node, eliminating backwards edges from the CDFG and therefore speculative scheduling. The approach is evaluated with 390 CDFGs, extracted from 12 applications, onto a CGRA with 16 FUs in a 4x4 arrangement. Scheduling is up to 170× faster, relative to state-of-the-art approaches.

A loop accelerator whose runtime programmability is the result of a modulo-scheduling based tool chain is presented in [FPKM08]. The accelerator template is a VLIW type architecture with schedule-time specified FUs and connections. A baseline loop is first used to create the initial architecture: a direct realization of that loop with no programmability. Architecture generalizations are analyzed to increase the runtime flexibility of the accelerator. Additional loops are then scheduled using a constraint driven modulo scheduler. The issue is modelled as a Satisfiability Modulo Theory problem and given to a solver. The approach is evaluated by creating base accelerators and then scheduling randomized variations of the base loop or loops from other applications.

A method to support nested-loop pipelining is presented in [KLMP12]. Loops with up to one inner loop are modulo-scheduled onto a 2D CGRA. In [CM14] the problem of modulo-scheduling CDFGs onto 2D CGRAs is modeled as a graph minor problem, where a modulo routing resource graph is used to minimize routing costs by route sharing. In [ATPD12], a slack-aware modulo scheduler targets a highly heterogeneous CGRA. Kernels are partitioned according to CGRA limitations in order to make it possible to modulo-schedule them. A valid schedule is found via simulated annealing by starting from an optimal yet possibly invalid solution.

2.3 Dynamic Partial Reconfiguration in FPGAs

The previous section introduced the notion of reconfigurable systems in the sense of reconfigurable processing units being used as accelerators. In this scenario, reconfiguration generally occurs at a coarse scale, and the target technology for implementation varies between ASICs and FPGAs.

In contrast to the designer-specified reconfiguration capabilities of some coarse-grain accelerators, typically implemented as controllable interconnects or configuration words, some FPGA devices support Dynamic Partial Reconfiguration (DPR) [Xil12b]. The logic circuitry in an FPGA depends on the contents of the underlying configuration memory which establishes fine-grained component connections and contents of logic cells. In traditional flows, an entire bitstream file is generated to write to the configuration memory. Instead, DPR allows for only specific regions of the configuration memory to be modified during runtime. The logic circuitry of the equivalent area can be altered without disturbing operation of the remaining circuit. This feature of FPGA devices allows for the development of systems which rely on reconfiguration in a number of ways.

This feature has existed for some time (as early as Virtex-II Pro devices), but has seen slow adoption, in part due to limited tool-support from vendors. This work focuses specifically on Xilinx devices, and on the flow they provide to dynamically reconfigure the logic circuitry. They contain an Internal Configuration Access Port (ICAP) which can be accessed via software to write *partial bitstream* files to the FPGA's configuration memory. Partial bitstream files can represent minute modifications to the implemented logic, or an entire hardware module which occupies a predefined region on the device [Xil12a].

The later type of approach defines one or more reconfigurable regions, where each region serves as a host for a set of designer specified modules. Creating this kind of design is equivalent to creating several static designs, which share a portion of their logic. A system may use several reconfigurable regions to instantiate different hardware tasks according to system demands. Each module is stored in a partial bitstream file, which is written to the device's configuration memory at runtime. Essentially, the same area of the device is used to implement several different functions by time-multiplexing hardware resources.

2.3.1 Examples of Partial Reconfiguration Applications

Despite the underutilization of DPR, several works, focused on markedly different application domains, have demonstrated the design potential of dynamically controlling circuit logic.

In [LPV10], a partial bitstream based FIR filter system is presented. Two reconfiguration variants are presented. One alters only filter coefficients, another the entire filter datapath. An output of 10 Mega-samples per second is achieved whilst reconfiguring the filter 70$\times$ per second. In [ESSA00], dynamic reconfiguration is used to create a self-healing fault tolerant system. In [GAA$^+$07], faults are purposely introduced for self testing by modifying the circuit under test via runtime reconfiguration. In [SRK11], partial bitstreams are used to implement some of the protocol layers of IPSec in a lightweight fashion. Three cryptographic functions are switched in and

out of a slot based dynamic area (as per vendor flows), allowing for an implementation which allows for significant area savings with performance comparable to full hardware implementations. The authors state that an equivalent non-DPR design on the same target device would not meet timing requirements. An implementation of a DVB-T2 coder/decoder is presented in [FISS12]. An FPGA with a reconfigurable area is used to host one of two developed modules in a time-multiplexed fashion. One module performs demodulation of the received signal and the second forward error correction. The FPGA communicates with an external central processor through USB. In [LP13], a system which performs single pixel operations is presented. Partial reconfiguration is used to instantiate more pixel processor cores, change the pixel operations performed to alter other parameters of the cores to meet power and performance goals.

Finally, in [KTB+12] a summary regarding DPR based architectures, application specific requirements and application examples is given. Tools geared towards DPR are also presented. In [PDH11] the factors introduced by DPR that influence performance are surveyed and experiments are conducted with a DPR based architecture to validate a cost model of DPR.

2.3.2 Design Considerations for Partial Reconfiguration Based Systems

The examples of applications briefly presented showcase the applicability of DPR based design. Also, several academic tools [SAFW11, BKT12, SF12] have been developed to explore DPR in ways not supported by vendor tools, which demonstrates the increasing interest in research regarding this technological feature. However, some issues are introduced when relying on DPR.

Firstly, storage space is required for partial bitstreams. Some works have addressed this by proposing more efficient bitstream compression/decompression methods [GC08, QMM11]. Secondly, reconfiguration times via ICAP are in the order of the millisecond, when relying on current vendor flows. This is the most important aspect when considering the use of DPR to support the reconfiguration capabilities of an accelerator. To reduce this overhead, some approaches have developed more efficient ICAP controllers. In [CZS+08], a controller with DMA access achieves speedups from 20× to 50×, depending on the device, relative to Xilinx's own controller. In [VF12] the proposed hardware solution to drive the ICAP decreases the reconfiguration time by an order of magnitude relative to existing vendor solutions. The work in [EBIH12] replaces the vendor ICAP controller itself with an alternative which the authors claim to be 25× faster, while also being capable of recovering from faults, and requiring half the device area to implement. In [HGNB10], the throughput of Xilinx's own ICAP controller is roughly doubled by modifying it so it interfaces it directly to the MicroBlaze processor, via a point-to-point connection. Finally, some approaches address this as a task scheduling problem, proposing configuration pre-fetching and/or scheduling, which sometimes considers pre-emptive configuration switching [RSS08, LEP12].

2.4 Concluding Remarks

This chapter presented a brief overview of existing approaches which focus on augmenting a main processor with an accelerator module. The proposed accelerators either function as processor

pipeline units providing special instructions or as larger units, apart from the processor, serving as loop accelerators. An accelerator design entails several aspects: the operations it supports, the granularity of its computational units, their number, layout and interconnections, how the execution is controlled, and what exactly is accelerated.

Most approaches presented rely on binary level-information and quantitatively designed accelerators, which are targeted by a custom tool chain. Consequently, the accelerator contain coarse-grain units (i.e., word-level operators), essentially behaving as an array of limited ALUs. Reconfiguration entails controlling these units and the flow of data between them via the existing interconnect structure. This connectivity is what varies more noticeably per approach, and is a significant aspect in determining what the accelerator is capable of supporting.

For the existing implementations, the accelerator hardware is generated prior to determining what will be accelerated. Adding reconfiguration capabilities in the form of multi-operation units and rich interconnections is an attempt at increasing the successful mapping of target loops on to the accelerator. However, this is not always successful, and the instantiated hardware may even be excessive for the loops to accelerate, meaning the benefits of circuit specialization are lost.

This work addresses this problem by generating the accelerator hardware after the detection of the application hot spots, thereby ensuring circuit customization, instantiating only the minimum required hardware to accelerate the target loops. Additionally, the capability for DPR is unique to FPGA devices and is promisingly matched with the notion of runtime reconfigurable heterogeneous systems. That is, systems capable of modifying the available hardware based on the required workload and one or more target metrics. So, DPR is exploited to further ensure circuit customization, by simplifying the interconnection logic that the FPGA circuitry implements at any one time, and also to provide higher resource savings. Finally, to make the approach easier to adopt by developers, both the generation of accelerator hardware, its integration to the system, and the migration of execution from software to hardware occur transparently, without need for software modification or manual hardware design effort.

Chapter 3

Overview of Implementations and General Tool Flow

As per the general approach explained in Chapter 1, the objective of this work is to generate and integrate specific customized instances of accelerators into a host system as co-processor units. Towards that end, several accelerator architectures and tools, as well as supporting hardware and system level architectures, were devised. A general system level and tool flow view are shown in this chapter, to provide a detailed overview of the approach and to establish the context for later content. Each accelerator implementation an evaluation is presented in succeeding chapters. The general approach relies on an automated step to identify frequently executing instruction traces, representing the portions of computation the accelerators will execute. This work relies on a specify type of loop trace called Megablock.

The following sections show a high-level view of the system level architectures used to evaluate the accelerator implementations. Also shown in Section 3.3 are the tools required to detect frequent Megablocks, transform them into accelerator instances, create the necessary communication infrastructure between host processor and accelerator, and configure the runtime migration mechanism. Section 3.5 concludes this chapter with a brief summary of the specific accelerator architectures and experimental results presented in later chapters.

3.1 System Level Architecture

Figure 3.1 shows a generic view of the type of system level architectures into which the several accelerator designs were integrated. These systems contain: the host processor, which for all cases is a MicroBlaze soft-core processor; either a local or external memory to hold code and/or data; the accelerator instance, connected to the host processor either via a Processor Local Bus (PLB) connection (Chapter 4) or a point-to-point Fast Simplex Link (FSL) connection (Chapters 5 to 7); and an auxiliary module called *injector* (of which there are several variants), whose purpose is to interface with the MicroBlaze' instruction bus and migrate execution to the accelerator by modifying the contents of the instruction bus.

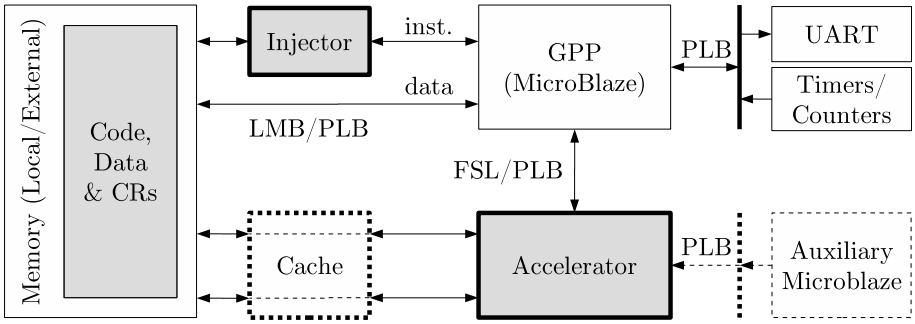


Figure 3.1: General overview of developed system architecture

The code memory is local, and holds automatically generated Communication Routines (CRs) which implement the communication between the processor and the accelerator. If the data memory is local, the MicroBlaze uses the Local Memory Bus (LMB) interface to access it. Two LMB multiplexer modules (shown in Chapter 5) are used so the BRAM's two ports are shared between it and the accelerator (not shown in Fig. 3.1). If the data memory is external, the MicroBlaze uses a memory controller to which the accelerator also has access, via a dual-port cache (Appendix A).

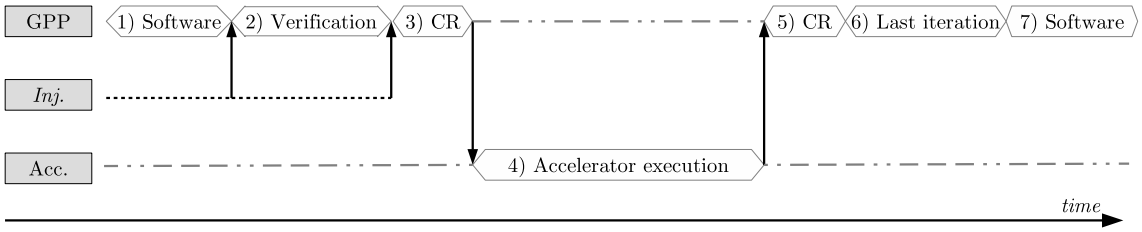
The systems also contain custom counter/timer modules to retrieve execution statistics and UART modules. The accelerator and *injector* modules control when the several timers and counters start/stop. An early implementation (presented in Chapter 4) also required an auxiliary MicroBlaze for reconfiguration tasks, which were later implemented via the *injector* or CRs.

As the next section explains, the system architecture variants used follow the same execution model. Architectural differences between them stem from the different capabilities of the accelerators under test (i.e., support for memory access and interfaces), or due to the need to test the accelerators under different conditions (i.e., use of external data memory to support larger applications). The experimental sections of this thesis describe which system architecture was used to validate the several accelerator design iterations.

3.2 General Execution Model

As was explained before, the present approach achieves transparent binary acceleration by relying on four general steps. The detection of Megablocks and their translation is performed offline, via custom tools. The transparent migration and acceleration take place at runtime. Figure 3.2a summarizes the migration-enabled execution flow.

The MicroBlaze executes the unmodified binary, and the *injector* monitors its instruction address. It intervenes when the address on the bus matches any one of the addresses it holds in an internal memory. These are the start addresses of the translated Megablocks. The *injector* begins the transparent migration step by replacing the fetched instruction with one or more instructions which result in an unconditional branch to a known memory position. Each start address corresponds to one such memory location. The right-hand side of Fig. 3.2b shows how the MicroBlaze behaves due to the migration mechanism.



(a) Migration of execution to accelerator via *injector* intervention

Original program:

<reverse>:			cycles
1b4:	addk	r4, r6, r0	
1b8:	andi	r3, r5, 1	**1**
1bc:	or	r3, r3, r6	**1**
1c0:	addik	r4, r4, 1	**1**
1c4:	addk	r6, r3, r3	**1**
1c8:	xori	r18, r4, 32	**1**
1cc:	bneid	r18, -20	**2**
1d0:	sra	r5, r5	**1**
1d4:	rtsd	r15, 8	

Total cycles per iteration: 8
Total cycles: 8 * 32 - 1 = 255

Megablock execution in Accelerator + CR:

<reverse>:			cycles	stage
1b4:	addk	r4, r6, r0		1)
1b8: **Replaced by:**				
Injector delay :			= 8	2)
+ 3 CR cycles			= 11	3)
+ 3 cycles p/iteration:				
11 + 3 * 32			= 107	4)
+ 15 CR cycles			= 122	5)
+ last iteration in software				
122 + (8 - 1)			= 129	6)
1d4:	rtsd	15, 8		7)

delta = 255 - 129 = **126** → **Speedup**

(b) Software only execution of loop kernel (left-hand side) and accelerator enabled execution (right-hand side)

Figure 3.2: Temporal diagram of migration and instruction level behaviour due to migration

The target location of this branch contains a CR which the MicroBlaze executes to communicate with the accelerator. Depending on the type of processor/accelerator interface, the exact contents of the CR vary. By executing the CR the MicroBlaze sends specific input operands to the accelerator from its register file. Depending on the system variant, the CR may also include configuration values to write to the accelerator. For other variants this process is performed by other system modules (the auxiliary MicroBlaze or the *injector* itself). The generation of CRs and their integration into the application code is explained in Section 3.3.3.

After receiving all operands, the accelerator executes the translated loop, exploiting ILP and loop pipelining. All instructions of a Megablock are implemented on the accelerator, including one or more *branch* instructions, which terminate the execution of that particular loop path. These conditions are evaluated using live input data, meaning the accelerator is capable of performing an arbitrary number of iterations per call. However, it also means that the last iteration must be executed through software. This allows for the software execution to follow the control flow which triggered the end of the accelerator call.

In earlier implementations, the MicroBlaze polled accelerator status registers to determine when to retrieve results. For later implementations, the MicroBlaze idles while the accelerator is executing. Either the CR itself or a hardware feature implements a blocking wait. In either case, the processor fetches results back into its register file by executing the remainder of the CR. The last step of the routine is a branch back to the instruction address where the *injector* intervened. Execution continues normally and the accelerator may be called again. If the *injector* is disabled, execution can proceed fully through software, as the binary is not modified by the offline tools.

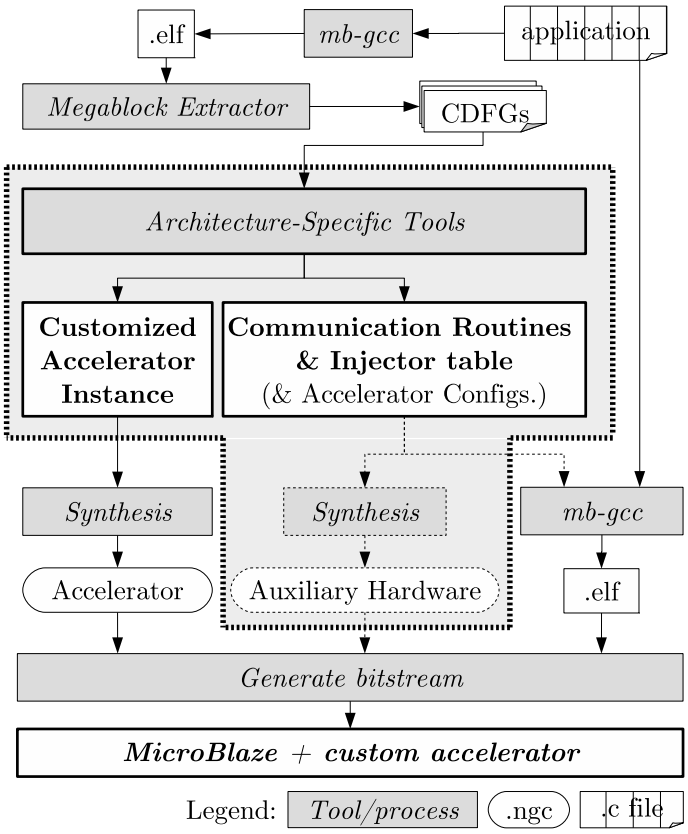


Figure 3.3: Generic tool flow for the several accelerator and system level architectures. Grey area denotes slight implementation dependent variations.

Utilizing the *injector* module to monitor and modify the instruction bus fulfils one of the objectives of the work: transparently migrating execution from processor to accelerator. No binary or processor modifications are necessary to integrate the accelerator. Designing the *injector* required taking into account the characteristics of the instruction bus, both in terms of interface and sequences of exchanged control and data. This in turn depends on how the MicroBlaze issues requests onto its instruction bus based on its internal pipeline. Section 3.4 explains the *injector* and provides a migration example. The following section explains both the process of configuring the *injector* and generating an accelerator instance tailored for a target application.

3.3 General Tool Flow

The generation of custom accelerators is supported by an offline tool chain. Throughout the several accelerator and system implementations, the supporting tool chain suffered some modifications but can be summarized by the diagram in Fig. 3.3.

The non-shaded area represents third-party or vendor tools. These segments of the implementation flow are independent of accelerator or system level architectures. The shaded area represents the bulk of the developed custom tools. Based on the specific accelerator architecture, the HDL

generation tool(s) differ. The generation of CRs is considerably more independent of that aspect, being only a function of the used interfaces and other minor features.

3.3.1 Megablock Extraction

The flow begins with the application code. The frequent loop traces of the target application are produced by generating a MicroBlaze executable and processing it through the Megablock extractor [Bis15]. The extraction process outputs any loop path which obeys the detection criteria that can be provided to the tool. These include: the maximum size of the loop pattern (i.e., number of instructions in one loop trace iteration); the minimum number of executed instructions (i.e., size of pattern multiplied by number of iterations); unrolling of inner loops (which produces large unrolled traces); and several minor optimizations such as constant propagation.

The loops detected by the extractor contain a single entry point, which is the first instruction of the trace, and multiple exit points. These are the control instructions (i.e., data-dependent *branches*) which cause execution to break away from the pattern and follow a different path (e.g., an *if-else* within a loop produces two possible loop paths).

Unconditional branches are removed by optimization, including subroutine return instructions. This means the detected loop traces may cross function call boundaries. For example, a function call could be encompassed within a *for* loop, and the detected trace could include the function call (depending on the contents of the function itself). This is a striking contrast to, for instance, HLS approaches which target the function body or even other binary approaches which target basic blocks. In terms of applicability, it means the source code does not need to be made more tool-friendly, as is required by some HLS approaches (e.g., by using *pragmas*).

The performance achieved by the accelerator are due to exploiting these detection features. But performance is also a function of the detection limitations. For this particular tool, they include: no indirect memory access analysis (i.e., no hazard detection/resolution for data dependencies through memory); loop unrolling is always performed up to the outermost loop (e.g., cannot unroll only up to a specified nesting level); since the loop trace extraction is performed by simulation, it cannot be assured that coverage is fully representative (i.e., functions may remain uncalled, or loop iteration counts may depend on data); the extractor is limited to the MicroBlaze instruction set; and loop traces are detected for regions of code which are not candidates for acceleration.

Regarding this last point, examples would be the loop traces that occur within the *printf* function or any built-in routines for data-type casting. The former does not constitute any relevant data processing, since execution time only due to the lengthy polling times of accessing a *stdout* device. As for built-in routines (e.g., type casting, software emulation of division or floating-point, etc) the respective loops are small, despite a large iteration count, and most of the instructions are branches. That is, there are many of possible paths, where none is dominant or data-oriented.

For the work presented here, no limitation was imposed on the trace size. Supporting large traces, however, does require an efficient accelerator design as the following chapters will show. Loop unrolling was performed on a per-case basis, avoiding cases where the number of inner loop iterations resulted in a cumbersomely large unrolled trace. Constant propagation was also enabled.

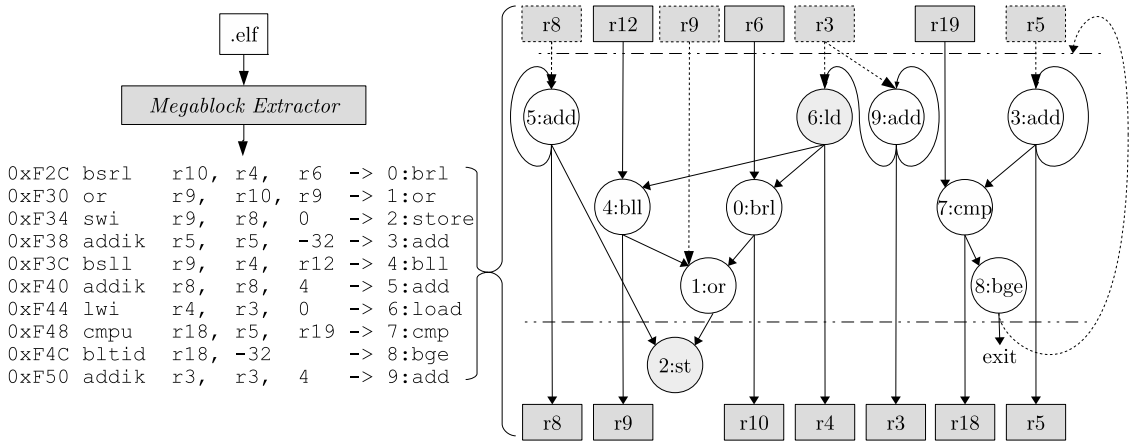


Figure 3.4: Example of extracted loop trace and resulting CDFG

The extractor produces a CDFG for any Megablock trace which repeats more than twice. This results in numerous traces per application, many of which are not very representative in terms of execution time. An automated selection step could filter out Megablocks which correspond to small portions of the execution. However, since uninteresting portions (e.g., *printf*) which are also extracted may represent large amounts of execution time, there is no way to disambiguate these cases. Thus, selection of CDFGs to process further is currently done manually by inspection of the *ELF* file and the Megablocks. The criteria used for selection include: the size of the trace, the number of iterations performed and, based on the capabilities of the accelerator and system, the type of instructions in the trace (e.g., floating-point or memory access instructions).

An additional criterion is the number of exit points of the trace. Traces with fewer exit points are generally better candidates for acceleration, since this makes it more likely that more iterations will be performed uninterruptedly. For instance, consider a *C* level loop construct containing a number of *if-else* clauses. This will result in multiple loop paths being extracted, each with a multitude of exit-points. As additional loop paths need to be followed, it may become more unlikely that any single loop path will iterate a large number of times.

This is related to a final limitation of this migration approach: if several loop paths start at the same address, only one can be accelerated. It is impossible to disambiguate which loop path should be accelerated relying only on the start address. In these cases, the most frequent path is chosen: when a start address is detected, the accelerator is called to execute that (presumed) loop path; once an exit is triggered, execution returns to software and one iteration is performed through another path; the accelerator is then called again. This is tolerable when one loop path is much more frequent than the remaining path(s). However, all loop paths may be equally frequent. The issue is exacerbated when two frequent paths execute alternately: every time the accelerator is called it performs no useful work, as at least more than 1 iteration must be executed.

These issues stem from a Megablock detection limitation: loop paths are not combined into multi-path traces. It would not be inconceivable to transform such multiple path representations into parallel conditional execution. As is, the present work targets large, frequent single path traces

which, as later work will show, may contain any instruction of the MicroBlaze instruction set.

Regarding specific file outputs, the Megablock extractor produces text representations of the CDFGs constructed from the Megablock traces. The information in these files includes which registers are inputs and/or outputs to the trace, what CDFG node feeds them, connections between nodes and topological information. The CDFGs represent a single trace iteration, and are therefore acyclical. Inter-iteration data dependencies (i.e., backwards edges between CDFG nodes) are determined by the translation tools.

3.3.2 Generation of the accelerator HDL Description

The accelerator architectures employed in the present approach are essentially heavily parametrizable templates. They are co-processors which allow for the specification of the number and type of computational resources (i.e., FUs), and the interconnections between them, among other minor aspects. The purpose of the developed tools is to generate a small number of Verilog include files containing parameters which are read during synthesis. This creates a specific instance which is capable of executing the desired CDFGs.

The HDL generation tools underwent continuous development according to the accelerator architectures. As such, the tool features and generated outputs varied in a manner difficult to summarize into a generic representation. The functionality of these tools can be summarized as: allocation of FUs, creation of interconnect structure, generation of configuration information for runtime use. As the following chapters show, the designed accelerator architectures include: a multi-row FU array with data directionality; a second implementation of this design with memory access capability; and a single-row accelerator, with units controlled per-cycle in order to execute pipelined loop iterations. Each architecture is supported by its own version of the translation tools.

The translation/scheduling process accepts multiple CDFGs. Each graph is processed individually. The generated accelerator is stored in a data structure and re-utilized while translating/scheduling the next CDFG. The final result is an accelerator instance able to execute multiple CDFGs. For each graph, a number of pre-processing steps are first performed. These include: determining backward edges, establishing control dependencies, establishing the topological dependencies between nodes, and their slack. Control dependencies ensure that all *branch* operations in an iteration must execute before: 1) a new iteration is initiated, and 2) any *store* operation belonging to the same iteration is performed (to avoid writing erroneous data to memory).

After pre-processing a CDFG, its nodes are translated into FU placements and interconnections. Each CDFG node is processed individually. The translation attempts to find an existing FU in the current accelerator structure (if any), to re-utilize for the execution of that node. For the multi-row architectures, this entails searching through the accelerator's rows of FUs within the node's permitted slack. For the single-row architecture, the process involves a modulo scheduling step explained in Chapter 6. If an available FU cannot be found, a new one is instantiated. According to the node's required inputs, the connectivity is updated. This involves specifying a multiplexer per FU input. For the multi-row designs, the edges between nodes are transformed into wiring between FUs which remains static throughout the execution of the respective CDFG.

Listing 3.1: Excerpt from HDL generation tool output for the accelerator architecture presented in Chapter 5

```
// Verilog accelerator parameters:
parameter NUM_CFGS   = 32'd1;
parameter NUM_IREGS  = 32'd4;
parameter NUM_OREGS  = 32'd5;
parameter NUM_COLS   = 32'd5;
parameter NUM_ROWS   = 32'd4;
parameter NUM_STS    = 32'd0;
parameter NUM_LDS    = 32'd2;

// FUS
parameter TOTAL_FUS   = 32'd17;
parameter TOTAL_EXITS = 32'd1;

parameter [0 : (32*NUM_ROWS*NUM_COLS) - 1]
FU_ARRAY = {
    // top row
    `A_ADD, `A_ADD, `A_ADD, `NULL, `NULL,
    `M_LD,  `M_LD,  `L_XOR, `PASS, `NULL,
    `A_MUL, `B_EQU, `PASS,  `PASS, `PASS,
    `A_ADD, `PASS,  `PASS,  `PASS, `PASS
    // bottom row
};

parameter [0 : (32 * NUM_CFGS) - 1]
    NR_INPUTS = {
        32'd4
};

parameter [0 : (32 * NUM_CFGS) - 1]
    NR_OUTPUTS = {
        32'd5
};
```

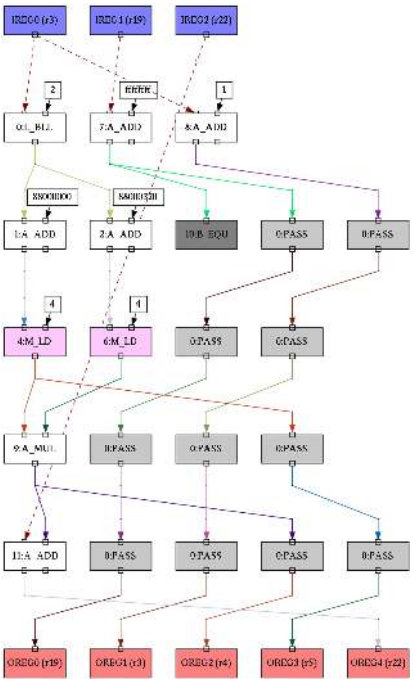


Figure 3.5: Example instantiation of Functional Unit array for the parameters in Listing 3.1

For the single row design, establishing the connections between FU requires per-cycle temporal awareness. In either case, the FU allocation process tracks the existing connectivity, attempting to place nodes onto existing FUs such that the final connectivity is reduced.

Listing 3.1 shows an excerpt of the translation tool's Verilog output for the multi-row accelerator architecture presented in Chapter 5. This file specifies some top level aspects of the accelerator instance as well as the array of FUs. The connectivity specification is to verbose too present here. Instead, a graphical representation of the resources and interconnections, output by the translation tool, is shown in Fig. 3.5. For this architecture, each supported CDFG corresponds to interconnect configuration. If several graphs had been used for this example, each would implement its own connectivity by setting the generated multiplexers appropriately prior to execution. What is shown in Fig. 3.5 is essentially a CDFG directly translated into hardware modules, with the required connectivity to implement data flow. Inter-iteration data flow is implemented by having the first row of FUs read the output registers at the start of every new iteration.

The translation tools for the multi-row architectures (Chapter 4, Chapter 5, and Appendix A) are written mostly in *C*, along with supporting bash scripts. The modulo scheduler for the single-row accelerator architecture (Chapter 6) is currently implemented in MATLAB. Its latest implementation targets the DPR oriented variant of the single-row accelerator design (Chapter 7).

```
Sending operands:
0x1e00:   nput      r4, rfsl0 // Sending 3 operands
0x1e04:   nput      r6, rfsl0
0x1e08:   nput      r5, rfsl0
Reading status:
0x1e0c:   get       r5, rfsl0 // MicroBlaze idles here (fetching completion status)
0x1e10:   beqi      r5, 12
0x1e14:   get       r5, rfsl0
0x1e18:   brki      r0, 440   // If no iterations were performed on the accelerator,
                              // then execution returns immediately to software
Retrieving results:
0x1e1c:   get       r18,rfsl0 // Fetching first result
0x1e20:   msrclr    r6, 4
0x1e24:   get       r6, rfsl0
0x1e28:   beqi      r6, 8
0x1e2c:   msrset    r6, 4     // If carry is computed on the accelerator then
                              // 4 instructions are used to set or clear carry if needed
0x1e30:   get       r6, rfsl0 // Fetching 4 more results
0x1e34:   get       r3, rfsl0
0x1e38:   get       r4, rfsl0
0x1e3c:   get       r5, rfsl0
Return Jump:
0x1e40:   brki      r0, 440   // Execution returns to software
```

Figure 3.6: Example of tool-generated FSL Communication Routine

3.3.3 Generation of Communication Routine

An additional tool generates the Communication Routines (CRs) which implement the communication between accelerator and MicroBlaze. One of the objectives of this work was to achieve a migration process transparent to both the hardware infrastructure (i.e., no processor modifications) and to the application programmer (i.e., no source code modification). To do this, the CRs are generated directly in MicroBlaze assembly and, via a second compilation step, added to the application. In order to generate a CR, the tool is provided with: a list of register file registers which are CDFG inputs, a list of registers which are outputs, and the start address of the Megablock trace.

Depending on the target accelerator, the tool outputs a sequence of MicroBlaze instructions which either read/write to the accelerator's bus address, or send operands and read results via a FSL. Figure 3.6 shows an example FSL-based CR. Single cycle *put* instructions send operands in a known sequence. Since the *get* instruction is blocking, the MicroBlaze idles waiting for the accelerator to output a single result, indicating completion status. The number of iterations to perform on the accelerator is variable per call, so it may happen that only 1 (or zero) iteration(s) is performed, depending on the inputs fed to the accelerator. In these cases the contents of the scratch-pad register, used to check completion status, are recovered (in this case *r5*) and software execution resumes. For bus-based communication, the CRs implement a polling wait.

Results are retrieved one at a time at the end of execution (if more than one iteration is completed). If the implemented loop trace contains operations which alter the carry bit, then the CR either clears or sets it. If multiple output registers are driven by the same CDFG node, then a minor optimization is performed: instead of requiring additional output registers on the accelerator, the

CR performs value assignments between the MicroBlaze registers. It also assigns constant values to output registers if required. Execution then branches back to the Megablock start address.

The CR generation tool is also capable of generating code to invalidate a predefined region of the MicroBlaze's data cache after accelerator execution (for scenarios where data resides in external memory). This invalidation is only performed when the executed Megablock performs *store* operations, as there is no loss of coherency otherwise. Early designs required configurations (e.g., interconnection settings) to be sent to the accelerator by an external mechanism. For these cases the CRs also contained instructions to write a sequence of 32-bit values to the accelerator. In order to be transparently added to the application, these generated instruction sequences are output as data contained within *C* arrays. By setting a *section* attribute and using a custom linker script, the CR can be placed at the expected location in the binary:

Listing 3.2: FSL-based CR in *C* container

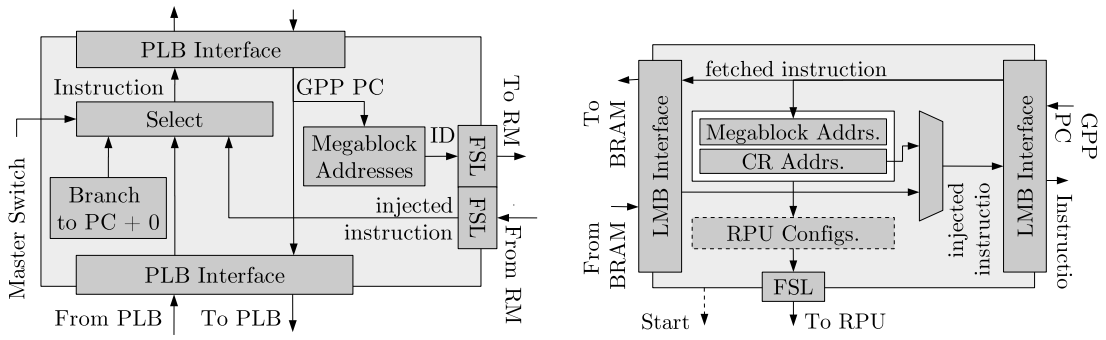
```
uint32 CR_0[18]
    __attribute__ ((section (".CR_seg")))
        =     {
        0x6c04c000, // Sending operands
        0x6c08c000,
        0x6c0ac000,
        0x6c09c000,
        0x6c07c000,
        0x6c03c000,
        0x6c600000, // Reading status
        0xbc03000c,
        0x6c600000,
        0xb6401180,
        0x20000000,
        0x6c600000, // Retrieving results
        0x6c800000,
        0x6ca00000,
        0x6cc00000,
        0x6e400000,
        0xb6401180, // Return
        0x20000000
    };
```

Listing 3.3: Linker Script excerpt to place Communication Routines at known position

```

// Define Memories in the system
MEMORY
{
   bram : ORIGIN = 0x00000050, LENGTH = 0
        x0001FCAF
   seg1 : ORIGIN = 0x0001F9FF, LENGTH = 0
        x00000600
}


(...)


.CR_seg : {
   KEEP (*(.CR_seg))
} > seg1
```

Listing 3.2 shows a smaller CR in a *C* array which, if the linker script definition in Listing 3.3 is used, will be placed at address *1FA00*. A custom linker script is required for two reasons: the *injector* must know the location of the CRs in memory, and the re-compilation stage must not change the location (i.e., addresses) of the trace loops, used registers, or locations of static data. A re-compilation stage is not strictly necessary, as the premise of this approach is that source code should not be modified and/or is not accessible. The re-compilation is employed by the tool chain for convenience, as this step could be replaced with only a linking step, or with the use of the *mb-objcpy* tool, in order to append the CRs to the end of the *ELF* file.

An alternative to solve the first issue would be to inspect the *ELF* after the re-compilation stage and determine the location at which the linker placed the CRs, and generate the *injector* address table accordingly. However, this would not ensure that the remainder of the *ELF* would remain unchanged. This is why the CRs are placed near the end of the available memory. An alternative to this approach would be to integrate the entire trace detection and CR generation into the compilation flow, but this is complex and against the proposed methodology of this work.



(a) PLB version of *injector* module

(b) LMB version of *injector* module

Figure 3.7: Architectural variants of the *injector* module

This is especially true given that a future fully-runtime design is envisioned (including runtime detection and translation), so it is not a viable option to interfere with the compilation tools.

This tool also produces information for the system elements under *auxiliary hardware* (most frequently this is only the *injector*). For the injector, a small Verilog include file is produced with the start address of each Megablock trace, and the location of each respective CR. For one system implementation (shown in Chapter 4), the *injector* was also responsible for sending configuration information to the accelerator. For this purpose, this tool also produced a read-only memory with this content. For an early implementation (also shown in Chapter 4), the configuration information was contained within the secondary MicroBlaze; a *C* file was generated for the application running on this processor. The following section discusses the *injector* in more detail, to provide a better understating of its design and the functioning of the migration mechanism.

3.4 The Injector Module

The variants of the *injector* module, shown in Section 3.4, serve the same primary purpose: to monitor the execution of the MicroBlaze processor and modify the contents of the instruction bus. By doing this, the *injector* controls, in a limited fashion, the execution of the MicroBlaze. Doing so allows for migrating the execution to the accelerator, when a start address of a translated Megablock is detected. Fig. 3.7a shows the PLB version of the *injector*, and Fig. 3.7b the LMB version. This last version has two variants: one which includes accelerator configuration data in an internal memory, and another which does not.

The *injector* operates in the following way: 1) monitoring for an address table match; 2) when a match is found, it replaces the fetched instruction with an unconditional jump to the same address; 3) when the same matches address is seen again, inject into the bus one (in some cases two) instruction(s) which result(s) in an unconditional jump to the respective CR address; 4) idle during accelerator execution; 5) ignore the next occurrence of the matched address, as it will be due to the software execution of last Megablock iteration; 6) return to the monitoring state.

Steps 2 and 3 are verification steps due to the behaviour of the instruction address bus imposed by the MicroBlaze. The MicroBlaze instances used contain a 5-stage pipeline. As such,

the *fetch* request is issued before the instruction is actually executed. This effect, combined with aspects such as the delay of branch instructions and instructions with delay slots means the MicroBlaze will frequently fetch an instruction it will soon after discard. For example, consider a backwards branch located at address X and that a Megablock trace starts at address $X+8$ (instruction addresses advance by 4 bytes per instruction). Due to the fetch behaviour of the MicroBlaze, the Megablock start address would appear, regardless of whether the backwards branch is taken or not. However, execution should only be migrated if the branch is not to be taken, i.e., if the Megablock trace execution is to proceed.

Injecting a branch to the same address as a verification step works by exploiting this behaviour: when the instruction at the Megablock start address is fetched but not executed, it is effectively ignored. Likewise, the branch to the same address which replaces it will be ignored if the Megablock is not to be executed, or taken if the Megablock is to be executed. In the later case, the Megablock start address will be observed twice. This allows for the *injector* to discard false positives.

In order to implement this, the *injector* must identify every branch instruction in the MicroBlaze instruction set, including determining whether or not the instruction has a delay slot, as this increases the window of uncertainty during verification. This verification step introduces a small amount of overhead (typically 3 to 4 clock cycles), which is negligible in nearly all cases. This also means that adapting this migration approach to another processor would require adapting the *injector* to the behaviour of the new instruction set. Also, as mentioned previously, it is impossible to disambiguate multiple trace loops starting at the same address. Finally, the *injector* does not support a MicroBlaze with an instruction cache. Since the cache is internal to the processor, the interface between them cannot be probed. This means the processor would directly fetch trace instructions from the cache and the migration mechanism, in its present state, would not function.

The *injector* has been thoroughly validated and is a very low overhead modification, both in terms of time and resources, which allows for transparent migration without modification of the application binary either offline or at runtime. Preserving the unmodified binary would allow for a single accelerator to be used in a time-multiplexed manner by two processors. While one processor utilizes the accelerator, the second would continue execution by falling back to software.

For purposes of measuring the executing time, the *injector* activates one of the existing counters after introducing the branch into the instruction bus, and disables it when returning to the monitoring state. When the injection of instructions is disabled, the same counter is used during execution of the trace iterations. In this way it is possible to compare accelerated and software-only execution.

3.5 Summary of Accelerator Implementations

This section presents a short summary the following chapters, briefly describing the accelerator architectures implemented, their features, the experimental evaluation and results. Table 3.1 contains a comparison of these implementations.

Table 3.1: Brief comparison of implemented accelerator architectures and results

Accelerator Architecture	Description	Benchmarks	Speedup	Chap.
Automatically generated, multi-configuration, multiple-row accelerator; Rows with forward data directionality, composed of single-function integer units; Per-configuration inter-row connectivity, implementing the equivalent of a CDFG structure	Non-pipelined execution of iterations with exploitation of intra-iteration ILP	15 integer	1.68×	4
	Augmentation of previous architecture with memory access support; Two parallel accesses supported to arbitrary addresses	37 integer	1.60×	5
Automatically generated, multi-configuration, single-row accelerator; Single row of FUs with custom register pool, FU input connectivity, and configuration memory;	Full floating-point support; sequence of per-cycle configuration words exploit ILP and loop-pipelining.	13 floating-point and 11 integer	5.60×	6
	Same accelerator architecture, partitioned for DPR for resource savings	9 floating-point and 11 integer	*N/A*	7
Automatically generated, multi-configuration, multiple-row accelerator; Rows of with forward and backwards data directionally;	Connections between non-adjacent rows; Loop pipelining via dynamic inter-row dependency resolution	12 integer	1.91×	A

In Chapter 4 an initial accelerator design is presented, along with the supporting translation tools. This design takes full advantage of the ILP found in the CDFGs, but does not support pipelined execution nor memory accesses. The design is essentially a template grid onto which the translation tools instantiate any number and type of FUs, to implement the operation parallelism. Three different system-level architectures with different interfaces were used. The implementation platform was a Digilent Atlys board containing a Spartan-6 FPGA. A total of 15 synthetic benchmarks are used mainly to validate the transparent migration process and tools. A geometric mean speedup of 1.69× over software-only execution was achieved for the best case scenario. Performance measurements were taken from the implemented hardware running on-chip.

In Chapter 5 the accelerator is augmented to support acceleration of traces containing memory access operations, and also to support multi-cycle FUs. Data parallelism is exploited by supporting up to two concurrent accesses. The entire data memory of the MicroBlaze is shared with the accelerator via a latency- and overhead-free mechanism. That is, the accelerator does not contain its own data memories, instead directly accessing the only data memory present. This avoids additional overhead due to lengthy data transfer and/or synchronization steps. Furthermore, the

translation process is enhanced to with a list scheduling algorithm which aims to increase the FUs re-utilization across configurations. Along with this, heuristics for the placement and access scheduling of memory operations is applied to reduce the total latency introduced due to memory access. For a total of 37 benchmarks, the geometric mean speedup is of 2.35×. Also, the average power consumption of the entire system decreases by 2.10 % for a measured subset of 7 cases.

In Chapter 6 the latest accelerator design is presented. Unlike previous designs, it relies on a single-row topology to increase utilization of instantiated FUs, and decrease overall resource requirements. The accelerator instances are generated via a modulo scheduling step [Rau94], which is capable of efficient loop pipelining by being aware of the memory access restrictions of the system. The second difference is the support for all single-precision floating-point operations. This implementation is much more efficient in terms of the resource/performance trade-off, relative to the previous. For 24 benchmarks the tool chain can generate accelerator instances which achieve a geometric mean speedup of 5.61×, requiring an average of 960 slices on a Virtex-7.

In Chapter 7 presents the realization of the proposed DPR reconfiguration mechanism for the accelerator. Instead of instantiating the accelerator's reconfiguration capabilities as multiplexing logic, the accelerator is partially reconfigured as needed to switch between supported configurations. The accelerator is partitioned into a static region and reconfigurable region, which contains FUs, interconnections and configuration words. In the evaluation performed, each Megablock is used to generate a variant for this reconfigurable region. For accelerators with larger numbers of configurations the resource savings due to DPR are very significant. Most noticeably, the accelerator requires 2.76× less LUTs than a non-DPR equivalent instance.

Appendix A details a variant of the multi-row accelerator design shown in Chapter 5, augmented to enable pipelined execution. For each supported graph, the accelerator instance contains logic determining all data dependencies all rows of FUs have relative to each other. No tool-level generation of control logic is required, only a bit-encoded parameter specifying from which row each row receives data from. During execution, a row of FU activates once all of its inputs hold valid data. This way, rows activate as quickly as possible in function of how FU are connected. Implementing the row activation logic in this fashion makes it easier to deal with variable memory access latencies in he scenario adopted for this implementation: shared external memory holding data, and a dual-port cache for the accelerator. Also, an improved node and load/store scheduling method is used to increase resource re-utilization and decrease memory access contention. For 12 benchmarks, this design achieves a mean geometric speed up of 1.91×.

Chapter 4

Customized Multi-Row Accelerators

This chapter presents the first accelerator design. The main concern when developing this first iteration was mostly regarding the integration of the accelerator itself with the remaining system. The capabilities of the accelerator were designed according to what was minimally required to successfully accelerate the Megablocks. Structurally, it was based on a relatively straightforward translation of CDFGs into a hardware structure so as to simplify the translation tools.

Rows of FUs are organized according to the topological levels of the CDFGs, and data flows with downward directionality. Along with simplifying the tools, this type of resource layout was also deemed appropriate for a future pipeline-capable version which would adequately use all instantiated resources. The architecture lacks major features such as memory access but nevertheless constitutes for a proof-of-concept of the migration system and is functionally correct in terms of the data results it produces when accelerating Megablocks.

Alongside the accelerator design, the first versions of the two supporting tools possessed the capability of: generating an accelerator specification which could execute multiple loops, reutilizing resources if possible; generating the CRs for transparent communication with between the accelerator and GPP, which includes operand/result transfer and accelerator reconfiguration.

This chapter contains a description of the accelerator architecture in Section 4.1 and an explanation of the supporting tools in Section 4.2. The experimental setup and results are explained in Section 4.3 for three system level architectures employing different interfaces and accelerator reconfiguration methods. Section 4.4 closes this chapter with remarks explaining the limitations of this initial design and the rationale that led to more sophisticated subsequent designs.

4.1 Accelerator Architecture

Like all accelerator architectures presented throughout this work, this first architecture relies on synthesis-time parameters, generated by the developed translation tools, which customize an architecture template. This customization varies the number, layout and type of FUs and their and interconnections. Aspects such as interfaces and control are equal for all instantiations, whilst the

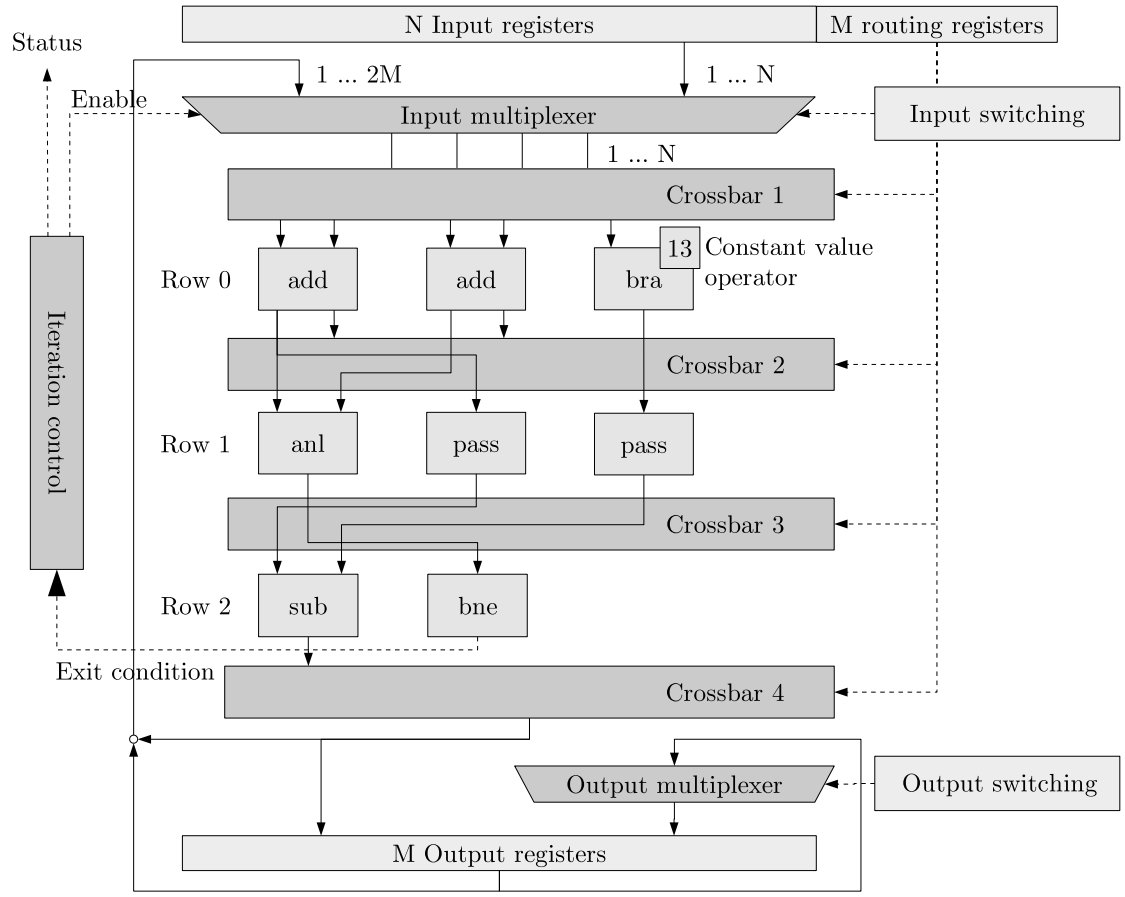


Figure 4.1: Synthetic example of 2D accelerator instance

array of FUs is tailored for a set of Megablock CDFGs. At runtime, the migration and accelerator reconfiguration mechanism utilize the accelerator to execute the target CDFGs. This section presents the architecture template and overall execution model.

4.1.1 Structure

Figure 4.1 shows a synthetic example of the first accelerator architecture, with interface details omitted. This design is essentially composed of rows of FUs of several types, which propagate data downwards via full crossbar interconnects. Each row may contain any number and type of unit. The depth, i.e., number of rows, of the array is also unbounded. Both aspects vary with the CDFGs used to generate the accelerator instance. Data is exchanged only between neighbouring rows. To transport data between distant rows, *passthrough* units are used. All FUs register their outputs, meaning data are propagated row-to-row synchronously, as a group. Data feedback, in order to provide operands to the following iterations, is only performed at the last row of the array. All accumulated data is aggregated, routed backwards and together with the *input registers* constitutes the data available at the start of an iteration.

Typically, the array contains more *passthroughs* in the bottom rows, taking on a triangular shape. This is due to redirecting all of the produced data back into the first row for the following

iteration. The number of *passthroughs* typically increases with each row as data accumulates.

The FUs are single-operation and single-cycle. Each type of FUs corresponds to one CDFG node type (e.g., a single MicroBlaze instructions). This accelerator implementation relied on a set of FUs implementing all 32-bit integer arithmetic (save for division) and comparison operations. The supported arithmetic include any operations involving *carry*, as it is also possible to feed and to retrieve the value of the GPP's *carry* bit into the accelerator. Also supported are pattern comparison and *exit* (i.e., branch) operations. This last class of FUs implements the equivalent of the branch instructions on the Megablocks, and are used to signal end of execution. Unsupported operations in this first design include all floating-point arithmetic and memory accesses.

The accelerator template supports FUs with any number of inputs or outputs (e.g., the 3-input adder with carry). Each input is fed by a multiplexer (part of the crossbars in Fig. 4.1) which fetches all outputs of the preceding row. A possible multiplexer configuration is shown for *Crossbar 2* and *Crossbar 3*. Some trace instructions receive constant input operands, and the Megablock extraction tools also perform constant propagation. As a result some FU input multiplexers are removed (e.g., the *bra* FU in Fig. 4.1). The multiplexers within the array are runtime controllable via writeable configuration registers. The configuration values per supported CDFG to write to the registers on a are also computed by the offline translation tools.

The register file values received from the MicroBlaze when the accelerator is invoked are represented at the top as *input registers*, which remain read-only throughout execution. Likewise, the values to be fed back are stored in a final set of *output registers*. The number of output registers values that are fed back is equal to $2M$: one set produced in the current iteration, and another in the previous. Values in the output register set can also be re-assigned amongst themselves. This emulates the behaviour of re-assigning values between registers on the GPP's register file.

The *Iteration control* module is responsible for controlling the *input multiplexer*. After the first iteration, the respective *enable* bit is set so that new values can be fetched from the feedback wires according to the *input switching* register, instead of the input registers. By counting clock cycles the control module determines when an iteration is completed and controls the *write-enable* of the output registers. Finally, it sets status bits when execution is over.

4.1.2 Interface

Two types of accelerator interfaces are explored. Figure 4.2 shows the interface for the version based on a Processor Local Bus (PLB). The Fast Simplex Link (FSL) version contains the same internal registers, but the interface is a low overhead point-to-point connection. Two types of interfaces allow for observing the impact of communication overhead on performance.

The interface-level registers of the accelerator include an instance-dependent number of input, routing and output registers (N, M and L). The input and output registers contain data inputs/outputs and the routing registers control the inter-row connectivity. The *input multiplexer* and *output multiplexer* are each controlled by a separate register. The *masks* register controls which *exit* FUs

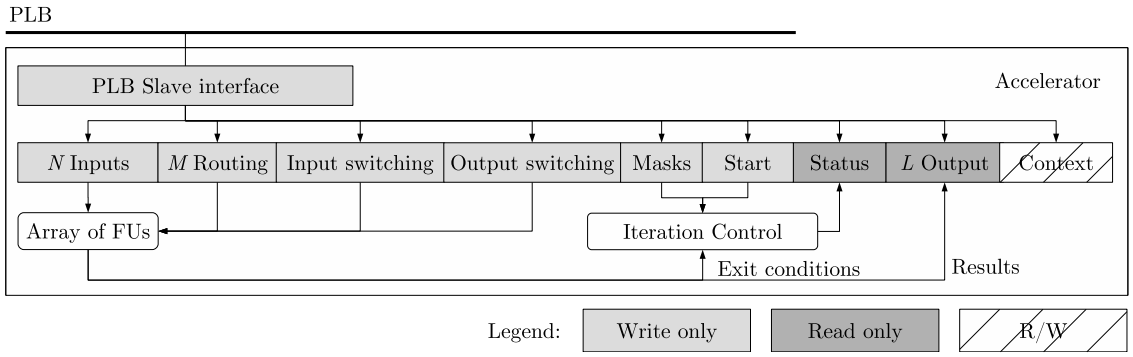


Figure 4.2: PLB type interface for the accelerator

in the array are enabled (for reasons explained below) and the *status* register indicates if the accelerator is busy and how execution terminated. The *start* register is used to begin execution and the two *context* registers are used as scratch-pad memory by the MicroBlaze while it executes the CR.

The number of input and output registers depends on the Megablock traces. Given a set of CDFGs used to generate an accelerator, N and L will be equal to the maximum number of inputs and outputs throughout all graphs, respectively. The number of routing registers depends on the array itself. Each row requires a different amount of routing bits in function of the number of outputs of the previous row. Per row, each output is given a numeric identifier. Each FU input multiplexer of the following row uses a binary number to select any of those values. For simplicity of implementation, the bit-width for all binary coded decimals is the same for all multiplexers, and is determined by the maximum number of outputs throughout all rows. In the case of Fig. 4.1, this would be the first row, with 5 outputs, which leads to 3 bits. Thus to drive all 12 FU inputs a total of 34 bits are needed, i.e., two 32-bit registers. Additional bits are needed for the last multiplexers which needs to drive M registers. In other words, the overhead of invoking the accelerator scales with its width and depth, due to the configuration values that need to be written to it. The *generate* constructs the accelerator relies on connect the specific bits of the registers to the multiplexers, so a single register may hold configurations relative to several rows.

For simplicity, the input and output multiplexers are each controlled by a single 32-bit register for every instance. This imposes a limitation on the joint number of inputs and outputs. For instance, consider that $M = 1$ (as per Fig. 4.1). For the *input multiplexer*, the number of possible choices, per output it drives, is 3: the current $M = 1$ output result, plus the same output from the previous iteration, previous 4, plus one starting value originating from the N input registers. This selection range requires 2 bits which means the total number of inputs supported is $N = 16$.

The *start* register is written while executing the CR, after operands are sent. For the FSL interface version, this is replaced with a signal sent by the *injector*, which is capable of determining when the MicroBlaze has sent all operands by detecting an FSL *get* instruction on the bus, i.e., the MicroBlaze stalls waiting for accelerator results. The *status* register indicates if more than one iteration was performed on the accelerator. If this is not the case, then the context registers are used to recover values into MicroBlaze registers which were used during CR execution.

4.1.3 Execution Model

Execution on the accelerator begins after all configuration information and inputs have been received. Inputs are sent by the MicroBlaze and configuration values via different mechanisms, depending on the accelerator interface (this is further explained in Section 4.3). A configuration sets all the multiplexer selections, which are constant throughout execution. That is, each loop that the accelerator is capable of executing corresponds to one global multiplexer context.

After execution begins there is very little control beyond counting the number of clock cycles required to complete an iteration, and writing output values to the output registers at that point. One iteration is complete after a number of clock cycles equal to the depth of the array. A single row of FU activates per cycle, that is, execution is not pipelined in this initial design. During the first iteration, the multiplexer receives *N* inputs from the *input registers*. After this, the input multiplexer instead fetches some of these values from the feedback lines. The remaining values remain constant throughout all iterations, and are always fetched from the read-only *input registers*.

The array can execute an arbitrary number of iterations. If the number of iterations is determined by a constant in the Megablock trace, this propagates into the accelerator as a constant value operator. In order to terminate execution, the accelerator always has at least one exit condition. The control module receives single bit outputs from all *exit* FUs and when any is true, execution ends. As the previous chapter explained, Megablock execution on the accelerator is atomic: an iteration either fully executes or is discarded. Support for non-atomic iterations would require discriminating which exit condition triggered, recovering the correct set of outputs and returning to a particular software address in the middle of the accelerated trace.

For an accelerator which supports multiple CDFGs, data is still propagated through FUs which may not be in use for a particular configuration. For data FUs this is not an issue since these results are simply not routed to the following rows or registered at the outputs. But the configuration must ensure that only the *exit* FUs relevant to a particular configuration are active. This is done by the 32-bit *mask* register, which disables or enables each such FU. This limits the number of exits allowed on the accelerator to 32. However, no observed combination of Megablocks in the utilized benchmarks exceeded this value.

Once execution completes, the results are read from the *output registers* by the CR and the control module resets the inter-row registers. The accelerator can then be invoked again. If the loop to accelerate is the same as the last one, the configuration process is skipped to reduce overhead.

4.2 Architecture Specific Tool Flow

Figure 4.3 shows the tool flow for generation of the custom accelerators, support architecture and CRs. As the previous chapter introduced, the accelerator generation flow is supported by CDFGs which are produced by the Megablock extractor tool. The tools presented here receive as inputs the Megablocks that are manually selected as acceleration candidates. The outputs of the tools are given to vendor synthesis tools and compilers.

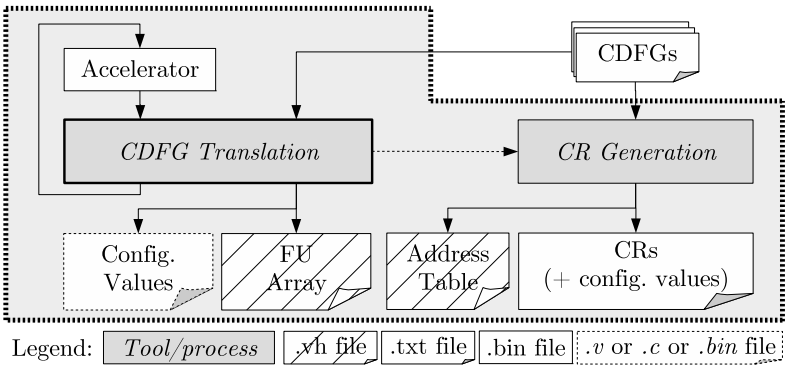


Figure 4.3: Architecture-specific flow for 2D accelerator design and supporting hardware

This version of the translation step processes each CDFG file individually, and produces a binary file with the accelerator specification. For each subsequent CDFG to translate, this file is read and the existing structure is updated. The final run outputs: a Verilog include file with parameters for the accelerator HDL template, and the routing register values per-configuration.

The overall execution flow of this tool is as follows. The CDFG file is parsed and according to the depth and maximum width (i.e., ILP) a number of data structures are pre-allocated. The CDFG nodes are then translated into FUs by assigning them positions on a two dimensional grid. Since unlimited connectivity is assumed, placement is unrestricted. In this implementation nodes are placed in the earliest possible row, and rows are filled from left to right.

Each node results in one FU, as in this architecture re-utilization of FUs only applies across different configurations, i.e., CDFGs. In other words, re-utilization of FUs across configurations is possible if two nodes of the same type in two different CDFGs occur in the same topological level. A single type of FU may support different types of CDFG operations. The most common is the implementation of the MicroBlaze *add* and *addi* (addition with an immediate constant value) via the same FU, since the distinction only exists at the level of the MicroBlaze ISA.

After the placement of CDFG nodes, the auxiliary *passthrough* FUs are placed; if a connection is required between FUs on non-adjacent rows, then *passthroughs* are added to all in-between rows. The tool performs *passthrough* re-utilization at this point; for instance, if two FU require the same output from a FU several rows above, only one chain of *passthroughs* is introduced. This process is implemented by checking the array from bottom to top: as *passthroughs* are added to row N they are themselves checked for the need of another *passthrough* when row $N-1$ is processed. Due to the nature of the CDFGs, passes tend to be created in an inverted pyramid shape. This leads to a frequent re-utilization of *passthrough* between configurations.

At this point, the position for every CDFG node is known, as well as their connections and the bit-widths of each row's crossbar, by analysis of the number of inputs and outputs of neighbouring rows. With this, the number of required 32 bit configuration registers are calculated so that enough bits are available to control all multiplexers.

The output Verilog file produced specifies only the coordinates of each FU to instantiate and interface level aspects such as the number of input, output and routing registers. The *generate*

based Verilog template instantiates as many FU input multiplexers as required, fetching the appropriate control bits from the routing registers. An auxiliary file is also produced and given to the CR generation tool. The number of accelerator interface registers needs to be known in order to compute each register's address for the PLB CR.

Listing 4.1: Reconfiguration information placed in *C* containers

```
#include "graphroutes.h"

// Routing registers for megablock 0
int graph0routeregs[NUMROUTEREGS +
    NUMFEEDBACKREGS]
        = { 0x1110040,  0x58c24c8,  0x8d1,
            0xa5};

// Routing regs array
int *graphroutings[1] = {graph0routeregs};

// Masks for exit FUs
int branchmasks[1] = {0x1};
```

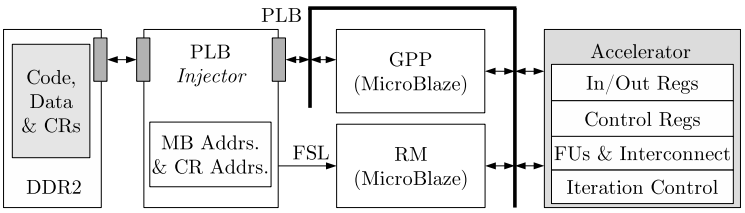
Listing 4.2: Reconfiguration information placed into a read-only memory module

```
module config_bram( clk, rst, addr,
    dataout );

parameter N_REGS = 7;

input                           clk, rst;
input  [clog2b(N_REGS) - 1 : 0] addr;
output reg [31 : 0]             dataout;

reg [31: 0] cfgmem [N_REGS - 1 : 0];

initial begin
    cfgmem[0] = 32'h1150040;
    cfgmem[1] = 32'h2c0d2644;
    cfgmem[2] = 32'h163446;
    cfgmem[3] = 32'h356;
    cfgmem[4] = 32'h0;
    cfgmem[5] = 32'h1;
    cfgmem[6] = 32'h6;
end

always@(posedge clk)
    dataout <= (rst) ? 0 : cfgmem[addr];

endmodule
```

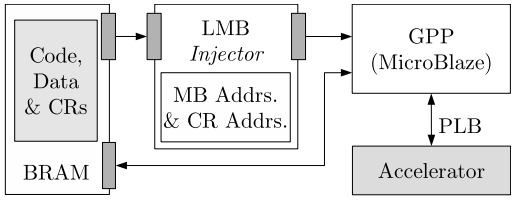
As Section 4.3 will show, this accelerator architecture was integrated into three different system architectures. Based on which system module reconfigures the accelerator, the routing information may be output in several formats. Listing 4.1 shows how this information is produced in order for it to be used by the auxiliary MicroBlaze. The per-configuration routing register values are held in *C* structures which are written via bus to the accelerator. It is also possible to have the CR generation tool include this process into each CR itself (not shown). Finally, Listing 4.2 shows a read-only memory module used to include this information into the *injector*.

Two additional steps are required when translating multiple CDFGs. Firstly, before the placement of new nodes, the existing accelerator's depth (stored in the binary file) is compared to the depth of new CDFG; if the former value is lower than the later, the existing configurations have to be updated by inserting *passthrough* chains which transport data from the previous maximum depth to the new one. This is due to the architectural limitation of this accelerator design which only allows for feedback of values from the very last row of the array. Secondly, previously generated routing register values need to be re-generated if: the width of any row changes (as the number of inputs and outputs varies), or if new *passthrough* are inserted as explained, since routing is necessary through the new rows.

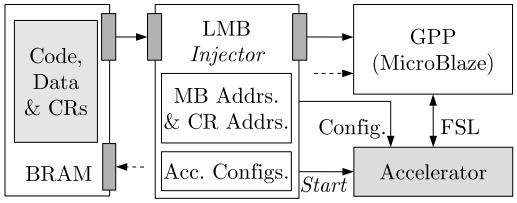
The CR generation process explained in Section 3.3 applies in this implementation: either PLB or FSL based CRs are generated along with the *injector* address table. The only other additional purpose of this tool is to generate the read-only memory modules for *injector* based reconfiguration of the accelerator (as per Section 3.4) or to include this information into the CRs.



(a) System1 - External code/data system memory and PLB based accelerator



(b) System2 - Local code/data system memory and PLB based accelerator



(c) System3 - Local code/data system memory and FSL based accelerator

Figure 4.4: System level variants used for evaluation, with minor accelerator and auxiliary hardware differences

These tools fully execute in the order of seconds, and the runtime scales in proportion to the amount of CDFGs to translate. Most of the runtime is due to file handling. For this implementation, CDFGs with memory access operations or floating-point operations are not supported, and the lack of a more sophisticated node scheduling leaves resource re-utilization under-exploited. In later design iterations, the translation tool is extensively overhauled to address these limitations and to support new accelerator architectures.

4.3 Experimental Evaluation

4.3.1 Hardware Setup

The accelerator underwent minor design modifications through the three experimental validations performed, each relying on a different system architecture. Regardless, all three accelerator implementations are comparable as the major execution model and design is unchanged. Figure 4.4 shows all three systems. They contain the custom accelerator instance, a MicroBlaze processor, and the *injector*. Also, vendor timer modules (not shown) attached to a PLB measure execution times. For these implementations, the timers were enabled, disabled, and read explicitly by the code running on the MicroBlaze.

Figure 4.4a shows the earliest system architecture. For this implementation, the accelerator and MicroBlaze communicate via a PLB. The MicroBlaze accesses external memory to retrieve code and data, and does not use data or instruction caches. Supporting instruction caches would require *injector* level support for the cache interface behaviour and also introduce issues with the migration mechanism. Since the caches are internal to the MicroBlaze, the migration could fail if the instruction to be replaced already resided in the cache.

The *injector* module used here communicates the address of the detected Megablock to an additional MicroBlaze processor. It has three functions. First, it copies the CRs to the external memory at boot so that the main MicroBlaze can later read them. Secondly, it writes configuration information via the PLB to the accelerator. During this time the main MicroBlaze is idle. Afterwards, it sends to the *injector* the CR address relative to a detected Megablock start address. This second processor was used to simplify accelerator reconfiguration and communication, since it is simpler to develop these tasks through software on an early design. Its program code, as well as the accelerator reconfiguration data, are held in local memories (not shown).

The second system design iteration shown in Fig. 4.4b removed the auxiliary processor and instead augmented the CR to include configuration information. If required, it is written to the accelerator during execution of the CR. Also, local memories are used for the MicroBlaze's code and data. The modification was made because: 1) the kernels supported by the current accelerator were too small to justify the overhead of external memory and 2) since the reconfiguration information now resides in the CRs, using a shared external memory between the auxiliary and main MicroBlaze is no longer required. The *injector* is now capable of generating the instructions to place on the bus autonomously, and was modified to interface with the LMB. In certain ways, this simplified the design due to the simpler and deterministic nature of this bus.

Finally, Fig. 4.4c shows a final attempt to reduce the overhead of invoking the accelerator by using an FSL between it and the MicroBlaze, and by performing the transmission of operands and configuration values in parallel. The *injector* now contains a local memory with the configurations which it sends via a separate FSL to the accelerator.

The development board used for this evaluation was a Digilent Atlys, which contains a Spartan-6 LX45 FPGA and an external 128 MB DDR2 memory. The MicroBlaze version used was *8.00a*, set for *Performance* and with the integer multiplication, barrel-shifter and pattern-comparison units enabled. Xilinx's *EDK 12.3* was used for system synthesis and bitstream generation. All generated systems were fed with a 66 MHz clock. The same clock signal is used for all system modules, accelerator included, in nearly all cases. Exceptions are explained in the following sections.

4.3.2 Software Setup

A total of 15 code kernels were used for this evaluation. The kernels employed for this evaluation are simple single-file applications which call a kernel function a given number of times, N. For these experiments, $N = 500$ for all cases. The kernels used are synthetic examples of simple data-processing loops. They contain no floating-point operations or memory accesses, and operate on 32-bit integer values. The input data given to the kernels are statically stored as a global array(s).

Since the applications are simple, only one candidate Megablock, corresponding to the kernel function, was extracted from each. In order to evaluate accelerators with several configurations, two additional applications which call 6 code kernels were used (*merge1* and *merge2*). In other words, two of the evaluated accelerators support 6 configurations. For both cases an evaluation was made of the overhead of calling each kernel N times in sequence (*m1* and *m2*), and calling each kernel a total of N times alternatively (*m3* and *m4*).

In order to measure execution times a single timer module was enabled through software before the kernel calls, and read afterwards. The output data is also read so as to compare the execution of the software and accelerated runs. By calling each kernel $N = 500$ times, the small C level overheads of enabling the timer and of function calls are amortized. Listing 4.3 shows the (simplified) code for one of the used benchmarks. The *for* loop within the function call is the accelerated portion in this case.

Listing 4.3: Simplified code for *even ones* benchmark

```
1 ||int evenOnes(int temp, int Num) {
2 ||
3 ||    for(int cnt = 0, int i = 0; i < Num; i++) {
4 ||        cnt ^= (temp & 1);
5 ||        temp >>= 1;
6 ||    }
7 ||    return cnt;
8 ||}
9 ||
10 ||int main() {
11 ||
12 ||    int i, result;
13 ||
14 ||    tmrStart(&XPS_Timer);
15 ||
16 ||    for(i = 0; i < 500; i++)
17 ||        result += evenOnes(i,32);
18 ||
19 ||    tmrStop(&XPS_Timer);
20 ||
21 ||    int cycles = tmrRead(&XPS_Timer, 0);
22 ||    printf("time:%d\r\n", cycles);
23 ||    printf("res:0x%x\r\n", result);
24 ||    return 0;
25 ||}
```

This method of performance measurement is intrusive and rudimentary but not a critical issue given the mostly synthetic nature of the benchmark code. The implementations in the following chapters employ more sophisticated and application-transparent methods to retrieve these metrics.

The applications were compiled with *mb-gcc 4.1.2* using the *-O2* flag and additional flags which enable the use of barrel shifter, integer multiplication and pattern comparison instructions. The CDFG extraction process did not employ loop unrolling for most cases.

4.3.3 Characteristics of the Generated Accelerators

Table 4.1 lists all kernels, along with a shorthand alias. From each benchmark a single Megablock trace was accelerated. The third column shows the number of MicroBlaze instructions in the trace, and the next column shows the average number of iterations performed per occurrence of the trace. Given the simple nature of the target loops, the resulting traces contain few instructions. The highest number of instructions occurs for *i13*, since the detected trace is of an unrolled inner loop. Despite the small number of instructions, the CDFGs for these traces have an ILP of up to 2.9 on average. Most loops execute a constant number of times (32 or 16). Kernels *i3*, *i7*, *i9*, *i10*, *i11*, and *i12* iterate a number of times dependant on the input values. For example, for *i3* the number of iterations is an arithmetic progression of an input value.

Table 4.1: Extracted Megablock and generated accelerator characteristics

ID	Kernel	# Trace Insts	Avg # Iterations	IPC_{SW}	#FUs	#Passes.	#Rows	IPC_{HW}
i1	count	6.0	32.0	0.86	6.0	6.0	3	2.00
i2	evenones	6.0	32.0	0.86	5.0	4.0	3	2.00
i3	fibonacci	6.0	249.5	0.86	4.0	6.0	3	2.00
i4	hamdist	6.0	32.0	0.86	6.0	11.0	3	2.00
i5	popcnt	8.0	32.0	0.89	8.0	7.0	3	2.67
i6	reverse	7.0	32.0	0.88	7.0	9.0	3	2.33
i7	compress	8.0	17.3	0.89	8.0	21.0	4	2.00
i8	divlu	5.0	31.0	0.83	5.0	4.0	3	1.67
i9	expand	8.0	17.3	0.89	8.0	21.0	4	2.00
i10	gcd	8.0	41.3	0.89	8.0	17.0	6	1.33
i11	isqrt	6.0	16.0	0.86	6.0	9.0	3	2.00
i12	maxstr	4.0	30.0	0.80	4.0	6.0	3	1.33
i13	popcount3	31.0	500.0	0.91	18.0	33.0	9	3.44
i14	mpegcrc	15.0	31.0	0.94	14.0	32.0	7	2.14
i15	usqrt	18.0	16.0	0.95	17.0	42.0	8	2.25
	mean	9.5	74.0	0.88	8.3	15.2	4.3	2.08
m1	merge1	6.3	159.5	0.86	16.0	12.0	3	2.10
m2	merge2	6.8	29.0	0.87	24.0	35.0	6	1.64

The next column, IPC_{SW}, represents how many Instructions per Clock Cycle (IPC) cycle the MicroBlaze is capable of executing. Most MicroBlaze instructions are single-cycle, but branch instructions have either 2 or 3 cycles of latency. Each trace has at least one backwards branch instruction. This means less than one instruction is executed per clock cycle throughout a complete execution of a single trace iteration. The values shown for IPC_{SW} hold only for *System 2* and *System 3*, where there is a 1 clock cycle latency for instruction fetch.

The following columns contain the characteristics of the generated accelerators. The *#FUs* column shows the total number of units in the array, including *passthrough* units, while the next accounts only for the *passthroughs*. Due to the array structure and to the amount of data to transport across iterations, *passthrough* units frequently outnumber other FUs. On average, there are 1.67 *passthroughs* per each other FU. However, they are frequently re-utilized in multi-configuration cases, since the same behaviour occurs while translating any CDFG.

The *#Rows* column shows the number of rows in the array. Since the CDFG nodes are implemented by single-cycle FUs, this means that the Critical Path Length (CPL) of the CDFG determines the array's depth. Rows execute one at a time, so the IPC on the accelerator, IPC_{HW}, is simply the number of trace instructions over the array's depth. Scheduling operations onto different rows (e.g., via list scheduling) would have no impact on performance, but there is a potential impact on resource usage, as the next chapters will show. Ignoring any overheads, speedups are

obtained when IPC_{HW} is larger than IPC_{SW}. As a general rule, this architecture performs best when executing CDFGs whose number of instruction is high, and whose CPL is low.

The value of IPC_{HW} is also indicative of how many units are not idle per cycle. Inversely, it is possible to calculate how many are inactive per cycle. Considering the number of FUs in each array, the number of rows it has, and its IPC_{HW}, this results in an average of 6.4 units being idle per cycle (not including *passthroughs*). Improving the per-cycle usage of resources for 2D accelerators was address in later implementations.

The number of FU in a single-loop array is, at most, equal to the number of MicroBlaze operations in the originating trace. Due to extraction and translation level optimizations, the number of FU, excluding *passthroughs*, is usually lower. For example, the 31 instructions of the loop trace from *i13* are implemented by 18 FUs. Constant propagation allows for removal of some operations, especially when the MicroBlaze's *imm* instruction is used. This instruction loads a special register with the upper 16-bits of a 32-bit immediate operand. The behaviour of these operations is implemented as constant inputs when specifying FU input multiplexers.

The *mean* row of Table 4.1 contains arithmetic averages for these parameters, and only includes benchmarks *i1* to *i15*. For *m1* and *m2*, several kernel functions are called. Benchmark *m1* calls the kernel functions *i1* to *i6*, and *m2* calls kernels *i7* to *i12*. For these two cases, the values shown for the number of trace instructions, number of iterations and IPC_{HW} are weighted averages which consider the individual kernels composing each case. Aspects such as the resulting number of FUs and depth of the array are dictated by the maximum of each individual case. So, for instance, an array supporting both *i7* and *i8* will have a depth of 4. This causes the total number of required *passthrough* units to increase for these cases. Also, the current execution model enforces that all rows are activated before an iteration is completed. This means that this accelerator architecture causes some CDFGs to under-perform due to artificially increasing their CPL.

This was taken into account when grouping kernels for *m1* and *m2*. In *m1* all the kernels had a depth of 3 so the resulting resource cost of the accelerator could be evaluated in a scenario where the execution of the individual CDFG would not suffer a performance impact. Benchmarks *i13*, *i14* and *i5* are left out of these groups since their CDFGs are considerably larger than all other cases. Benchmark *m2* groups all remaining cases. The general rationale for these groups was that similar CDFGs should be grouped together, since this would promote more resource re-utilization. Finally, a specific strategy for choosing kernels to group was not wholly important, as the main objective at this point in development was to verify the functional correctness of a multi-loop accelerator, both at an architecture and translation tool level.

One of the objectives of creating multi-configuration accelerators is to reuse units between configurations, ideally resulting in a resource usage lower than the sum of resources that would be required by deploying several single-loop accelerators. For *m1* and *m2*, the sum of FUs (excluding *passthroughs*) of their respective individual cases is 36 and 39. The two combined accelerator instances instead contain 16 and 24 FUs, even though the re-utilization is limited to nodes within the same topological level. Interpreted another way: the number of FUs required for *m1* and *m2* is 2.0× and 3.0× higher than the maximum number of FUs required amongst the respective

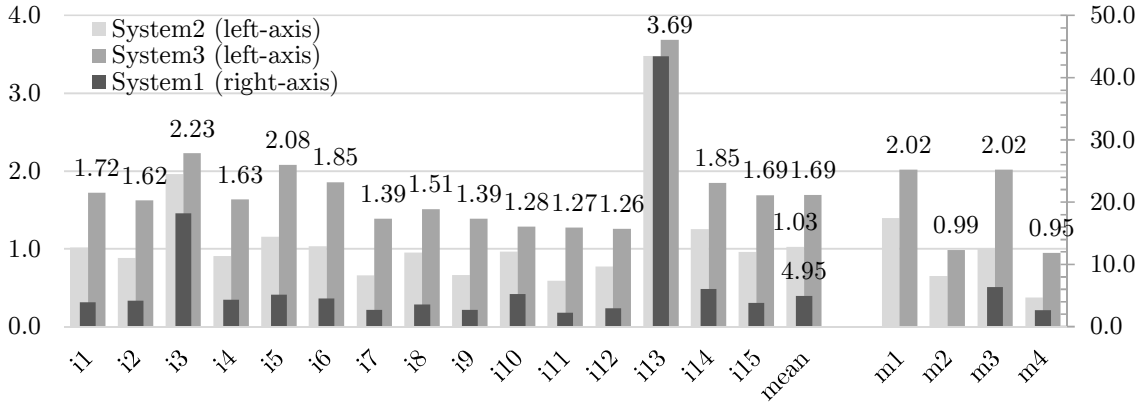


Figure 4.5: Speedups for several types of system architectures vs. a single MicroBlaze processor. Labels shown for *System 3*

individual benchmarks. The *passthrough* units are more frequently shared between configurations. Using the same metric, the number of *passthroughs* required for *m1* and *m2* is 1.1× and 1.7× higher than the maximum for the respective single configuration accelerators.

The average IPC_{HW} of each loop supported in *m1*, when each executes in its respective single-configuration accelerator, is 2.17. This does not change when executing the same loops in the multi-configuration accelerator of *m1*. In contrast, for *m2*, the average IPC_{HW} of the individual cases, when each executes in a dedicated accelerator is 1.72. The IPC_{HW} for the same loops on the accelerator in *m2* is 1.08, since some configurations are penalized by the higher CPL.

4.3.4 Performance vs. MicroBlaze Processor

Fig. 4.5 shows all measured speedups for all three system architectures employing the same accelerators. The speedups for each case scale depending on the overhead imposed by each system's interfaces. The speedup mean shown is geometric and includes benchmarks *i1* to *i15*. Labels shown are for *System 3* (the best results for a realistic baseline).

Results for *System 1* are shown on the right axis for readability. Speedups are much higher for this system since the software baseline is very pessimistic: code and data reside in external memories and the MicroBlaze does not have either type of cache. It was measured that an average of 23 clock cycles are required to retrieve an instruction via PLB. The version of the bus used did not support burst mode. Executing several operations found in a single accelerator row requires only 1 clock cycle, hence the marked speedups despite the CR overhead still incurred due to PLB based communication with the accelerator for reconfiguration and data transport. Given that software execution is so hindered in this scenario, the acceleration results should not be considered as realistic. Regardless the fully functioning system provides the proof-of-concept for the migration approach, architecture and tools.

The remaining two systems can be considered realistic scenarios. For *System 2*, the geometric mean speedup is of 1.03×, and slowdowns occur for 9 out of the 15 cases. For the cases where a

speedup is possible the geometric mean is of 1.48×. Slowdowns occur if the time spent on the accelerator plus the CR execution time exceeds the software-only execution of the loop in question. Given the small number of iterations performed for these test cases per accelerator call, the overhead becomes very significant. Consider only kernels *i1* to *i6*. They all perform the same average number of iterations (save for *i3*), but two other factors cause minor performance variations. For instance, between *i5* and *i6*, it is the higher IPC_{HW} possible for *i5* which justifies the performance difference. This in turn is due to the trace for *i5* containing a single additional instruction relative to *i6* coupled with the fact that the number of accelerator rows is equal. For these two cases, the communication overhead introduced is equal: the CRs contain the same number of instructions. When comparing *i1* and *i2* we find that the IPC_{HW} is the same as well as the average number of iterations. The CR for the former requires 108 clock cycles to execute (excluding the time required to reconfigure the accelerator, which in this implementation is embedded in the routines) and the later 139. This results in a speedup for the former and a slowdown for the later. The difference between CR execution times is small and only reasonably relevant due to the small number of iterations, but this comparison helps to demonstrate the effect of overhead on speedup.

For *System 3*, the use of dedicated links for communication reduced both the data transfer latency and the number of instructions in the CRs themselves due to the different types of instructions required. The resulting geometric mean speedup is of 1.69× and no slowdowns occur. As the impact of overhead diminishes, the achievable speedup approaches the IPC_{HW} value.

Finally, consider the benchmarks *m1* and *m2*. The weighted geometric average of the speedups for *i1* up to *i6* is 2.04× for *System 3*. The speedup for *m1* for the same system is 2.02×. That is, the acceleration achieved for each individual loop does not decrease relative to their single-loop accelerators, since each loop's IPC_{HW} does not decrease. The marginal performance decrease is due to additional reconfiguration overhead. Performing the same comparison for *m2*: the weighted geometric average for *i7* up to *i12* is 1.34× for *System 3*, but the measured speedup for *m2* is of 0.99×. In other words, the individual speedups of each accelerated loop possible in *System 3* are lost when using this multi-loop accelerator. This was already associated to the decrease in IPC_{HW} but this accelerator (for *m2* and *m4*) also happens to suffer from higher reconfiguration overhead as well. The impact of overhead on performance, especially for *m3* and *m4*, are discussed shortly.

In short, the attainable speedup is influenced by two major parameters: the IPC_{HW} achievable on the accelerators and the communication overhead. The speedup is higher with a higher IPC_{HW}, and a low overhead prevents degradation of the performance gains of the accelerators. The IPC_{HW} is a function of the CDFGs and mostly of the accelerator architecture. For this evaluation, the variable parameter is the overhead of the several systems. Table 4.2 contains both the per-benchmark overhead for *System 2* and *System 3* and also the number of instructions and number of clock cycles required to execute the CRs based on the interface type.

The second column shows the number of MicroBlaze instructions in each PLB-based CRs. On the left hand side is the number of instructions which execute if reconfiguration of the accelerator is not necessary, and inside the parenthesis is the additional number of instructions needed otherwise. Likewise, the next column shows the respective number of clock cycles required considering

Table 4.2: Communication Routine characteristics and Overheads for Systems 2 and 3

ID	# PLB CR Inst.	# Overhead Cycles	System 2 Overhead	# FSL CR Inst.	# Overhead Cycles	System 3 Overhead
i1	27 (+24)	108 (+91)	53%	12	12 (+4)	11%
i2	34 (+25)	139 (+92)	59%	18	18 (+3)	16%
i3	27 (+21)	124 (+80)	14%	14	14 (+2)	2%
i4	35 (+25)	132 (+92)	58%	17	17 (+4)	15%
i5	35 (+25)	132 (+92)	58%	17	17 (+4)	15%
i6	35 (+25)	132 (+92)	58%	17	17 (+4)	15%
i7	35 (+33)	148 (+139)	68%	19	19 (+6)	22%
i8	25 (+17)	90 (+68)	49%	10	10 (+3)	10%
i9	35 (+32)	148 (+115)	68%	19	19 (+5)	22%
i10	32 (+29)	113 (+104)	31%	15	15 (+6)	6%
i11	34 (+24)	123 (+91)	72%	16	16 (+4)	25%
i12	25 (+21)	90 (+80)	50%	10	10 (+4)	10%
i13	37 (+39)	142 (+161)	3%	18	18 (+9)	0%
i14	36 (+42)	157 (+172)	42%	20	20 (+8)	8%
i15	31 (+49)	160 (+195)	56%	18	18 (+11)	12%
mean	32 (+29)	129 (+111)	49%	16	16 (+5)	13%
m1	32.2 (+32.8)	127.8 (+138.8)	38%	15.83	15.8 (+6.7)	7%
m3			57%			10%
m2	31.0 (+53.2)	118.7 (+223.2)	44%	14.83	14.8 (+15.2)	9%
m4			69%			16%

local code and data memory only. For *System 3*, operands are sent by executing the CR, and reconfiguration is performed in parallel by the *injector*. The left-hand value in the sixth column is the number of clock cycles required to execute the CR alone, and the value in parenthesis is the number of cycles the *injector* requires to reconfigure the accelerator.

For nearly all cases, each CR executes 500 times. The only exceptions are *i13* and *m1* to *m4*. For the former the CR executes only once, and for the former cases a total of $500 \times 6 = 3000$ CR executions are required. The resulting overhead is shown for both cases as a percentage of clock cycles spent from the moment the injector intervenes until the return to software.

The lowest overhead for all cases occurs for *i13*, for a particular reason. While all other kernels are called $N = 500$ times, this function was in-lined into the calling location. Since the kernel loop only iterated three times, it was unrolled. This resulted in a larger Megablock trace which comprised all $500 \times 3 = 1500$ iterations of the inner loop. In this case, a single iteration on the accelerator corresponds to the three unrolled iterations of the inner loop. Since the accelerator is only invoked once, the CR execution penalty is only incurred once. For this case it is observable how a low overhead increases the speedup to a value close to the $\mathrm{IPC}_{\mathrm{HW}}$.

As was already explained, *System 1* (not shown in Table 4.2) suffers the most overhead. Considering the number of cycles to return to software after the *injector* intervenes, the average com-

munication overhead for this system (excluding *m1-m4*) accounts for 84 % of the time. Since CRs are in external memory, the communication time is high relative to accelerated time. For *System 2*, all code and data reside in local memories. The average communication overhead for this case is 53 %. For this case, slowdowns occur frequently since the low number of iterations per accelerator call does not amortize the communication overhead now that MicroBlaze execution is realistically efficient. The average number of iterations performed on the accelerator per call is 74.0, as shown in Table 4.1. Given the average number of rows, this means the accelerators execute for approximately 320 consecutive clock cycles. In comparison, if reconfiguration is not required, a PLB-based CR contains an average of 32.2 instructions, which require 129 clock cycles to execute. Finally, for *System 3*, the average communication overhead is of 26.5 %. Writing configuration values to the accelerator in parallel with CR execution allows for mitigation of some overhead, but most of the reduction in these experiments is due to the use of the FSL-based CR.

Consider again cases *m1* (*m3*) and *m2* (*m4*). For *m1* and *m2*, the 6 called loops are executed sequentially. The accelerators for *m3* and *m4* are equal to *m1* and *m2*, respectively, but the supported kernel functions are called differently. The former case is the one for which overhead is lowest: all calls of each supported loop occur before the following (i.e., the loop for *i1* is called 500 times, followed by *i2*, etc). This means that the accelerator only needs to be reconfigured 6 times. For the later case, the first supported loop is called once, followed by the second, etc. That is, the accelerator needs to be reconfigured per call, introducing the most overhead.

Also, the CRs are affected by accelerator size and number of configurations, since this affects the amount of reconfiguration values. For instance, execution of the kernel from *i1* on the accelerator for *m1* may incur more overhead when executing the respective CR, relative to the accelerator for *i1*. The accelerator for *m1* requires 1.35× the MicroBlaze instructions for PLB-based reconfiguration alone (i.e., excluding transfer of operands/results), relative to the average of the respective individual cases. For *m2*, the increase if of 2.04×. This corresponds to an increase of 1.22× and 1.56× of the number of clock cycles required to execute an CR. For the FSL-based equivalents, the increase in number of clock cycles is similar: 1.15× and 1.55× for *m1* and *m2* respectively.

Given these two factors (i.e., how often reconfiguration is required and how lengthy the reconfiguration process is), it is observable when comparing *m1* and *m2* to *m3* and *m4*, respectively, that despite additional reconfigurations the speedup is not affected when FSL-based CRs are used (e.g., *System 3*). For instance, for *m1* and *m3*, the PLB-based communication corresponds to 38 % of total execution time for the former case and 57 % for the later. For the FSL case the increase is much lower, from 7 % to 9 % clock cycles. This holds for *m2* and *m4*, but the additional overhead reconfiguration is higher since the respective accelerator requires more configuration information.

Considering all these factors, a speedup estimation on a per-loop basis can be made using Eq. (4.1). The number of accelerated MicroBlaze instructions is represented by N_{rInsts}, and the number of required FUs (*passthroughs* excluded) by N_{rFUs}. The $N_{rHWCycles}$ and $N_{rSWCycles}$ parameters are the number of cycles required to execute the Megablock fully in software and hardware respectively. The number of iterations performed on the accelerator is given by N_{rit} and the number of overhead clock cycles by OH_c. This factor must include the CR execution, the number

of cycles required to execute the last iteration of the loop through software, and the small overhead introduced by the *injector*-driven migration. The overhead lowers the attainable speedup by increasing the denominator in the ratio between the software and accelerator IPC.

$$\text{Speedup} \simeq \frac{N_{\text{rInsts}}}{N_{\text{rFUs}}} \times \frac{\frac{N_{\text{rInsts}}}{N_{\text{rSWCycles}}}}{\frac{N_{\text{rFUs}}}{N_{\text{rHWCycles}}} + \frac{\text{OH}_\text{c}}{N_{\text{it}} \times N_{\text{rFUs}}}} \tag{4.1}$$

Alternatively, Eq. (4.2) directly uses the IPC values computed as presented in Table 4.1.

$$\text{Speedup} \simeq \frac{\frac{1}{\text{IPC}_{\text{SW}}}}{\frac{1}{\text{IPC}_{\text{HW}}} + \frac{\text{OH}_\text{c}}{N_{\text{it}} \times N_{\text{rFUs}}}} \tag{4.2}$$

This formula has an average estimation error of 2.6 % and 2.2 % for *System 2* and *System 3*, respectively. Using this formula the maximum potential speedup can also be estimated by considering that OH_c is zero: the maximum geometric mean speedup for these benchmarks would be 2.51×. Note that to account for the external memory effect in *System 1* the IPC_{SW} would have to be computed accordingly. Also, this formula is valid for this accelerator execution model, where the total number of FUs on the array (used for a given configuration) corresponds to the total number of operations to execute.

4.3.5 Resource Requirements and Operating Frequency

The MicroBlaze processor used for these systems (no floating-point unit and no caches), requires 1359 and 1068 LUTs and FFs, respectively, according to the synthesis reports. By using this as a metric, it can be found that the average accelerator requires 2.66× the LUTs and 1.36× the FFs a MicroBlaze requires. The number of required LUTs and FFs for all cases, normalized to the cost of a MicroBlaze, is shown in Fig. 4.6, along with the reported synthesis frequency.

The entire system (accelerator, MicroBlaze, buses, etc), requires an average of 5375 LUTs and 2777 FFs, according to post place and route reports. In other words, the accelerator corresponds approximately to 62 % and 51 % of the system's total resources on average. The accelerators do not require any BRAMs. The resource-related averages presented do not include *m1* and *m2*.

The number of required LUTs and FFs scales in a reasonably linear fashion with the number of FUs. Each FU incurs a LUT cost and requires adding a new *O-to-I* multiplexer where *O* is the number of outputs in the previous row and *I* the number of FU inputs. Its output(s) must also be stored in 32-bit registers. The *passthrough* units only represent additional FF cost, which also scales linearly with the number of these units. The correlation coefficient between LUTs and FUs is of 0.91, and 0.97 between FFs and FUs. The accelerators for benchmarks *i13*, *i4* and *i15* require the most resources, given that they contain the most FUs and also have the highest number of rows.

For nearly all cases, the critical path usually involves an adder carry chain. Variations mostly involve the multiplexer complexity. For instance, for *i2*, *i3*, *i4* and *i5* the critical path is determined by an adder carry chain, but the multiplexer feeding one of the inputs is more complex in *i3* relative to the other cases. Four cases have their maximum frequency determined by other elements; for

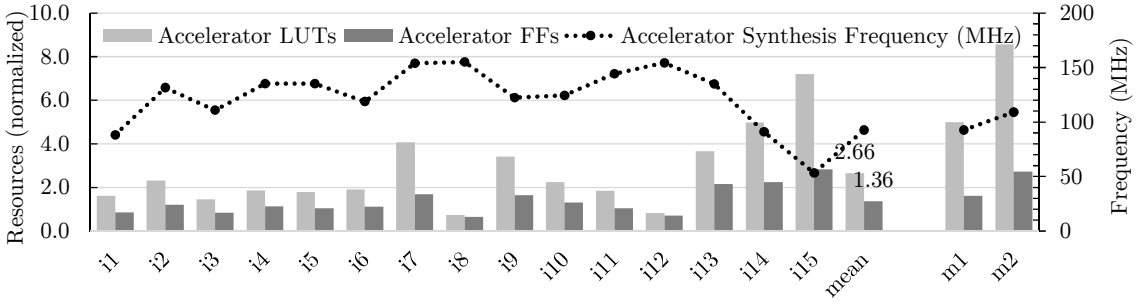


Figure 4.6: Synthesis frequency and resource requirements of the generated accelerators, normalized to the resource requirements of a single MicroBlaze

i1, this is due to the complex barrel shifter unit; for *i7*, the critical path is related to a much simpler constant shift unit; and for *i14* and *15*, the frequency is determined by a multiplication unit. It is difficult to determine the reason for the difference between *i4* and *i5*, since the synthesis reports only state that most of the delay in the critical path is due to the DSP unit (which implement the multiplication FU). The combinatorial delay through the DSP is different for both cases, and the registers which feed the DSP have a much greater fanout in *i15* relative to *i14*.

Only the accelerator for *i15* had a synthesis frequency below the target frequency of the system (66 MHz), so this case was implemented at 33 MHz, for purposes of functional validation. The systems for *m1* and *m2* were also implemented at 33 MHz despite the higher synthesis frequency of the accelerators. This has an obvious implication on speedups, since the actual time required to execute the loops increases relative to a non-accelerated MicroBlaze running at 66 MHz. However the conclusions drawn previously regarding execution performance hold, as this is technological and not an approach-related issue. Later chapters improve the accelerator architecture regarding both resource usage and frequency. Also, this validation used an FPGA geared towards low-power applications, with a small amount of resources, which hinders synthesis of large systems.

It is expected that supporting multiple loops increases resource consumption and may decrease synthesis frequency, as more FUs incur a cost by themselves and, especially, by increasing the width and depth of the array may introduce an ever higher cost associated with connectivity. For *m1*, the number of LUTs used by the accelerator is 2.15× higher than the maximum amount of LUTs used by the accelerators of cases *i1* to *i6*. Likewise the number of FFs is 1.34× higher. However, the amount of LUTs and FFs are only 0.46× and 0.26× as much as is required by the total sum of individual accelerators, respectively. This highlights the savings versus deploying multiple single-loop accelerators. For *m2*, the accelerator requires 0.65× and 0.39× the amount of LUTs and FFs relative to the sums used by *i7* up to *i12*.

Relative to frequency, the critical path for *m1* is found between the *input multiplexer* of the array, the first row's multiplexer and a barrel shifter unit in the first row. It is essentially the same critical path as *i1*. However, the synthesis frequency of 92.7 MHz is actually marginally higher than that case (the minimum of the respective individual cases). For *m2*, a path through the *input multiplexer* and an adder carry chain leads to a frequency of 109 MHz, which is 0.89× lower than

the minimum of the individual cases. Given that this is still above the system clock frequency, and that both the MicroBlaze and accelerators operate at the same frequency, this decrease is not significant. In a system where the overall operating frequency was regulated based on the maximum accelerator frequency, eventual decreases due to supporting multiple loops in a single accelerator would have consequences on performance.

The previous section discussed how the CRs may increase in size due to additional interconnects to reconfigure as more resources are added, which happens in these two cases supporting multiple loops. As a comparison, the number of total multiplexer configuration bits required by *m1* and *m2* are 208 and 436. This is an amount 2.08× and 2.42× higher than the maximum number of bits required amongst the respective individual accelerators.

As a final note consider the cost of the *injector* per implemented system. For *System 1* the PLB-type interface introduces a total cost of 239 LUTs and 78 FFs. For *System 2*, the different interface requires less logic, but now the *injector* is more complex internally, resulting in 164 LUTs and 45 FFs. For *System 3*, the internal memory holding accelerator configurations requires 1 BRAM and the remaining injector logic has a cost of 209 LUTs and 72 FFs.

4.4 Concluding Remarks

This chapter presented initial design iterations on fully functional transparent binary acceleration systems relying on automated hardware/software partitioning. The general approach was validated together with the capabilities of the translation tools. Experimental evaluations were conducted using on-chip implementations for execution time measurements.

Three system types were studied in the process of determining the more appropriate environment for further accelerator design iterations. The design for *System 1* seems excessive, but note that it was initially conceived with the objective of eventually porting most of the hardware/partitioning workload to runtime. Hence the auxiliary MicroBlaze which would be responsible for Megablock processing and configuration generation for a runtime reconfigurable accelerator architecture. Initially relying on external memory was also forethought towards the support of larger applications for future experimental evaluation.

For the same reason, the accelerator architecture was oriented towards a 2D grid to simplify the task of future runtime embedded translation tools. The use of crossbars provides a simpler first approach in this scenario as it would be more difficult to generate an interconnect structure at runtime versus generating configuration information for a rich static interconnect.

As the next chapters show, the development steers away from this direction, and continues to rely on offline generated CDFG information to focus on accelerator architecture improvements. The next two chapters particularly address limitations and issues identified during this initial experimentation: 1) re-utilization of resources between configurations required improvement, 2) idle time of FUs within a single configuration had to be diminished, 3) memory support was critically required to increase applicability and increase performance. The next chapter deals essentially with this last point.

Chapter 5

Accelerators with Memory Access Support

This chapter augments both the accelerator and system level architectures to enable the accelerator to access the MicroBlaze's data memory. It was observed, during validation of the first accelerator design, that lack of support for direct memory access by the accelerator greatly limits applicability of the approach, since it prevents acceleration of realistic applications. Namely, the accelerator was incapable of treating data in a stream-like fashion, processing successive array elements per iteration. Instead, in order to process data, one or more data array elements were passed as register file operands per accelerator call. This meant that to iterate through an array with N elements, N accelerator calls were required (generally). The inner loop iterations of each call processed a single datum per data array involved in the kernel. This is unlike realistic streaming applications were the inner loop iterates through the arrays, via sets of paired memory read and subsequent memory write operations. This chapter retains the overall 2D accelerator design presented and introduces the use of *load* and *store* FUs which, via a bus sharing mechanism, can read/write the entire range of the MicroBlaze's local data memory. The outer accelerator layer is equipped with only two memory ports, which exploit access parallelism by using dual-ported BRAMs. Since there can be an arbitrary number of *load/store* FUs, two methods to arbitrate access to the ports are used: a runtime heuristic, and a tool-generated static schedule. The accelerator also underwent a number of optimizations aiming to reduce resource consumption: the use of crossbars is replaced with tailored multiplexers with minimal connectivity and the use of *passthrough* units is optimized.

The accelerator architecture is described in Section 5.1. Section 5.1.2 specifically explains the handling of memory access units, and the system-level memory sharing mechanism. The redesigned translation tools are explained in Section 5.2. An evaluation is presented in Section 5.3 using a set of 37 benchmarks which includes: a performance comparison with software-only execution, an analysis of the effects of memory access optimization and list scheduling, comments on the resource requirements and operating frequency, and finally a short power consumption analysis. Section 5.4 presents final remarks on the achieved improvements and still existing limitations.

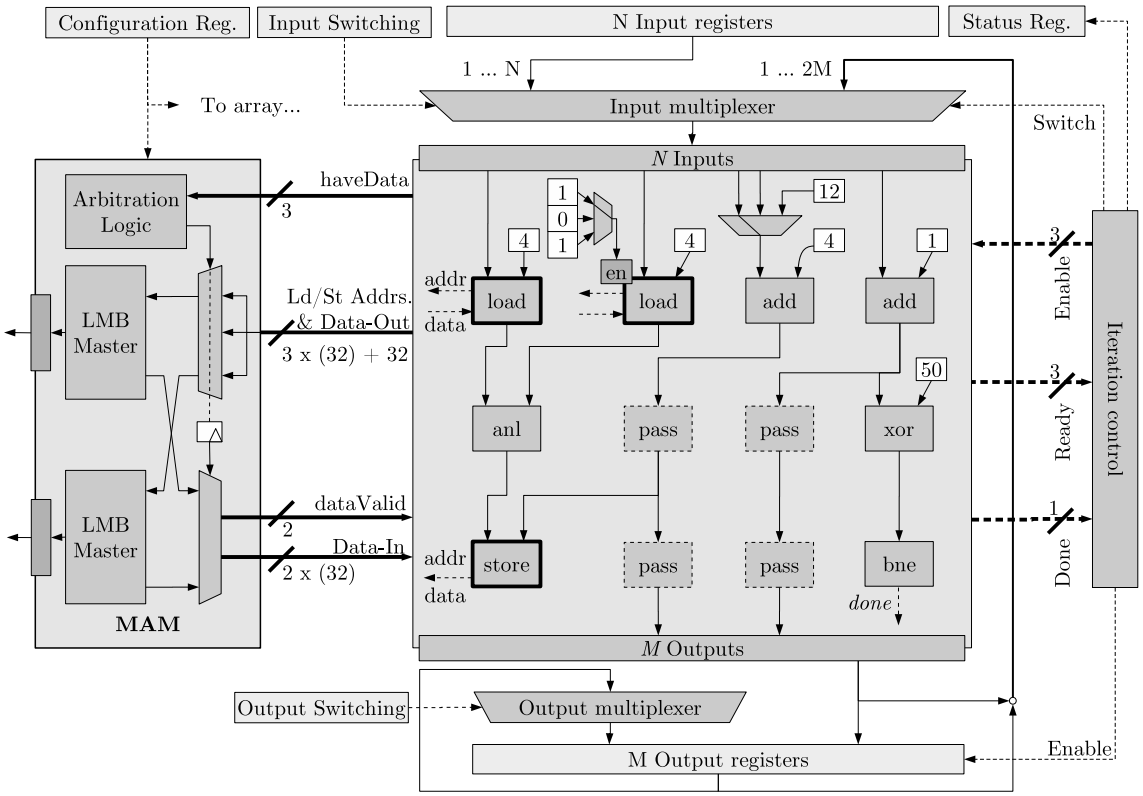


Figure 5.1: 2D Accelerator with memory access logic

5.1 Accelerator Architecture

Figure 5.1 shows the re-designed accelerator architecture. Much of the existing structure is re-utilized relative to Chapter 4: the accelerator template relies heavily on synthesis-time parameters and Verilog *generate* statements to create a specialized instance. As before, the customization varies the type and number of units in the 2D array, which determine its width and depth. Instead of relying on full crossbars between rows, this design also customizes the connectivity. Only the minimally required set of FU connections exists, so that the required data flow for all supported CDFGs can be implemented. As Section 5.3 will show, this implementation relies on the dual-port BRAMs present in the FPGA architecture. The Memory Access Manager (MAM) shown on the left-hand side is coupled to the array and contains LMB master logic to drive the buses connected to the data memory. It also contains control logic to arbitrate access of the *load* and *store* units on the array to the two ports. The accelerator is re-configured at runtime by the same method presented in the previous chapter: the *injector* detects the imminent execution of a translated trace by observing the instruction address bus. The transparent migration mechanism chooses one out of a set of configurations, which determines: which FUs are active, all multiplexer selections, and, in this architecture, also configures the arbitration logic. This section explains the structure of the accelerator, the support for memory access, and the execution model.

5.1.1 Structure of Functional Unit Array

The accelerator contains a 2D array of FUs, connected by runtime configurable multiplexers, tailored for a set of CDFGs. There is no limitation on the number of any type of FUs that can be instantiated, including *load* and *store* FUs, each of which is a separate type of unit. Also, integer division by a constant is now supported, via a reciprocal multiplication method [War02]. All other integer 32-bit arithmetic, logic operations, comparisons, and *exit* operations are still supported.

All FUs still execute in a single-cycle, except for the *load*, *store* and the integer division. The *load* units have a latency of two clock cycles, which may increase due to access contention to the two memory ports. The *store* units add no latency to the execution. The data to write to memory is buffered if no ports are immediately available. However, idle cycles may still be introduced if the buffering capacity is exceeded. This is further explained in Section 5.2.2. The division FU has a constant latency of 2 clock cycles.

The operands are received into the set of *input registers* as the MicroBlaze executes a CR. The *output registers* are driven only by the last row and are read back at the end of execution. Data is registered at each FU output. These registers then feed the inputs of the FUs in the following row and, unlike the previous design, may also be directly connected to more distant, non-adjacent, rows. *Passthroughs* are instantiated conditionally based on a manually adjustable parameter that either: instantiates *passthroughs* in every row, instantiates no *passthroughs* at all, or instantiates *passthroughs* only at specified rows. The option to completely remove *passthroughs* was introduced in order to reduce the number of registers required for large accelerators. The capability to still introduce *passthroughs* in certain rows was kept to avoid long connections which lead to critical paths. Removing some, or all, such units has no implications on execution, because rows do not activate before their previous results have been consumed.

In the previous design, every FU input was driven by an *M-to-1* multiplexer where *M* was the number of outputs of the previous row. For this design, each multiplexer is optimized to fetch only the required outputs throughout all supported CDFGs. In order to do this, the translation tools output an additional set of parameters which specify which FU outputs drive each multiplexer. Section 5.2.3 (page 85) contains an excerpt of this HDL specification. The *generate* directives in the HDL template instantiate a dedicated multiplexer per input for all FUs. The multiplexer is optimized away for cases where the connectivity is *1-to-1*. Also, it is still possible to fed the FU inputs with synthesis-time constants. This implementation requires significantly less resources relative to the crossbar implementation in the previous chapter.

A configuration of the accelerator corresponds to one global multiplexer context, which is set at the start of execution. Fig. 5.1 shows this for the add FU in the first row, which for two configurations is fed by values retrieved from the feedback lines, and which is fed by a constant value for a third. Similarly, each configuration also enables or disables each FU using small single-bit multiplexers, as shown for the second *load* FU in the first row. This is only relevant for exit FUs so that control progresses correctly, and for the store FUs, so no erroneous accesses are issued.

The interface with the processor remains unchanged relative to the previous design. The exception is that only one configuration register is required to set the connection context. These architecture modifications do no imply a change in the migration mechanism or the CRs.

5.1.2 Memory Access Support

The MAM shown in Fig. 5.1 is responsible for granting the *load* and *store* units in the array access to either of the two existing memory ports. The translation process shown in Section 5.2 does not take the MAM into account when generating the array. It simply instantiates as many *load* and *store* units required, as with any other type of unit.

The MAM receives all signals from the *load* and *store* units, including addresses, data-out and data-in signals, along with byte-enables and *haveData* control signals, which indicate outstanding data to be read or written. In Fig. 5.1, there are three units, resulting in a *haveData* bus of 3 bits. The MAM has static synthesis-time knowledge about which type of access each bit corresponds to. When enabled, a *load* or *store* unit asserts an access request and waits for a reply. A row containing *loads* only finishes executing once all such units complete their access. *Store* units may not cause the same effect as they may buffer up to one datum. This means that simultaneous requests to the MAM may originate from different rows due to these outstanding stores.

There are two supported arbitration methods: 1) runtime selection logic which assigns a port to existing access requests based on the topological order of the units, or 2) the translation tools create a static access ordering which aims to minimize access latency. In the first case, there is a round-robin type logic which sequentially allows access to the ports to each unit. There is an optional parameter to determine whether both ports handle all units and make mutually exclusive choices at runtime, or whether each port handles only half of the existing units on the array. For the implementation presented in this chapter, local data memory was used. Since the latency of the BRAMs is constant at one clock cycle, this leads to an eventual steady state in terms of which units access the bus at which time. The static access scheduling reproduces this by deterministically assigning two units to the two ports per cycle, thus removing the runtime arbitration logic. Since this is done by the translation tools, it is explained in detail in Section 5.2.2.

The Megablock instructions which compute the runtime addresses for memory access are implemented in the array as well. That is, no dedicated address generation logic is used, as this process is itself part of the accelerated trace. This means that arbitrary memory accesses are supported by the accelerator. The accelerator has no capability to issue exceptions when an access is attempted at a non-valid location (e.g., outside the existing memory range, or reads/writes into code memory). This is not expected to occur however, since addresses are either specified as synthesis time parameters which originate from compile time constants in the trace instructions, or are computed by the MicroBlaze itself prior to accelerator invocation. Regardless, behaviour after such an event is undefined, although falling back to software after an access exception (e.g., address out of bounds) by the accelerator would be a trivial feature to implement.

In order to support memory access at a system level, an additional type of module is required: Fig. 5.2 shows the LMB multiplexer. It drives a single BRAM port by selecting between the

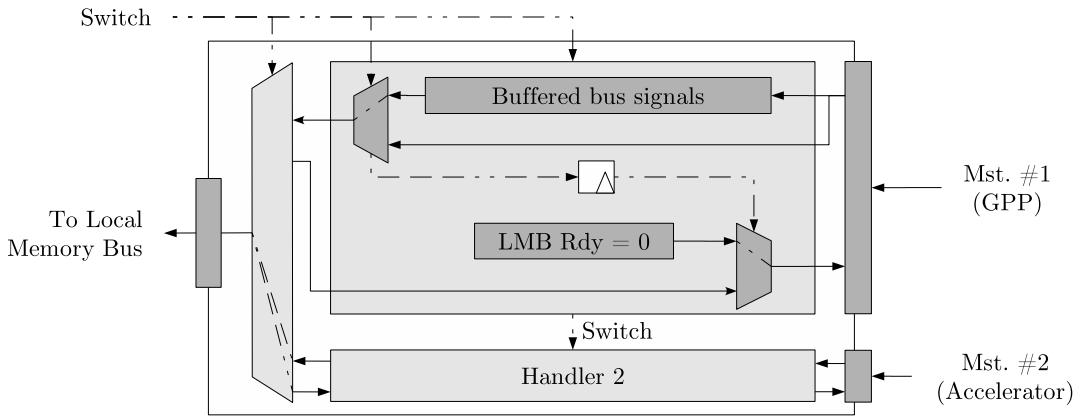


Figure 5.2: Local Memory Bus Multiplexer module

signals originating from the MicroBlaze or the signals originating from one of the accelerator's memory ports. The multiplexer routes memory replies back into the selected master. Prior to execution, the accelerator controls the switching signal, which causes the multiplexer to buffer up to one MicroBlaze access and reply with a *wait* signal. The LMB bus does not issue a time-out, so the MicroBlaze waits indefinitely. Also, the LMB multiplexer does not add any signals to the bus or latency to the bus transactions. When the *switch* signal changes, the downstream (e.g., master to bus) signals are immediately connected to the memory. The upstream signals are connected in the following clock cycle, so that the response to the previous access is directed to the correct master.

Sharing the data memory in this fashion is only possible through use of local BRAMs. Despite this, some higher end FPGA devices still contain a considerable amount of memory (e.g., Xilinx's Virtex UltraScale family [Xil15a], with upwards of 25 Mbit) which may be enough for some applications. The main advantage of this architectural design is that the entire data memory range is shared without any additional data transfer overheads. This would be the case if, for instance, the accelerator contained its own local memories. Also, by taking advantage of the readily available dual-port capability, its possible to straightforwardly exploit data parallelism by using two LMB multiplexer modules. Section 5.3 shows this experimental setup. The LMB multiplexer was designed prior to the release of versions of Xilinx's LMB which support multiple masters. Using those modules for BRAM sharing would result in the same functionality.

5.1.3 Execution Model

Execution on the array begins a soon as the expected number of operands is received. The number of operands required varies per selected configuration, which is set by the *injector* after detection of the trace start address. All FUs in one row activate concurrently when that row's *enable* signal is set (and also according to each FU's configuration dependent enable).

In this implementation, each row may take more than one clock cycle to execute for one of two reasons: there is a multi-cycle FUs in the row (i.e., a load or a division) or stalls occur due to memory access contention. To support multi-cycle rows, the control logic now relies on handshaking

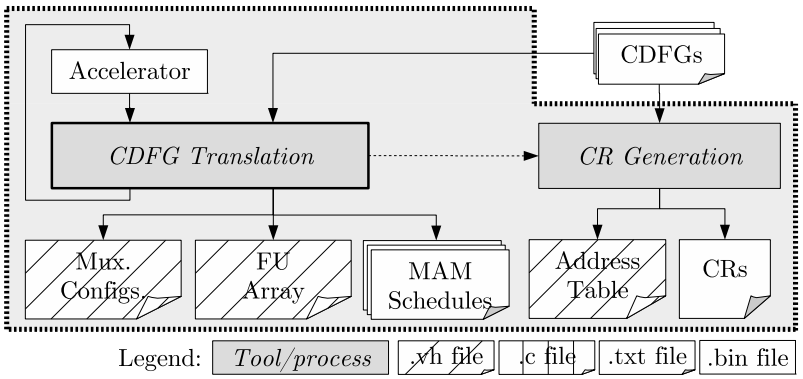


Figure 5.3: Architecture-specific flow for this 2D accelerator design

signals. A row asserts a *done* signal once every FU on the row issues its own completion signal. Only then is the *enable* signal of the following row asserted. This way, the control of the array is separated from its per-instance structure, and especially from the memory access behaviour.

After the first iteration, the *iteration control* module controls the *input multiplexer* so that data feedback is established. Like the previous design, the feedback of data is performed only from the last row to the first, and it is possible to feed back results from the newly completed iteration as well as the immediately previous one. Unlike the previous design, the possibility of connecting distant rows without intermediate *passthroughs* means that configurations whose CPL is smaller than the number of rows of the array can execute without additional penalty. That is, not all rows need to activate in order to complete an iteration. By implementing the Megablock's branch operations in the array, the accelerator is capable of executing an arbitrary number of iterations determined by live input data. An iteration either executes completely or, if any exit condition triggers, it is discarded. The active configuration also determines which exit FUs are enabled. The results of the latest completed iteration are sent back to the MicroBlaze, which resumes execution at the start address of the Megablock trace, executing the last loop trace iteration in software.

5.2 Accelerator Generation and Loop Translation

Fig. 5.3 shows the architecture-specific tool flow for this implementation of the accelerator. The translation tool now generates two main additional files specifying each multiplexer's connectivity as well as a configuration file for the MAM. As per the general approach, the CDFGs are produced by a simulated execution stage on the Megablock extractor. Most of the description of the CDFG translation tool given in Chapter 4 applies here. The notable differences are the fact that list scheduling is now performed when assigning CDFG nodes to a position on the array and that, after the translation step which assumes unlimited connectivity, the multiplexers are specified.

5.2.1 List Scheduling

For each run of the translation tool, one CDFG is processed. After N runs, the generated accelerator supports execution of the N used CDFGs, by having N configuration contexts. After being parsed, the CDFG is processed to determine initial values for each node's earliest and latest possible placement, based on their connecting edges. An additional placement condition is set for *store* nodes: they cannot execute before all *exit* operations execute. This is to ensure that the last iteration on the accelerator. which is discarded, does not write any data to memory.

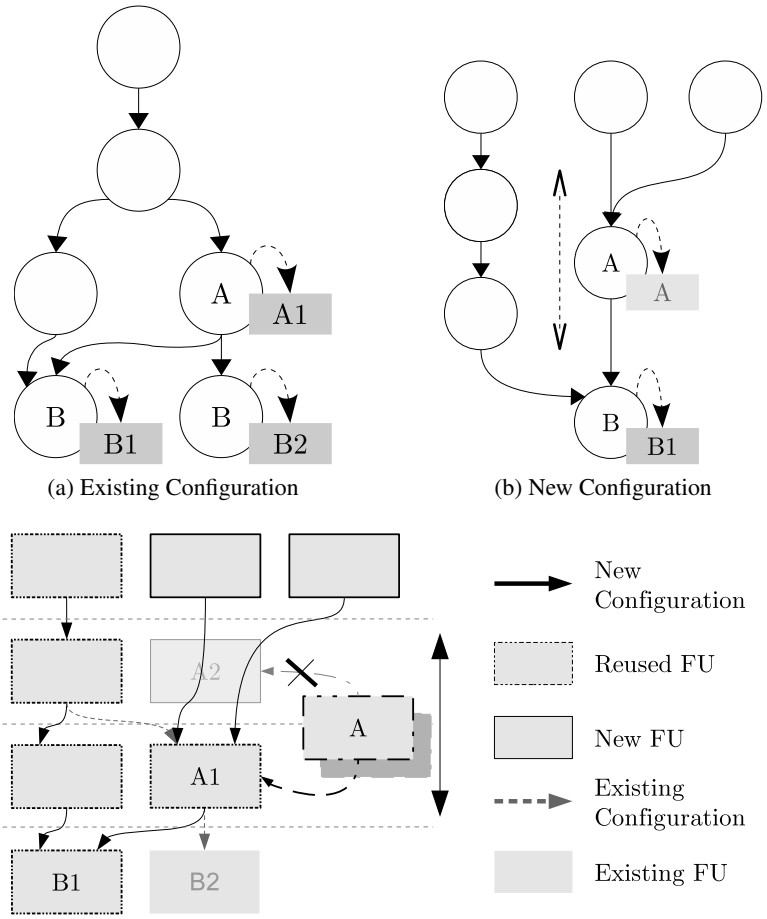
As before, translating a CDFG involves assigning nodes to a 2D grid. Consider an already existing grid array, due to a previous translation. First, the grid size is adjusted for the new CDFG if necessary, then each node is then placed individually in topological order. When placing a node, its placement range is re-computed to take into account the location of other nodes, from the same CDFG, which are already placed. Let l_e to l_l be the earliest and latest row. The rows are searched, starting from the l_e, for an existing FUs which can execute the node's operation. If one exists then the node is assigned to it. Otherwise a new FU is placed at the leftmost free position in the earliest possible row, l_e. The placed node is annotated with a value, l_r, indicating the chosen row.

The values l_e and l_l are both computed iteratively, prior to placement of each node, by traversing the CDFG, starting with the node to place. Algorithm 1 shows the pseudo-code for the calculation l_e. The value for l_e is first set to zero. Incoming edges (i.e., inputs) are then followed, starting from the node being placed, until an already placed node is found.

Algorithm 1 Iterative calculation of earliest possible node placement

```
1:  function CALCASAP(graph, node)
2:      l_e ← 0                                              ▷ Initial value
3:      if node is placed then
4:          return node.l_r                                  ▷ node is locked to its assigned position
5:      end if
6:      for every input of node do
7:          l_e_source ← CALCASAP(graph,input.sourceNode)    ▷ Compute asap for source node
8:          if l_e_source + 1 > l_e then
9:              l_e ← l_e_source + 1                         ▷ l_e is computed based on already placed nodes
10:         end if
11:     end for
12:     return l_e
13: end function
```

To avoid traversing the graph circularly, input edges which originate from nodes in a topological level further down (i.e., backwards edges) are not followed. Nodes are placed from top to bottom, so a placed node which defines the earliest possible placement of the following nodes is usually found. A similar process is applied to determine l_l by following outgoing edges. Backwards outgoing edges are ignored during this. An already placed node is sometimes found during this search, which constrains l_l. Since this is not usually the case, the value for l_l is bound by setting it to maximum of the CPL of the graph and the number of existing rows in the array.



(a) Existing Configuration

(b) New Configuration

(c) Resulting FUs

Figure 5.4: List scheduling example for a Functional Unit with available slack

Figure 5.4 shows: a CDFG that has already been translated (rectangular tags indicate the associated FU); a new CDFG to translate; and a simple representation of the array of FUs. The CDFG in Fig. 5.4a generated the FUs A1, B1 and B2, as well as the three additional unlabelled FUs in the leftmost column of Fig. 5.4c. When translating the CDFG in Fig. 5.4b, these three FUs are re-utilized, and two new FUs are added to the first row. Node *A* can be placed in either row 2 or row 3. The later row is picked to re-utilize the existing FU, *A1*, instead of instantiating a new FU in row 2. Node *B* is assigned to FU *B1*. The *B2* FU remains idle in this second configuration.

The *load* and *store* nodes are placed with some awareness of their effects on execution. The placement tries to minimize the number of *load* and *store* FUs in a single row to decrease the execution latency from port access contention. To do this each row is given a placement cost when placing a new *load* or *store* FU, based on how many FUs of the same type already exist on that row. This process is essentially a heuristic which attempts to reduce the total amount of idle time introduced due to memory access, based on the execution model and memory access limitations. It is independent of the arbitration method the accelerator uses during execution.

Placement of a new *load* unit considers only existing *load* units. This is a simplification adopted by relying on the ability to buffer *stores* for later cycles. Likewise, placement of a *store*

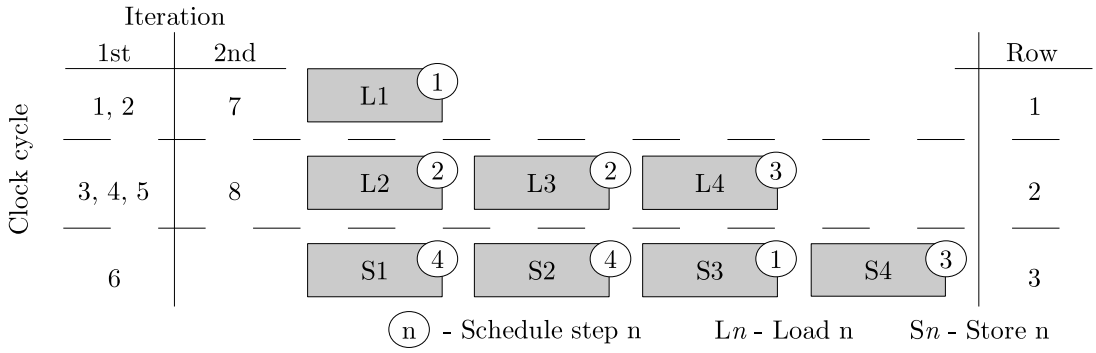


Figure 5.5: Assignment of *load/store* units to ports and cycles, after Functional Unit placement

unit considers only the existence of other *store* units. This naive strategy relies on the tendency for *store* units to be placed on lower rows of the array, which happens because: 1) they must not execute before any *exit* operations, and 2) usually *stores* operate on data retrieved by earlier *loads*.

Given this, consider that a row without memory operations completes in a single clock cycle. If a *load* operation is added, an additional cycle is required. So, placing a *load* in a row with no such FUs will incur a cost of 2. If a row already contains one *load* FU, then placing an additional *load* incurs no additional cost: the second access can be performed in parallel via the remaining free memory port. Placing a *load* in a row with two existing *loads*, only incurs a cost of 1, since the arbitration logic pipelines accesses to the memory. That is, a memory access strobe is issued while the datum from a previous access is being read. The former rationale holds for any odd number of *loads*, and the later for any even number. The same cost metric is used while placing *stores*. The scheduling of accesses is performed post-placement, as explained in the following subsection.

After placing all nodes, the same *passthrough* insertion step is performed: chains are inserted by analysing the array bottom to top, so that connections only span 1 row.

5.2.2 Memory Access Scheduling

As Section 5.1.2 presented, the MAM uses a runtime arbitration which selects operations topologically, or it contains static access schedules. The static schedules are generated during translation and rely on the *load/store* placement heuristic to achieve more efficient results.

The static schedule is held in distributed memory in the MAM, which is initialized at synthesis time with one of the translation tool's outputs. Each schedule step specifies which memory FUs are granted access too during each cycle. A single step may grant access to units from different rows, and the number of schedule steps need not be the same as the number of rows.

Figure 5.5 shows an example of assigning *load/store* units to execution cycles. Each row represents a row of the array (i.e., a total of three). Only *load* and *store* units are shown for clarity. The location of the units is the result of the placement process, which allocated all *stores* onto the last row, and was unable to pair off the four *load* units due to the available slack. On the left-hand

side, the values represent clock cycles. Each is associated to a corresponding row and iteration. Each unit is annotated with the clock cycle during which it is granted access.

To arrive at this schedule, the translation tool performs a manner of simple simulated execution by iterating through the rows post-placement. First, *loads* and *stores* are placed onto two separate topologically sorted lists. Per row, the tool first assigns as many unscheduled *load* units to available ports (which given the limitation is at most two). If any ports remain free, existing *stores* are assigned. The number of remaining free ports per step is tracked during this process.

The simulated execution then does one of two things: 1) advances the schedule step and remains on the same row if necessary (e.g., unscheduled *loads* prevent execution from continuing), or 2) advances to the following row, which means the step must also advance if any operation was scheduled during that cycle, even though a port may still be free. That port may be assigned to an outstanding store in a future pass through that row. There are two ways to determine when to advance to the next row: execution advances despite unscheduled *stores* to prioritize *loads* in the following rows, or the schedule ensures that the access order is preserved relative to the original trace (i.e., all units must be scheduled before execution advances). The former strategy permits advancing to the next row even if unscheduled *stores* exist. The later strategy is adopted to prevent violation of RAW dependencies. Advancing to the next row despite outstanding *stores* also allows for the scheduling of said *stores* during execution of a row without *load/store* units.

For the example in Fig. 5.5 this process occurs as follows: execution starts in row 1 and the schedule is at step 1; $L1$ is assigned to one port during this cycle, and one port remains free; execution advances to row 2, and the step must advance also; $L2$ and $L3$ are assigned to the ports, execution cannot continue yet due to $L4$; an additional cycle (i.e., step) is required; as a result this row requires 3 clock cycles to execute (cycles 3, 4 and 5); execution advances to row 3 and the schedule to step 4; $S1$ and $S2$ are assigned to the ports during this step; execution returns to row 1; the maximum number of steps in the schedule is now determined, since it must repeat from this point on; store S3 is assigned to step 1 (occurring in parallel with the second execution of $L1$); finally, $S4$ is assigned to step 3. All pending *stores* are scheduled before their units are activated again, so they introduce no additional cycles. This scheduling results in both memory ports being in use during every cycle. The total number of clock cycles required to complete an iteration with this access scheduling is 6. If a conservative schedule was used, then 7 clock cycles would be required, since the last row would not allow execution to continue.

For this particular example, if the runtime selection logic had been used, the number of clock cycles required would be equal. However, this would imply an additional resource cost and most likely a decrease in operating frequency. As was mentioned before, when using runtime selection logic each port may be fed only with half the access request signals on the array to reduce complexity and often the critical path. The disadvantage is that a sub-optimal result may be obtained since a single port may end up connected to a group of *loads* on the same row, for instance. Using static schedules simplifies the selection logic in the same manner without this issue.

5.2.3 Multiplexer Specification

The post-placement step which involved computing configuration register values is replaced by a similar HDL parameter generation step. Per row, each FU output is given a numerical identifier starting from zero. During synthesis, the *generate* loops instantiate multiplexers with one 32-bit output and a variable number of 32-bit inputs. The following is a synthetic example of the multiplexer specification file for the array in Fig. 5.1:

Listing 5.1: Excerpt from HDL specification of multiplexers for row 2 of the accelerator in Fig. 5.1

```
parameter [ 0 : (33 * 6 * NUM_CONFIGS) - 1 ] ROW2_CONFIGS = {
        {32'h0, 1'b0},   // load1   output 1 -> anl   input 1
        {32'h1, 1'b0},   // load2   output 1 -> anl   input 2
        {32'h2, 1'b1},   // add1    output 1 -> pass  input 1
        {32'h3, 1'b0},   // add2    output 1 -> pass  input 1
        {32'h3, 1'b0},   // add2    output 1 -> xor   input 1
        {32'h50, 1'b1}   // constant, "0x50" -> xor   input 2
};
```

The row in question has 4 FUs and a total of 6 inputs. Each line of the parameter shown represents one input multiplexer. Each multiplexer input is specified by two fields. A single bit field which determines if the associated 32-bit value is a constant, or a numerical reference to the outputs of row 1. For this example, there is only one configuration, so each row shown (i.e., each multiplexer specification) contains only one 33-bit parameter.

5.3 Experimental Evaluation

The experimental evaluation presented throughout this section aimed to validate the accelerator capability for memory access. As such, benchmarks which contained inner loops operating on data arrays were selected from several benchmarks suites. This section contains comments on the general characteristics of the generated accelerators (Section 5.3.3), an evaluation of accelerator-enabled execution versus a single MicroBlaze processor (Section 5.3.4), determines how advantageous the list scheduling and memory scheduling efforts are (Section 5.3.5), reports on the resource cost and operating frequency of the accelerators (Section 5.3.7), and finally presents a short analysis on power and energy consumption (Section 5.3.8). Following are the hardware and software setups for these experiments.

5.3.1 Hardware Setup

Figure 5.6 shows the system architecture used to validate the accelerator's capacity for memory access. The program code and data of the benchmarks reside in local BRAM memories. Two LMB multiplexers are used drive its ports. Each multiplexer receives signals from one of the accelerator's ports, and one of the MicroBlaze's LMB ports, of which there are two (one for data accesses and another code instruction accesses). The *injector* is coupled to the instruction bus before the multiplexer. Both modules are transparent to the MicroBlaze, and to each other. The accelerator and MicroBlaze communicate via FSL.

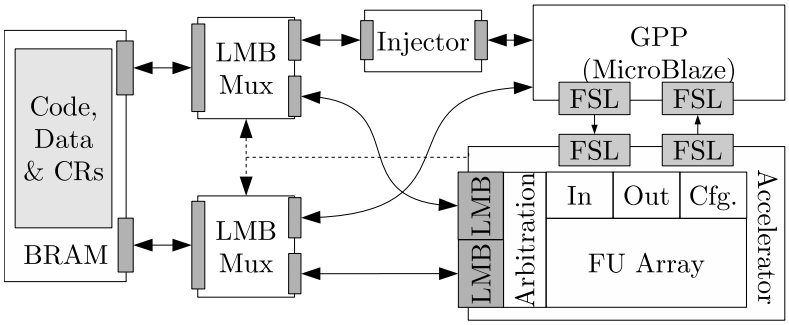


Figure 5.6: System architecture for validation of accelerator local memory access

The system was implemented on a Digilent Atlys board, with a Spartan-6 LX45 FPGA. The frequency of the system clock was of 66 MHz for most cases. The MicroBlaze version used was *8.00.a*, and it was set for *Performance*. The integer multiplication, pattern comparison and barrel-shifter units were enabled. For two benchmarks, the integer division unit was also used.

The implementations in Chapter 4 relied only on explicit calls to vendor timer modules to measure execution time. This system employs custom timer modules attached to the PLB (not shown) to retrieve several measurements. The timers are controlled by the *injector*, accelerator and MicroBlaze. Using the *injector* the execution time of the accelerated regions of code only (as opposed to the entire application) can be measured. The short migration overhead time is also measured. The accelerator controls a timer to measure execution on the array, and a timer to measure idle time due to memory access. To measure the total execution time, the benchmarks are minimally modified so that the first and last lines start and read a global timer which measures the execution time of the entire application.

5.3.2 Software Setup

A total of 37 benchmarks were used to test the system. The benchmarks originate from Texas's IMGLIB library [Tex], the SNU-RT [Seo], WCET [GBEL10], and PowerStone [MMC00] benchmark suites and from other sources [Int96, War02, Jen97]. Benchmarks used were ones which resulted in Megablock traces containing *load* and/or *store* instructions. The benchmarks are separated into three sets: (*A*) benchmarks whose source-code has been manually *if-converted* to remove short *if-else* statements; (*B*) simple benchmarks with a single candidate Megablock resulting from an inner loop with no *if-else* statements; (*C*) larger benchmarks from which several Megablocks were extracted and accelerated. Xilinx EDK 12.3 was used for RTL synthesis and bitstream generation. Benchmarks were compiled with *mb-gcc 4.1.2* using the *-O2* flag.

Regarding set *A*, note that the rationale underlying the proposed approach of binary acceleration still relies on not modifying the source code. This was done for validation purposes. The code was minimally modified via *if-conversion* [AKPW83]. This is a technique which converts high-level control structures into arithmetic statements. Using this modification, its possible to extract a single Megablock trace from the loop, as opposed to several traces starting at the same address.

As a reminder: the present migration mechanism cannot disambiguate between two traces starting at the same address, and neither translation nor execution of multiple-path traces are supported.

The benchmarks used in the experiments in Chapter 4 were simple calls to an inner loop function; all benchmarks were structurally very similar in terms of *C* code. The benchmarks for sets *A* and *B* in this evaluation are similar to those. The benchmarks for set *C* are relatively more complex. For instance, the *pocsag* benchmark from the PowerStone suite is a single file with 500 lines of code. The code is structured such that a complex call graph results from the 8 integer functions, which operate on several global data arrays.

5.3.3 General Aspects

Table 5.1 shows the characteristics of the detected Megablock traces and of the resulting accelerators. The table is split into three sections, according to the benchmarks sets. The means shown at the bottom row of each section are arithmetic, and the *total mean* row accounts for all benchmarks. The third, fourth and fifth columns show the number of accelerated traces, average number of trace instructions, and average executed iterations per trace occurrence. These averages account for all the accelerated traces of each respective benchmark, and are weighted based on the execution frequency of each trace. In total, 73 traces were targeted for acceleration.

The average number of instructions for the accelerated traces of this set is much higher than for the implementation shown in Chapter 4. The average number of instructions in the later case was of 9.5, whereas the average for this evaluation is 40.5. For all the accelerated Megablocks, an average of 13 % of all trace instructions are *loads*, and an average of 9 % are *stores*. That is, there are on average twice as many loads as stores, which is to be expected, because typically several input data produce one output datum. Also, a *load* or *store* instruction actually corresponds to two accelerator operations: the memory access is separated into an addition which generates the address, and the actual access.

The size of the traces increases for two reasons. Firstly, and mostly, inner loop code without memory handling operations is unrealistically small. The benchmarks for this evaluation contain inner loops which are larger simply because they implement more realistic processes handling large amounts of data residing in memory (e.g., two inner loops in *b17* with approximately 200 lines of code each). Secondly, the support for *load* and *store* instructions allows for a larger execution scope to be targeted. For example, the benchmarks in set *B* are (for the most part), simple code kernels called within a loop construct, as is the case with *a6*:

```
1 || int crc32(int data) {
2 ||
3 ||     //(...)
4 ||     for(int i=0; i < 8; i++) {
5 ||         //(...)
6 ||     }
7 ||     return ans;
8 || }
9 ||
10 ||    //(...)
11 ||    for(int i = 0; i < N; i++)        // the accelerated trace for a6
12 ||        output[i] = crc32(input[i]);  // begins at this loop boundary
13 ||    //(...)
```

This is identical to the test cases in Chapter 4: a inner loop kernel called a predictable number of times, where each call corresponded to a small number of iterations operating over a single set of data. Without memory support, the read and write operations present in the outer (calling) loop at line 12 could not be accelerated. The trace for the innermost loop would be targeted, and the accelerator would be called a total of N times (in this case). Instead, since the nested loop is small enough to be unrolled, the Megablock extractor provides a trace capturing the outer loop code as well. As a result, the trace is larger and the accelerator is only called once. This single call performs N iterations of the outer loop, corresponding to the total $N \times 8$ iterations of the inner loop. This is also an example of how the Megablock trace detection crosses function call boundaries.

In short, there is a relationship between the scope of code viable for acceleration, the number of trace instructions, number of iterations and accelerator calls. These last two aspects are especially correlated. For the 37 benchmarks in this evaluation, the number of accelerator calls is sometimes data dependant and in some cases the accelerator is only called once (for 7 cases of set A, 9 cases of set B and 2 cases of set C). The average for the entire benchmark execution is of 475 calls. The average number of iterations per call is 523.

Supporting larger CDFG also results in larger accelerators. Given the translation method, the array scales in a linear fashion with the CDFG size. The sixth column shows the number of FUs in the array, excluding *passthroughs* and including *load* and *store* units. The number of instantiated units of these types is shown in the following two columns. Relative to the implementation in Chapter 4, the average accelerator contains 5.8× more FUs (excluding *passthroughs*), 8.2× more *passthroughs* and 3.2× more rows. Section 5.3.7 discusses the effect this has on required FPGA resources. There are 2.66 *passthroughs* per every other type of FU. Similar to the number of memory operations in each trace, the average number of *load* units corresponds to 11 % of all FUs (excluding *passthroughs*), and the average number of *store* units to 6 %.

Given that the accelerators support multiple loops, not all instantiated FUs are active for all configurations. Consider then only the cases where the accelerator supports multiple loops. For each accelerator, the weighted average number of active FUs is computed based on the number of iterations performed per configuration. These per-accelerator values in turn result in an average of 28.3 FUs being used per configuration out of a total of 48.3. Likewise, for an average total of 5.1 and 2.6 *load* and *store* units on each array, respectively, 3.2 and 1.7 are activated per configuration.

Also, within a single configuration, not all FUs are active every clock cycle, given the execution model and memory access latency. When considering all implemented traces, the average number of idle FUs per clock cycle is of 28.3 (arithmetic average for all configurations). We can also compute how many FUs are idle per clock cycle per accelerator instead: 37.6. This is a non-weighed average, for all accelerators, of the individual weighted averages of each accelerator's number of idle FUs (according to the execution frequency of each supported configuration).

By analysing the resulting accelerators, we also find that the average number of rows is 13.6. The accelerators have two access ports and, for this implementation, an access latency of 1 clock cycle. Given this, an estimation can be made indicating that: the $3.6 + 2.1$ memory operations can be completed in $(5.7/2 + 1) = 3.85$ clock cycles. Since $13.3 > 3.85$ this seems to indicate

Table 5.1: Extracted Megablock and generated accelerator characteristics

id	kernel	# Cfgs.	Avg # Insts.	Avg # Iter.	# FUs	# Passes	#ld/st	#rows	HW IPC
a1	boundary	1	25.0	68.0	24	45	3/2	6	3.13
a2	bubble_sort	1	19.0	62.0	21	71	2/2	10	1.58
a3	changebright	1	20.0	99.0	20	94	1/1	12	1.54
a4	compositing	1	24.0	199.0	24	112	2/1	15	1.50
a5	conv_3x3	1	81.0	149.0	77	233	18/1	19	2.89
a6	crc32	1	131.0	999.0	80	237	1/1	50	2.57
a7	mad_16x16	1	15.0	15.0	16	47	2/0	9	1.36
a8	max	1	13.0	2047.0	14	38	1/0	10	1.18
a9	millerRabin16	1	59.0	8.4	52	52	6/6	6	5.90
a10	perimeter	1	24.0	479.0	28	86	5/1	10	1.85
a11	pix_sat	1	19.0	1999.0	21	81	1/1	14	1.27
a12	rng	1	84.0	499.0	74	154	6/6	18	3.65
a13	sobel	1	52.0	957.0	56	223	8/1	20	2.08
	mean	1	43.5	583.1	39.0	113.3	4.3/1.8	15.3	2.35
b1	blit	2	11.0	999.0	13	26	1/2	4	2.20
b2	bobhash	1	10.0	999.0	11	24	1/0	8	1.11
b3	checkbits	1	63.0	499.0	64	169	2/2	16	3.71
b4	checksum	1	149.0	124.0	155	482	8/0	52	2.66
b5	fft	2	28.9	28.9	64	126	10/8	10	2.73
b6	gouraud	1	17.0	1999.0	15	37	0/1	6	2.83
b7	lookup2	1	46.0	999.0	48	105	3/0	22	1.92
b8	motionEst	1	13.0	15.0	13	48	2/1	7	1.63
b9	perlins	1	123.0	1023.0	123	575	4/1	29	3.97
b10	popArray1	1	27.0	100.0	22	94	1/0	15	1.69
b11	popArray2	1	56.0	32.0	46	200	3/0	20	2.55
b12	quantize	1	11.0	199.0	11	35	1/1	6	1.57
b13	sad16x16	1	12.0	15.0	14	39	2/0	8	1.33
b14	ycdemuxbe16	1	14.0	999.0	20	18	4/4	3	2.80
b15	bcnt	1	72.0	15.0	98	155	25/1	13	2.77
b16	Menotti_fdct	2	103.5	53.5	136	291	8/15	17	4.93
b17	jfdctint	1	78.0	7.0	95	171	9/8	11	4.88
b18	fir	1	9.0	16.0	10	26	2/0	5	1.29
	mean	1.1	46.9	451.2	53.2	145.6	4.8/2.4	14.0	2.59
c1	pstn_adpcm	7	9.5	6.9	31	40	8/6	6	1.43
c2	pocsag	2	45.4	20.5	67	121	12/0	11	2.84
c3	wcet_adpcm	13	4.3	1386.1	38	50	8/6	7	1.33
c4	edn	6	12.7	51.5	50	79	7/4	9	1.58
c5	jpeg	8	13.0	1320.8	117	193	10/11	14	1.48
c6	g3fax	3	5.9	858.4	18	33	2/1	6	1.26
	mean	4.8	15.1	607.4	53.5	86.0	7.8/4.7	8.8	1.65
	total mean	1.97	40.5	522.9	48.3	124.6	5.1/2.6	13.6	2.35

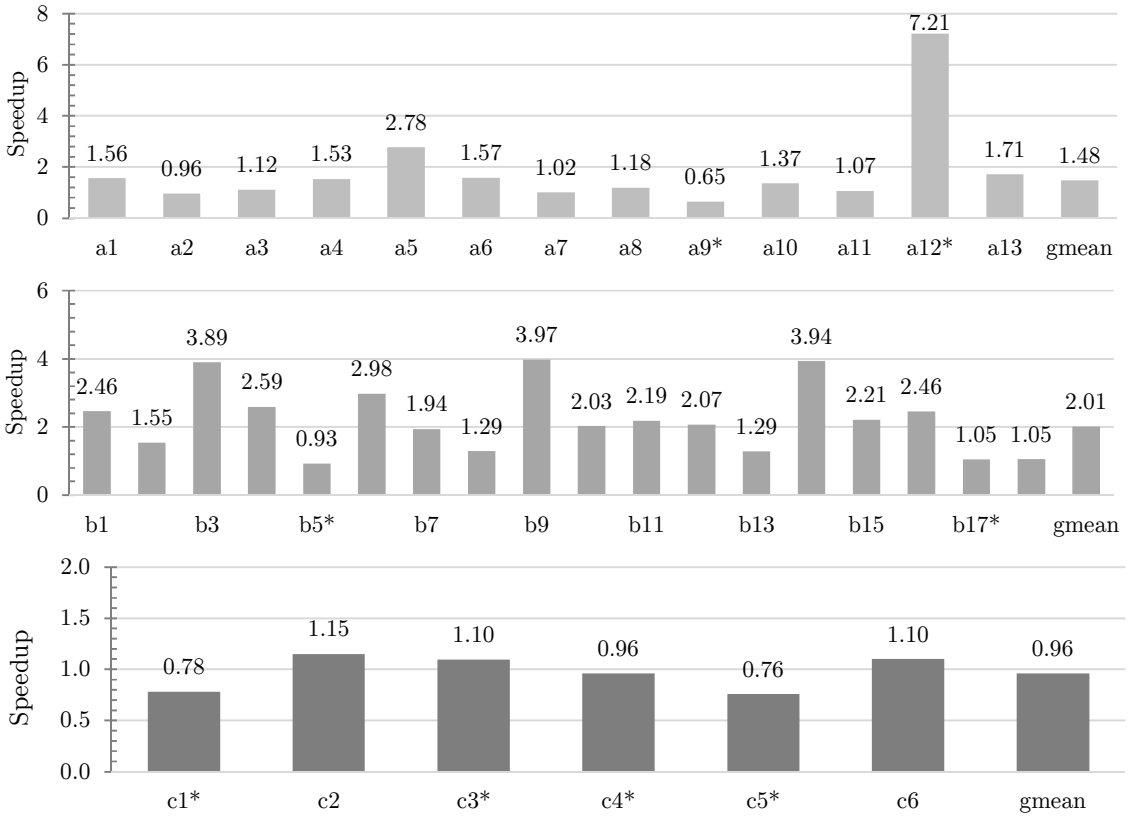


Figure 5.7: Speedups for the three benchmark sets (benchmarks for which the execution frequency was below 66 MHz are denoted with $*$)

that two ports are, in general, enough so memory access latency does not nullify the performance gains derived from ILP. Of course this varies on a per-case basis given each CDFG. However, in general, two low latency ports seem adequate, given the ratio of number of memory accesses to number of arithmetic operations. Note that this analysis only holds for non-pipelined execution.

5.3.4 Performance vs. MicroBlaze Processor

The seventh column of Table 5.1 shows the number of executed Instructions per Clock Cycle (IPC) on the accelerator, IPC_{HW}. It is computed as the number of accelerated trace instructions over the number of clock cycles required to complete an iteration. Unlike the previous chapter, this value is not the same for all the supported configurations of an accelerator. Firstly, not all rows may need to activate; for each supported configuration, only as many rows as the respective CDFG's CPL need to activate (at best). Secondly, there is the memory access latency. Given this, the IPC_{HW} shown for each benchmark is equal to a weighted average of these per-configuration IPC_{HW} values.

Likewise, the number of instructions the MicroBlaze executes per clock cycle, IPC_{SW}, can be computed. This is omitted from Table 5.1, since its value, 0.85, is nearly identical for all cases. The only exception is *a12*, which contains 4 integer division instructions, each of which requires 32 clock cycles to execute, resulting in an IPC_{SW} of 0.40.

Figure 5.7 shows the measured speedups for all the sets. These speedups account for the execution of the entire application, not just the accelerated regions. Considering all benchmarks, the accelerated traces correspond to an average of 89 % of the benchmark's total execution time. Sets *A* and *B* skew this average since, being simpler kernel function calls, the accelerated trace represents most of the benchmark. For set *C* alone, the accelerated regions represent 71 % of the application. Geometric mean speedups are shown for each set, and the overall geometric mean is of 1.60× For some cases, the system frequency was below the 66 MHz baseline due to the accelerator. These cases are marked with an asterisk (*) in Fig. 5.7. The calculation of speedup takes the different clock frequencies into account.

IPC, Overhead, and Speedup The relationship between IPC_{HW} and speedup is the same as presented in the previous chapter (as shown in Eq. (4.1), page 71); the speedup increases with the accelerator IPC. The IPC_{HW} is higher if the number of CDFG instructions increases, and its CPL decreases (note that this is valid for non-pipelined execution). The effect of memory access support is accounted for in the IPC_{HW}, as additional clock cycles are required to complete an iteration relative to the accelerator's number of rows alone. However, the overhead (due to accelerator-MicroBlaze communication, exacerbated by low iteration count accelerator calls) and how representative the accelerated traces are both weigh on the final performance.

In general, a speedup is achieved for all benchmarks. Exceptions include *a2* and *a9*, for reasons explained shortly, and *b5*. For this later case, both of the accelerated traces are frequent, but for one of them the iteration call per count is very low. This negates any speedup, despite an IPC_{HW} of 2.60 and 3.46 for the two traces. For set *C*, half of the benchmarks experience a slowdown. This is due to the reduction in operating frequency relative to the MicroBlaze-only baseline, the reasons for which are later explained in Section 5.3.7. The accelerated traces are less representative than the ones for sets *A* and *B*, and fewer iterations are performed per call, increasing the overhead. The average overhead suffered by sets *A*, *B* and *C*, respectively, is of 7 %, 5 % and 13 %.

As was stated, the accelerated traces represent only a portion of the executed code. The speedup of a benchmark is influenced by how much execution time each accelerated trace represents, but the accelerator performance is more evident when considering only the trace execution time (i.e., the kernel speedup). For all benchmarks, the geometric mean kernel speedup is of 1.95×. For sets *A* and *B* this value is 2.16×, and for set *C* it is 1.13×.

The previous chapter employed a set of 15 integer benchmarks, which is comparable to set *B*. The geometric mean speedup of the later case is 1.19× higher than that of the former. Similarly, the average IPC for the same set is 1.25× higher.

Effects of Memory Access Latency If memory accesses did not incur any additional latency, the average IPC throughout all accelerators would be 3.00, versus the achieved 2.35. Memory access operations add an average of 2.5 clock cycles to each loop iteration. The two extremes are *b15* and *b6*. In the former case, the 25 *load* operations add 13 clock cycles to the execution of 1 iteration (over the number of rows alone); for the later case, there is only 1 *store* operation, which does not

introduce any additional latency. According to the retrieved measurements, the array's stall time corresponds to 17.5 % of all accelerator execution time on average. Stall time accounts for any additional cycle (that is spent executing memory accesses) beyond the cycle which activates a row. For instance, a row with a single *load* represents one clock cycle of stall time.

The worst three cases are *a9*, *b14* and *b15*. Although these cases do not contain the most memory instructions per arithmetic operations, the resulting accelerators do contain the most *load* FUs per row (considering the average number of active units of this type per configuration, and the average number of activated rows per configuration). For *b15* especially, most of the execution time is spent accessing memory. Although the two ports allow for roughly twice the access bandwidth relative to the MicroBlaze (demonstrated by the respective speedup of 2.21×), this approach does not primarily target this type of workload.

Performance on *if-converted* Benchmarks Set *A* contains 13 benchmarks whose code was modified to via *if-conversion*. *If-conversion* typically increases the code size due to the additional arithmetic implementing what otherwise would be control statements. For example, consider the following loop with a conditional statement:

```
1 ||    for (i=0; i<n; i++) {
2 ||        if (v[i] > max) {
3 ||            max = v[i];
4 ||        }
5 ||    }
```

The following *if-converted* code is functionally equivalent, but there is only one loop path:

```
1 ||    for (i=0; i<n; i++) {
2 ||        condition = v[i];
3 ||        asm("cmp %0,%1,%2": "=r" (condition): "r" (max), "r" (condition)) ;
4 ||        condition = condition > 0;
5 ||        max = v[i]*condition | max*!condition;
6 ||    }
```

Execution of the converted code is less efficient for software-only execution. Due to this, the speedup for these cases is measured versus the original, non-converted code. On the other hand, the IPC_{HW} is computed using the number of instructions in the accelerated trace (extracted from the *if-converted* code), since it is difficult to determine the number of instructions being accelerated relative to the non-converted code, due to multiple execution paths and differing binaries.

The *if-conversion* leads to the only slowdowns for all benchmarks: *a2* and *a9*. When comparing accelerator execution to the software execution of the *if-converted* code, the accelerator achieves speedups of 1.80× and 0.79×, respectively. The later case still experiences a slowdown due to the decreased clock frequency of the accelerator-augmented system. Even when comparing the number of clock cycles alone, this case still suffers a slowdown since each accelerator call executes only an average of 8.4 iterations.

Accelerating High Latency Instructions The speedup achieved for *a12* is the highest for all benchmarks. Despite the IPC_{HW} for this case not being the highest, the accelerated division instructions heavily penalize software execution. The four division instructions in the trace require a total of 128 clock cycles to execute on the MicroBlaze. On the accelerator, three of these

operations execute in parallel in only 2 cycles. Chapter 6 shows how acceleration of floating-point instructions achieves the same effect via low latency floating-point FUs which are also fully pipelined. The trace for this case also contained a total of 12 memory operations. The placement process minimizes the latency of the loads by scheduling most to the same row, pipelining their execution. An entire iteration of this large loop requires only 23 clock cycles.

This benchmark and *c3* are the only two where a division operation appears in the source code. Two conditions are required for traces with division operations to be candidates for acceleration: 1) the division unit must be enabled on the MicroBlaze, so that the operation is implemented with an explicit *div* instruction, instead of a *soft-div* implementation; 2) the divider must be constant. This does not happen for *c3*, since its division operations are contained in loops which compute a new divider per iteration. Regardless, the division unit was enabled so MicroBlaze execution was not penalized. Support for non-constant division is addressed in Chapter 6.

If a software subroutine is used to compute the division iteratively, it may still be possible to accelerate that routine. This depends greatly on how many execution paths the routine has, and how frequently the execution alternates between paths. The frequently occurring modulus operation subroutine displays this behaviour: it is very irregularly structured, contains a large number of branch operations and execution through its loop paths is very irregular. Although this routine is very frequent in some benchmarks, it is not a good candidate for acceleration. This is also an example of how high-level context is lost at the binary level: the trace extraction would be incapable of inferring that a division operation was being performed by simple observation of the *soft-div* routine. In contrast, a high-level synthesis approach would retain this information, regardless of the processor having, or not, a dedicated divider.

Regarding the constant divider, there are two ways the trace may contain this constant operator: 1) it is explicitly specified in the source code, or 2) the Megablock extraction process performs constant propagation. The former case is true for *rng*. The later case demonstrates the kind of runtime information that is possible with this type of binary acceleration approach. Consider that a kernel function containing a non-constant division is called from several locations in the code, with different sets of arguments per call. Although executing a non-constant division is not supported on the accelerator, it might be possible to arrive at a constant divider based on propagating the calling arguments. This type of contextual operator specialization is only possible using runtime data, and the concept extends beyond the present work. For instance, generating different hardware modules based on a single *fir* filter function, by knowing the different calling contexts (i.e., possible sets of coefficient values and/or filter size).

Performance with Multiple Traces The benchmarks for set *C* resulted in more than one candidate trace for acceleration. The traces for these benchmarks contain fewer instructions on average relative to the other sets. Table 5.1 shows the average number of trace instructions per benchmark in the fourth column. Note that this value, per benchmark, is an average of all accelerated traces, weighed by each translated trace's executed number of iterations. When considering the pool of all traces for this set (a total of 39), the average number of instructions is 16.1.

These benchmarks are more complex in terms of high-level control structures, are structured around sparse conditional calls of short functions which contain small loops with low iteration counts. For all cases, except *c2*, most extracted traces originate from such small loops. The rationale in evaluating the approach with these benchmarks was that the retrieval of binary level information would expose large frequent paths which would otherwise not be obvious via high-level analysis of the application. The average number of trace instructions for these cases alone (excluding *c2* and two large traces for *c5*, for reasons discussed shortly) is only of 9.3. In general, the benchmarks in this set are too control oriented for large, very frequent, traces to be detected.

The two exceptions are *c2*, and two traces for *c5*. For *c2*, there are two detected traces, which are paths through a *while* loop containing several conditional statements and inlined functions. It was possible to simultaneously translate both Megablocks since each starts at a different address. For *c5*, a total of eight traces were accelerated. Unlike other cases for this set, whose several traces were of similar sizes, two of the traces for *c5* both contain 106 instructions. The average number of instructions for the remaining six traces is 7.8. Although execution of the smaller traces on the resulting accelerator is not penalized due to supporting the two larger traces, the resources required to execute the latter two are underutilized for nearly half the time. Considering all accelerated instructions for all eight traces (i.e., the product of the number of instructions and iterations performed, for each trace), the the two large traces represent 47.8 % of the execution.

Supporting only the two large traces would not affect the amount of time during which the required FUs are utilized, as they would still represent the same amount of computation relative to the entire benchmark. However, the resulting two-configuration accelerator would be less complex in terms of FU connectivity. The logic relative to the MAM would also be simpler, since less *load/store* units would be instantiated. For this accelerator design, supporting multiple traces must take into account their respective resource utilization, especially if the CDFGs are very distinct topologically. Even for a set of very similar CDFGs, the overall accelerator performance (in terms of frequency and increased resources) might be penalized by supporting a additional configuration which is infrequent relative to the rest. For a large set of traces, a more adequate design choice might be to deploy several accelerators, each targeting a subset of similar CDFGs. Benchmark *c5* is in fact one of the cases for which the accelerator complexity required lowering the system operating frequency. Details on accelerator operating frequency are presented in Section 5.3.7.

Potential Speedups and Other Comparisons As with the IPC_{HW}, we may also compute the optimal speedup for a system free of memory access latency. This can be done easily by taking the number of clock cycles spent on accelerator execution, and subtracting that count from the total. Since the number of iterations performed is known, an estimated execution time is added back using the ideal IPC_{HW} values. For all benchmarks, the geometric mean speedup becomes 1.82×. The three benchmarks which would benefit the most from a potential decrease in access latency are *a5*, *b14* and *b15*. Although *a9*, along with *b14* and *b15*, suffer the most memory access latency, as shown previously, the speedup does not increase significantly for *a9*. This is because the accelerated trace represents a small portion of the entire benchmark. Specifically, the

Table 5.2: Number of cycles required to complete an iteration and respective IPC, for greedy and optimization-enabled allocation

ID	Megablock	IPC (greedy)	CPIter (greedy)	IPC (optim.)	CPIter (optim.)
a5	1	2.79	29.0	2.89	28.0
b4	1	2.61	57.0	2.66	56.0
b5	2	3.21	14.0	3.46	13.0
b9	1	3.84	32.0	3.97	31.0
b14	1	2.00	7.0	2.80	5.0
b16	1	4.41	22.0	4.62	21.0
	2	4.86	22.0	5.10	21.0
c4	1	1.25	8.0	1.43	7.0
c5	2	5.58	19.0	5.89	18.0
	7	5.58	19.0	5.89	18.0
mean		3.61	22.9	3.87	21.8

time spent on the accelerator is 2.3 % of the total execution time. The accelerated kernel function corresponds to the deeper layer of a function call chain whose outer layer functions contain loops with modulus operations. As was previously explained, the built-in routine for calculation of the modulus is not viable for acceleration, despite representing most of the computation time for *a9*.

Regarding potential speedup, recall that some of the tested cases suffered from decreased operating frequency due to the accelerator complexity, leading to sub-optimal speedups. We can instead compute the speedups based on number of clock cycles alone, which reflects that potential acceleration by exploiting instruction and data parallelism. The resulting geometric mean speedup for set *A*, *B* and *C* is of: 1.54×, 2.11× and 1.25×, respectively. The accelerator design presented in the next chapter is more efficient, avoiding decreases in clock frequency even for accelerator instances with numerous configurations.

Finally, a small evaluation of the PLB-interface based accelerator was also performed using 7 of these benchmarks. Also, for this evaluation, the translation process did not perform list scheduling, did not attempt to optimize memory port usage via intelligent placement of loads and stores, and employed the runtime arbitration for the MAM. The resulting geometric mean speedup was of 1.44×, versus a speedup of 1.85× in this evaluation, for the same subset. The following section discusses the effects of memory access optimizations further.

5.3.5 Effects of Memory Access Optimizations

The performance results shown so far relied on translation runs which placed *load/store* FUs according to the cost metric explained in Section 5.2.1 (by making use of list scheduling), and which generated the MAM static schedules by allowing stores to be postponed (as per the example in Fig. 5.5). An initial implementation of the translation tools instead performed a *greedy* placement and scheduling. Units were placed as early as possible, without regard to any units of the same type already present, and the static access schedule did not allow for *stores* to be postponed.

All 73 detected traces were processed with this later approach to arrive at a comparison between the two cases. For 63 out of all 73 traces, no improvement is found in terms of IPC. Table 5.2 shows the 10 traces for which the achievable IPC increased. The average increase in IPC is small for all cases, being of 9 % on average. Any improvement in IPC is a result both of the FUs placement and of the static access schedule. We can observe that the former has little impact, and that most of the optimization is due to the later.

Pair-Aware Placement In Section 5.2.2, the placement cost metric used by the translation tool for these units was explained: a cost of 2 when placing a unit on a row with no units, 1 if the row has an even number of units, and 0 for an odd number. In truth placing a new unit in a row with an even number of units incurs the same cost than placing it in a row which contains no units.

As an example, consider an accelerator with two rows, and a total of 3 load nodes to place. Placing all three loads on the first row would lead to an execution latency of 3 clock cycles for that row: the activation cycle during which other FUs execute and during which the access strobes for the first pair of loads are concurrently issued; a second cycle to read the data for the first pair of loads and issue the access strobe for the third load; and a third cycle to read the third datum. These three cycles would add to the one clock cycle required by the second row (without load), totalling four. If two loads were placed on the first row and another on the second, the first row would require 2 cycles, and the second 2 cycles, hence the result would be equal. The same would apply for the placement of four loads into two rows: the total number of cycles would be minimized by minimizing the number of rows with an odd number of units.

This is not completely unexpected by mere inspection of the accelerator's execution model. Consider a total number of N memory operations to execute. Placing each on a different accelerator row means a penalty of N additional cycles per iteration. At best, this is reduced to $N/2$, if all accesses are paired. However, even for this subset of 10 cases, an as-soon-as-possible placement for these units leads to a near optimal placement, as only one or two units are left unpaired at most. In short, only one clock cycle is typically gained by pairing-aware placement.

Store Postponing Postponing stores to later cycles is a conjunction of two effects, depending on whether they are postponed to: 1) execution cycles pertaining to the same iteration or 2) cycles belonging to the next iteration (at most). The former case is essentially equivalent to placing the store at a lower (i.e., later) topological level. The later is a limited form of loop pipelining, where only store operations of up to two consecutive iterations are partially overlapped.

This is evident for *b14*, and in a smaller degree for *b5*. These cases contain the most stores per other type of operation, meaning that optimizing their execution has greater impact. This is especially true for *b14*, whose number of accelerator rows is only 3. The number of stores per arithmetic instructions correlates with the differential increase in IPC_{HW} with a coefficient of 0.96, whereas there correlation with the number of loads per arithmetic instructions is only of 0.57.

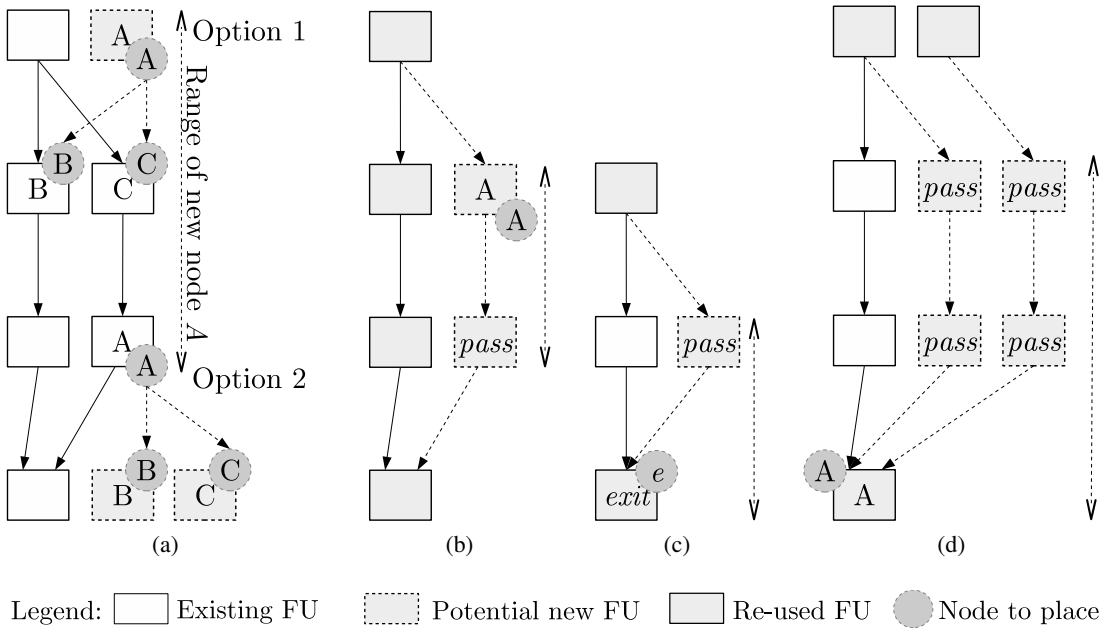


Figure 5.8: Effects of list scheduling on instantiation of *passthrough* units

In general, the inherent limitations of this execution model hinder these optimization efforts. These findings lead to a re-design of how operations are executed on the accelerator, especially memory operations, in order to decrease their impact on performance, as shown in Chapter 6.

5.3.6 Effects of List Scheduling on Functional Unit Reuse

List scheduling has an effect on the total *load/store* latency within the context of one configuration, but mostly it affects FU re-utilization between configurations. To evaluate this, the translation tool was run with list scheduling disabled for the 9 benchmarks with multiple configurations. The average decrease in the number of required FUs (including *passthroughs*) due to list scheduling is only of 1.7 (2 %). When discriminated, the average decrease in number of required FUs, excluding *passthroughs*, is of 1.8, and the number of required *passthroughs* actually increases by 0.1.

The list scheduling process has direct and indirect effects on several aspects: the total number of arithmetic (i.e., non-*passthrough* FUs), the total number of *passthroughs*, and the number of FUs that are saved (i.e., not instantiated via re-utilization of existing available FUs by list scheduling search). This is exemplified in Fig. 5.8. The effects of list scheduling on the final result end up being very unpredictable since the process is essentially a local, per-node, search. Thus it is removed from the overall context of the CDFG, and from the order in which CDFGs are translated.

Effect on Number of FUs Due to this, re-utilizing a given FU might actually result in an increase in the total number of FUs. For instance, a node a feeds nodes b and c. When placing an FU for node a, its latest possible position is used to re-utilize an existing compatible FU, which avoids the instantiation of one new FU at any one earlier row. However, when placings nodes b

and c, any two existing compatible FUs that remained in rows preceding the row chosen for node are now unreachable. Two new FUs may have to be instantiated (Fig. 5.8a).

Effect on Number of Passthroughs The number of *passthroughs* is especially unpredictable since insertion of *passthroughs* is performed post-placement, and the list scheduling search is unaware (i.e., it does not predict) of the effects of assigning a node to a particular row. Moving a node through its range of rows has a number of effects on *passthroughs*: 1) if the node is between two other arithmetic nodes, which are themselves bound to a single position, the number of *passthroughs* required to complete the connections between all three will be the same regardless of the row chosen (Fig. 5.8b); 2) moving either an exit type or store operation downwards relative to its earliest possible position to re-utilize an existing FU may implies an increase in the number of *passthroughs*, since these two types of nodes have no outputs themselves (Fig. 5.8c); 3) nodes typically have two inputs and one output, so moving a node downwards to re-utilize a unit decreases the amount of *passthroughs* between that node and any downstream consumer nodes, but may lead to a greater increase of *passthroughs* preceding the node in order to feed its inputs (Fig. 5.8d).

Effect of Order of Translation on Placement The list scheduling of each individual node is itself a local search, but the order in which a set of CDFGs is translated also influences the final number of FUs. The translation of each CDFG is under initial conditions due to the translation of the previous CDFGs. Consider the following examples, and that henceforth *passthroughs* are not included when numbers of FUs are mentioned.

As a first example: the two similarly sized CDFGs for *b16*. Both CDFGs contain nearly the same number of nodes (115 and 105), and depth of 17 and 16 levels. The total number of FUs and *passthroughs* when translating the CDFGs without list scheduling is independent of the order of translation. For this case, there are 129 FUs. A total of 91 FUs are reused due to the similarity of the CDFGs, regardless of translation order. Enabling list scheduling, and translating the CDFGs starting with the smallest, the total number of FUs is 136. This minor increase took place and only 84 FUs were reused when translating the second CDFG. Reversing the translation order yields a total of 132 FUs . In this case, 88 FUs are reused when translating the second CDFG.

As a second example, consider two of the CDFGs for *c6*, one being composed of 11 nodes, and the second of 106 nodes. Without list scheduling, the total number of FUs is of 109; 8 are reutilized when translating the second CDFG, regardless of translation order. With list scheduling, the number of FUs is nearly the same for both translation orders (109 and 108). But translating the smaller CDFG first does lead to 19 fewer *passthroughs* being required.

Even for a single CDFG to translate, the different positions of nodes with available slack would affect the total number of *passthroughs*, since an FU's position may influence the number of upstream and downstream *passthroughs* as previously explained. However, the exploration of list scheduling in the translation tool was geared towards reusing existing non-*passthrough* FUs (reuse of these units is performed during a later translation stage). This is why some CDFGs,

when translated after previous translations have taken place, end up requiring less *passthroughs* compared to a run when they are processed first. The existence of previous FUs when translating a new CDFG causes different node positions to be explored, leading to this side-effect.

For this accelerator architecture, this has an influence that is hard to quantify in terms of resources. additional *passthroughs* which appear due to node movements do not represent additional FFs cost,, they may increase the complexity of the interconnections. However, the number of *passthroughs* would become significant for a multi-row architecture capable of pipelining (i.e., activation of multiple rows concurrently). In this scenario, data must be properly synchronized between rows (as per a typical fully pipelined datapath). Additional register stages which appear as a consequence of the list scheduling process could imply significant additional resources.

To conclude, the list scheduling efforts for this particular architecture do not yield any significant improvements. Additionally, even if a decrease in number of arithmetic FUs had taken place, it is difficult to relate these high-level design aspects to low-level resources (i.e., LUTs and FFs), due to both the fact that other accelerator architectural aspects vary in function of this and the effects of the synthesis tools optimizations.

5.3.7 Resource Requirements and Operating Frequency

Figure 5.9 shows the resource requirements of the accelerators, and the reported synthesis frequency. The resources (left-axis) are normalized to the resource requirements of a single MicroBlaze: 1361 LUTs and 1028 FFs (according to synthesis reports). Considering all cases, the average accelerator requires 2.8× the LUTs and 2.6× the FFs that of a MicroBlaze. The average accelerator corresponds to approximately 50 % of all system resources (excluding BRAMs). The average number of slices for the entire system is 2027. According to post-map reports, a MicroBlaze requires 640 slices. As a side note, each LMB multiplexer requires 308 LUTs and 147 FFs.

For this accelerator architecture, the number of FUs (including *passthroughs*) does not correlate strongly with the number LUTs. The correlation coefficient is of 0.50. Relative to FFs the coefficient is 0.72. By excluding *passthroughs*, these two values become 0.72 and 0.89. This is expected since: 1) the specialized connectivity represents LUT costs which do not necessarily scale linearly with the number of FUs and are also affected by multiple configurations; and 2) the *passthroughs* are no longer registered in most cases.

The accelerator for *c5* requires the most LUTs. Although its does not contain the most FUs, nor the highest number of rows or configurations, the joint complexity of these factors results in the highest resource requirement in terms of LUTs. The largest accelerators in terms of number of FUs are actually those for *b4* and *b9*, which are also two of the five cases with the highest number of rows. Despite this, they both require approximately half the LUTs of *c5*, and the comparable case of *b16*, due to supporting only one configuration. The cases for which resource requirements were lower are, predictably, those with fewer FUs and supporting only a single configuration.

The connectivity specialization gains can be measured against the resource requirements from the accelerators in Chapter 4. Considering all cases, the accelerators for this evaluation require 0.90× the LUTs and 1.69× the FFs relative to the average accelerator in Chapter 4 (including the

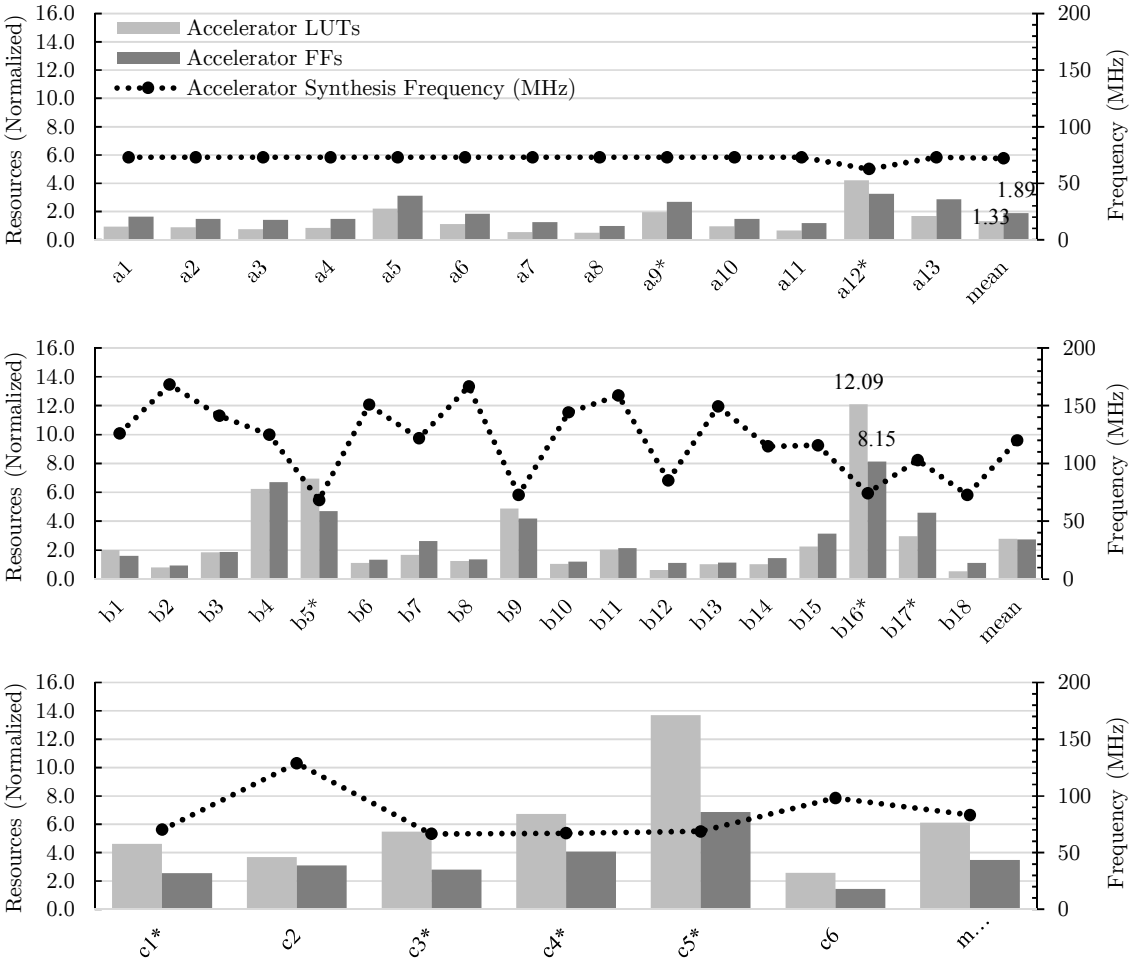


Figure 5.9: Resource requirements and synthesis frequency of the generated accelerators.

two multi-configuration cases). The decrease in LUTs is most likely related to the connectivity specialization. The increase in FFs is related to the increase in number of FUs, as each non-*passthrough* FU registers its outputs.

For a better comparison, consider a subset composed of the 15 benchmarks containing the least amount of FUs (excluding *passthroughs*), taken from all three sets of this evaluation. The average number of FUs in this subset is 16, which is still 1.92× as much as the average for the 15 single-configuration cases in Chapter 4 (i.e., excluding *m1* and *m2*). Still, the average number of required LUTs and FFs is 0.38× and 0.89× relative to that case, respectively. As a second comparison consider the multi-configuration cases of Chapter 4 (*m1* and *m2*), and cases *c1*, *c3*, and *c6*, which have a similar number of FUs, *passthroughs* and configurations. These three cases required an average of 0.63× the LUTs and 0.89× the FFs relative to *m1* and *m2*.

The average synthesis frequency is 97 MHz when considering all benchmarks, but there is a distinct behaviour between sets in terms of operating frequency. For set *A*, the *if-conversion* process adds at least one multiplication instruction to the trace. As a result, the frequency for all cases is identical (save one) because the critical path is the same in all instances: it occurs due

to the multiplication FU. For *a12*, the division FU contains the same multiplication logic, and requires additional arithmetic, leading to a slightly worse synthesis frequency. The critical path is due to an inefficient multiplication FU. The first accelerator design required single-cycle FUs, so the multiplication was implemented as a combinatorial operation, which was reused in this design.

For set *B*, there is a greater variability in synthesis frequency. For *b5*, *b9*, *b12*, and *b16* through *b18*, an instantiated multiplication unit is again the cause of the critical path. For the remaining cases, there is no single reoccurring critical path throughout all accelerators, but the critical path for each typically involves signals from the memory port logic. The frequency for these cases varies relatively little (a standard deviation of 18.4 MHz around a mean of 140.2 MHz). Consider again the subset of 8 kernels tested with the PLB based accelerator, which instead of the static memory access schedules employs the runtime arbitration. The average synthesis frequency is of 107.7 MHz, versus the 122.2 MHz for the same subset according to the results shown in Fig. 5.9.

For set *C*, every critical path includes a multiplexer followed by an FU (either a multiplier or the carry chain of an adder), except for *c3* and *c6*, where memory access logic introduces the critical path. The average number of LUTs for the accelerator in set *C* is 2.19× that of set *B*, and the frequency is 0.69× lower. This is a considerable downturn in cost per performance, especially since the geometric mean speedup is 0.96× (only three cases benefit from a modest speedup).

The synthesis frequency of the accelerator is lower than the baseline 66 MHz only for *a12* and *c3*, but when accounting for the entire system, 9 systems operate below the baseline frequency. These cases are marked with an asterisk (*) in Fig. 5.9: *c5* operates at 33 MHz and the remaining cases at 50 MHz. For *c5* the *passthroughs* in the first and third rows were registered, to attempt a reduction in critical path length. For all other cases, the *passthroughs* were not registered.

Decreases in frequency occur mostly for set *C*, for which the accelerators are much larger than the average. For instance, the number of LUTs required by the accelerator for *c5* corresponds to 96 % of all system LUTs. This corresponds to over half the LUTs on the device. This in itself justifies the lower synthesis frequency as the designs become more constrained, although it is merely a consequence of employing a low-end Spartan-6 FPGA. The largest system for these experiments, *c5*, requires 5929 slices. This is not a large design relative to high-end devices (e.g., the Virtex-7 family, with up of 74650 slices), but represents 87 % of the Spartan-6 FPGA used.

Regardless, this type of multi-row design and translation quickly becomes less viable for large traces, especially if a rich connectivity to support multiple configurations is desired. This, in part, lead to the final accelerator design presented in the next chapter, which deals efficiently with multi-loop supporting, incurring very little cost for additional supported loop, registering nearly no decrease in operating frequency, and maximizing the performance of each CDFG.

5.3.8 Power and Energy Consumption

Augmenting a MicroBlaze with an accelerator has consequences both for performance and for power and energy consumption. The additional hardware implies additional power consumption, and in migrating some of the application execution to it, the power consumption of the MicroBlaze and remaining system elements varies as well. The application domain targeted by this approach

Table 5.3: Power consumption for software-only and accelerated runs

ID	Software-only Power Consumption (mW)			Accelerator-enabled Power Consumption (mW)			
	System	MB	BRAM	System	MB	BRAM	Acc.
b15	145.78	15.46	8.76	141.79(-3%)	11.58(-25%)	6.26(-29%)	0.38
b17	149.84	19.87	7.72	144.74(-3%)	15.16(-24%)	5.76(-25%)	0.14
c1	148.69	18.67	8.80	139.64(-6%)	8.45(-55%)	5.28(-40%)	3.35
c2	147.24	17.25	8.45	141.38(-4%)	12.11(-30%)	5.91(-30%)	0.87
c3	127.06	8.32	1.90	123.34(-3%)	4.63(-44%)	1.10(-42%)	0.70
c4	149.04	19.38	8.93	127.18(-15%)	2.09(-89%)	1.75(-80%)	5.89
c6	150.39	18.97	8.39	132.88(-12%)	5.71(-70%)	3.47(-59%)	4.47
mean	145.43	16.85	7.56	135.85(-6%)	8.53(-48%)	4.22(-44%)	2.26

Table 5.4: Energy consumption for software-only and accelerated runs

ID	Software-only Energy Consumption (uJ)			Accelerator-enabled Energy Consumption (uJ)			
	System	MB	BRAM	System	MB	BRAM	Acc.
b15	2.61	0.28	0.16	1.15(-56%)	0.09(-66%)	0.05 (-68%)	0.00
b17	2.89	0.38	0.15	2.66(-8%)	0.28(-27%)	0.11 (-29%)	0.00
c1	108.32	13.60	6.41	130.56(21%)	7.90(-42%)	4.94 (-23%)	3.13
c2	74.50	8.73	4.28	62.13(-17%)	5.32(-39%)	2.60 (-39%)	0.38
c3	38.37	2.51	0.57	34.00(-11%)	1.28(-49%)	0.30 (-47%)	0.19
c4	130.29	16.94	7.81	115.79(-11%)	1.90(-89%)	1.59 (-80%)	5.36
c6	2285.95	288.35	127.53	1834.21(-20%)	78.82(-73%)	47.90 (-62%)	61.70
mean	377.56	47.26	20.99	311.50(-15%)	13.66(-55%)	8.21 (-50%)	10.11

are embedded tasks. Depending on the specific application, the relevant metric might be either energy (for energy-constrained systems) or power (for applications which execute continuously during an indeterminate period of time).

To estimate the power consumption of both software-only and accelerator-enabled runs, a subset of the implemented systems was put through ModelSim simulation. For each benchmark, the respective post-place-and-route system was simulated to application completion. The testbench generated a node activity file which was used by Xilinx's XPower Analyzer to estimate a hierarchical breakdown of the average power consumption. The total energy consumption was thus computed from the execution times.

Table 5.3 and Table 5.4 show the power and energy consumption for 7 benchmarks, mostly from set *C*. The first three columns show the hierarchical power consumption for software only execution, and the following four for accelerator-enabled execution. The *System* category accounts for all components (MicroBlaze, BRAMs and accelerator included). The last three columns show the changes in energy consumption, percentage-wise, by taking into account the execution time for each case. The execution times are computed given the operating frequency of each case and the

number of clock cycles required to execute the benchmark, as measured by timer/counter modules. In general, the energy consumption decreases for all cases, despite the fact that the operating frequency is of 50 MHz for *b17*, *c1*, *c3* and *c4*. Directly comparing the energy consumption of these benchmarks is not very conclusive, due to the different running times of each case.

There is a general marginal decrease in average power consumption for the entire system. Regardless of the total execution time and total energy consumption, the average power consumption for the MicroBlaze and BRAMs decreases for all cases, the averages being of 48 % and 44 %, respectively. The greatest decreases in power consumption by both the MicroBlaze and BRAMs happen for *c1*, *c4* and *c6*. This is due to the number of iterations executed on the accelerator for these cases being the highest for this subset, i.e., more work is offloaded from to the accelerator. The decrease in BRAM power consumption is related, since the accelerator does not have to access it to retrieve instructions; the total number of data accesses, however, remains the same.

For *c1* and *c2*, the operating frequency of the system, when augmented by the accelerator, was of 50 MHz. As a result, there were marginal slowdowns. For *c1*, the relationship between decreased power and increased runtime actually resulted in an increase in total energy consumption. For *c4*, the trade-off was beneficial, energy-wise, leading to a small decrease in energy consumption in exchange for a decrease in total execution time. Note however that the decrease in operating frequency penalizes the portions of the application which do not execute on the accelerator.

Executing the most code in the accelerator for these cases also means its power consumption is higher, but for all three cases the accelerator consumes less power than what was saved on the remaining system. For the entire subset, the average increase in power consumption due to the accelerator is of 2.26 mW, whilst the difference in total power consumption is of 9.58 mW, in favour of the accelerator-enabled run. The power analysis also breaks down consumption into total and dynamic. In this evaluation, the dynamic power consumption is of relevance, since any differences are due to the different switching activities of the MicroBlaze and accelerator. The average decrease in dynamic power is of 4.29 mW, which corresponds to a decrease of 3.58 % in dynamic power alone, and 2.10 % of the total power consumption.

5.4 Concluding Remarks

The main contribution presented in this chapter is the flexible support for accelerator memory access. By allowing for the accelerator to directly access the entire data memory of the processor, without incurring data transfer overhead, it is possible to accelerate a wider range of trace loops. This is very significant as data-oriented loops operate on one or more input data arrays or streams, typically producing a set of data per iteration. The support for two concurrent accesses to arbitrary addresses efficiently exploits data parallelism, as the experimental results demonstrated.

Additionally, the translation tools were modified in an attempt to better utilize FUs by employing list scheduling. Although a minor improvement was observed, it only occurred for a small subset of all targeted traces. The re-utilization of FUs in this manner is limited by the node mobility, and is also a function of the similarity between the CDFGs being translated.

The greatest improvement on resource requirements, relative to the previous accelerator implementation, was the specialization of the inter-row connectivity. Since less reconfiguration information is required the migration overhead is decreased. The resource requirements also decrease, with the average accelerator instance requiring approximately as many resources as a MicroBlaze processor. Abandoning the crossbar based design is a valid design decision, since the connectivity is effectively known *a priori*. General interconnects such as crossbars are more appropriate for scenarios where the configuration information is generated during runtime (i.e., on-chip).

A significant limitation of the architecture presented in this chapter is the lack of exploitation of intra-iteration ILP. That is, the accelerator is only capable of concurrently executing operations belonging to the same trace iteration. As a result, many FUs remain idle during accelerator executing at any one given time. Finally, the lack of support for floating-point operations also hinders the applicability of the approach. The following chapter deals with these two issues by relying on loop-pipelining and introducing fully-pipelined FU for floating-point operations.

Chapter 6

Modulo Scheduling onto Customized Single-Row Accelerators

In the previous chapters it was shown that accelerating the targeted loop traces, i.e., Megablocks, results in noticeable performance improvements over a software baseline. The multi-row accelerator architectures and tool chain proved sufficient as proofs of concept for the transparent binary acceleration approach. Augmenting the first accelerator design with memory access support further demonstrated the acceleration possible with straightforward and automated accelerator generation.

However, the presented multi-row architectures still suffer from some drawbacks. Firstly, there are resource considerations. Since the multi-row array is a one-to-one translation of CDFGs, the amount of required resources becomes prohibitive for large CDFGs. Large CDFGs are typically extracted from large loops, which means that being unable to accelerate these regions decreases the applicability of the approach. Also, as the number of Megablocks used to generate a multi-row accelerator increases, the resource requirements due to multiplexers also increases considerably.

Secondly, directly translating CDFGs into a multi-row array is inefficient in terms of resources, since the execution model is intrinsically defined by the structure of the graphs. The re-utilization of each FU is limited, since only local searches within the list scheduling range of each node are performed. For instance, an *add* FU in the first row cannot be reused by a node which must be scheduled between the third and fourth rows. A new FU is required, and the *add* in the first one may not even be used at all during that time. Also, for the presented architectures, only one row is active at any single time, which leaves all other resources idle. Each row of the array represents one topological level of the CDFG, so loop pipelining could be achieved by backwards connections, increasing the per-cycle utilization of resources. However, this still would not be an optimal solution. For example, if the Initiation Interval (II) is 2, then each row is idle for one out of two clock cycles; if the II is 3 then each row is idle for two clock cycles out of every three. The II can also increase considerably due to access contention to the limited number of memory ports. Adding pipelining capabilities to the accelerator would only worsen this, since more memory access operations would activate per-cycle. Appendix A demonstrates these effects with an implementation of the multi-row architecture, augmented with loop-pipelining.

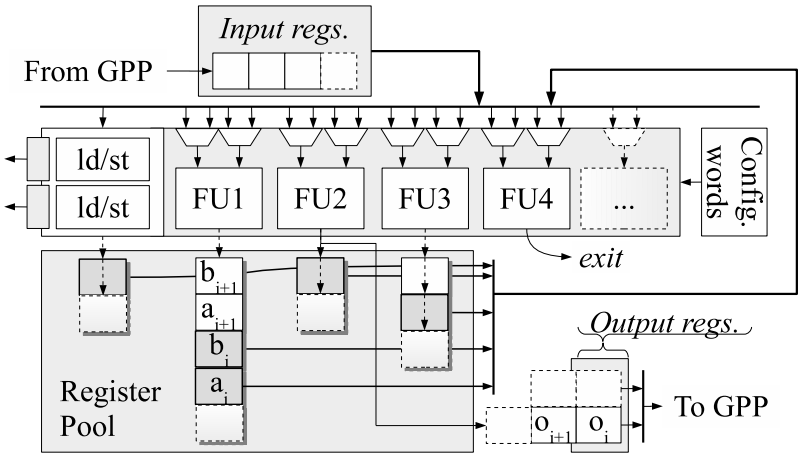


Figure 6.1: Accelerator template, containing one row of Functional Units and a possible register pool structure, feeding back into multiplexers

This chapter presents an accelerator architecture which addresses these issues in a different manner. Like previous implementations, the accelerator is a heavily parametrizable module, customizable at synthesis time. Figure 6.1 shows the simplified architecture template. Unlike previous designs, this architecture is based on a single customizable row of FUs, similar to a VLIW layout. Previous implementations relied on directly translating the structure of CDFGs into hardware elements. Instead, this implementation relies on a scheduler which efficiently generates compact accelerators by instantiating FUs while modulo scheduling operations.

Adopting a template based on a single row comes from the realization that multi-row designs are adequate when the required inter-FU connectivity is unknown. Namely, inter-FU connectivity must be rich enough to increase the likelihood that future graphs/loops can be successfully mapped, Simultaneously, it must not be too complex due to resource costs, e.g., full crossbars. A mesh design addresses the connectivity issue by transporting data between nearest neighbours and using the FUs themselves to transport data. A row-based array relies on a single direction for data-flow and instead increases the amount of FUs to ensure that graphs can be mapped.

For both cases, the translation/scheduling tools must be sophisticated enough to keep track of data as it travels through the array, especially when loop pipelining via modulo scheduling. For a row based design, scheduling requires spatial awareness along the width of the array and temporal awareness along its height. In either case, it is more difficult to find valid schedules as the interconnect, FU and storage availability decreases.

In summary, data transport and storage dictate much of the design aspects of an accelerator, more so than the availability of FUs. By generating fully custom accelerators, and not only configuring an existing accelerator structure, a simpler single-row model for allocation of units can instead be used. The connectivity and registers for storage can be specified after node scheduling and FU instantiation, implementing only the required connections to fulfil the data flow required by the translated graphs. The consequence of spatially compacting the accelerator architecture into a single row is that FUs are better utilized. All topological levels of the CDFG are executed by the

same resources. The previous multi-row designs remain (reasonably) adequate for loops with an II of 1, since resource utilization is increased. However, the single-row architecture copes more efficiently with graphs containing more *load/store* operations. Also, large CDFG are supported more efficiently, since the accelerator size does not scale linearly with the size of the CDFGs.

Throughout this chapter the single-row modulo scheduled accelerator is presented and evaluated. Section 6.1 details its architectural aspects. The main differences relative to previous implementations are the full support for single-precision floating-point operations and the execution model, which efficiently implements loop pipelining.

Supporting floating-point operations allows for targeting realistic embedded workloads. These types of operations require several arithmetic steps which could lead to critical paths if implemented in a single clock cycle, so some of the developed floating-point FUs are multi-cycle. This meant adapting the accelerator accordingly. Unlike a VLIW however, these units are fully pipelined, meaning that a floating-point operation can be completed every cycle on the same FU. Also, the scheduler is sophisticated enough so that multiple other single-cycle operations continue to be issued on other FUs while a previously activated multi-cycle unit operates.

The supporting scheduler is shown in Section 6.3 along with a complete scheduling example. In Section 6.4 contains an extensive experimental evaluation of the fully implemented accelerator for a total of 25 benchmarks (13 floating-point kernels and 12 integer kernels). This architecture is also compared with several fixed-resource accelerators in Section 6.5 (e.g., instances with a fixed, manually specified, number and type of units) to determine the advantages and disadvantages of employing the type of proposed full customization approach. Section 6.6 the modulo scheduled accelerator is also compared with several VLIW processors, via Hewlett Packard's VLIW Example (VEX) simulator [FFY05], and the open-source ρ-VEX processor [SABW12].

6.1 Accelerator Architecture

Figure 6.1 shows the simplified template for the single-row accelerator. It includes: input and output registers; a set of 32-bit FUs; two *load/store* ports present in every accelerator instance; multiplexers to route FU inputs; a register pool of 32-bit registers to hold operands/results; and a configuration memory. This memory is read-only and implemented with LUTs (i.e., distributed memory). Specialization of this template involves determining the number of: input and output registers; FUs and their type; the number of registers in the pool and their layout. Also, multiplexer connectivity is specified and a set of configuration words is generated per supported loop trace.

The *input* and *output* registers hold data exchanged with the MicroBlaze via an FSL interface (*from GPP* and *to GPP*). Operands are shifted into the *input* registers, which are fed into the FUs. These registers are read-only; they hold values that are valid for the first iteration (e.g., initial value of an iterator variable) and values that remain constant throughout all iterations (e.g., iteration bound or a base address for memory accesses). The *output* registers are fed by the FU outputs. Since modulo-scheduling overlaps loop iterations, multiple values for the same variable may be produced before a single iteration (i.e., a full set of output values) is complete.

To address this, each FU drives a chain of registers which hold the output values of the CDFG operations it executes. Each chain's length is determined during scheduling, by computing how long each value needs to live until it is consumed by all downstream operations. This is exemplified by *FU1* in Fig. 6.1, where results for two CDFG nodes, *a* and *b*, for iterations *i* and *i+1*, are stored.

The multiplexers are specified based on the knowledge of where every computed result resides at every time step, which is known since the schedule is static. The required values per FU input are fetched from the appropriate register, based on the CDFG operations the respective FU executes (shown in grey in Fig. 6.1). For example, the value a_i might be produced during time step 1, and be consumed during timesteps 2 and 3. Value a_i can be read from the first register in the chain during timestep 2, but if b_i is produced in the same timestep, then a_i will be read from the second register during time step 3. This information is kept for all generated values during scheduling and is used to generate the required connectivity and configuration words.

An accelerator instantiation will contain as many FUs as necessary to achieve a minimum II for all the scheduled loops. Integer and single-precision floating-point arithmetic and comparison operations are supported, as well as bitwise logical operations. Other operations include conversion from floating-point to integer and vice-versa, and a set of units to evaluate termination conditions. Each FU implements a single operation type, except for floating-point addition and subtraction, which are implemented in a single FU. All FUs are pipelined, except for the non-constant integer division and floating-point division, which are multi-cycle units. Implementing pipelined FUs avoids the need to increase the II while scheduling by effectively increasing resource availability.

Most integer units have a latency of 1 clock cycle, except for the division FU (35 clock cycles). A specialized integer division module provides division by a constant with a latency of 3 clock cycles (via reciprocal multiplication). Like the MicroBlaze processor, the floating-point units do not support denormalized operands or issue denormalized results. The floating-point addition, multiplication and division units have latencies of 4, 3 and 32 clock cycles, respectively.

Memory access patterns can be arbitrary, since the loop operations which generate access addresses are also executed on the accelerator. A constant memory access latency is assumed when generating schedules, but the accelerator is also capable of stalling when an access takes longer than the expected time. Since the implementation presented here relies on on-chip FPGA memories, the *load/store* units have a latency of 2 clock cycles.

6.1.1 Execution Model

The accelerator is idle until it receives a single 32-bit word via a separate FSL (not shown). This determines which configuration words to read from its internal configuration memory, which FU will fed each output FIFO, and how many operands to expect from the GPP before executing.

Operands are received via FSL in a specific order, as the accelerator expects each particular processor register value to reside in a given input register. After the expected number of operands is received, the accelerator sets the starting address into its configuration memory according to the loop to execute. One configuration word is then read per-cycle. The accelerator executes the steps of a loop schedule by cycling through a number of configuration states.

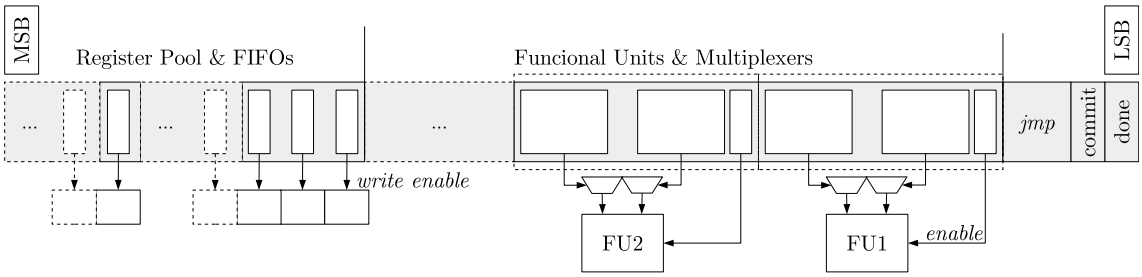


Figure 6.2: Representation of one configuration word; width varies per accelerator instance.

A generic representation of a configuration word is shown in Fig. 6.2. Configuration words are single-cycle, and their bit-width varies per instance, depending on the number of FUs, multiplexer widths and number of registers in the pool. A single word defines the complete state of the accelerator. That is: which FUs are active, the multiplexer controls and which registers in the pool will be written. It also signals when to commit values to the output FIFOs and when execution is concluded. Unlike the input multiplexers, who can be controlled every cycle by the configuration words, the output FIFOs are driven by a single FU output throughout the execution of a loop. The *jmp* field denotes how to update the configuration word access address.

Configuration words are read by a 2-stage pipeline, similar to *fetch* and *decode/issue*. The two stages are required to update the configuration access address in a timely fashion, without introducing idle time when cycling through the words of the steady state. One word corresponds to one time step of the respective modulo schedule. A schedule has as many t time steps as required so that all the N operations of the loop are executed on the F instantiated FUs, where $F \leq N$.

Executing operations in multi-cycle FUs, such as the division unit, does not halt execution. Configuration words continue to be read while the operation completes, as the static schedule takes into account the FU latency. If the FUs is also pipelined, (e.g., floating-point addition), both aspects are taken into account while scheduling: the FU can be enabled every cycle and it is known at which time each result will be produced. In general, operations without control or data dependencies execute in the same time step. By scheduling, in a single time step, operations from two or more successive iterations (loop-pipelining), iterations are initiated and completed in less than t time steps, depending on the data and control dependencies between successive iterations.

The number of iterations does not need to be known either at synthesis time or prior to the start of execution. The termination conditions (i.e., loop exits) of the executing loop can be evaluated on every iteration by the FUs. When an iteration triggers an exit, the *jmp* field is ignored from that moment on. Execution continues through the configuration words of the epilogue. At this stage, the most recent iteration (yet incomplete) is discarded, no new iterations are initiated and ongoing iterations are completed. The top of the output FIFOs will contain a full set of output values which correspond to GPP register file contents. The GPP retrieves these values in a known order, by executing the remainder of the CR, and branches back to the start address of the Megablock, completing the last loop iteration in software.

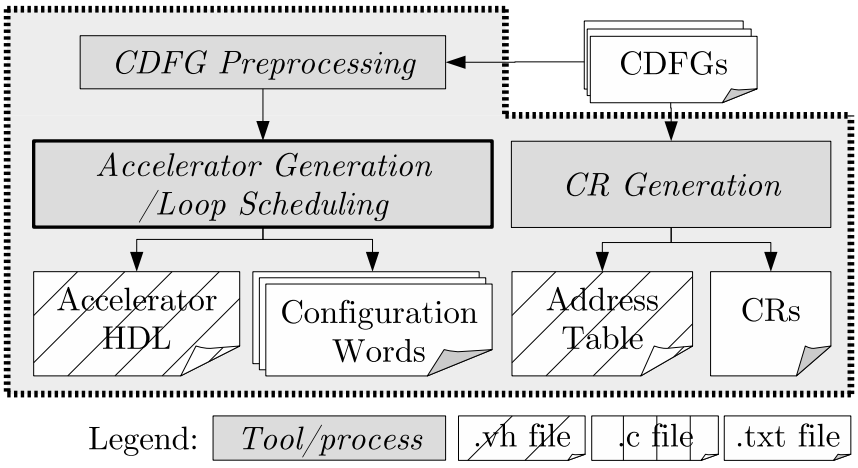


Figure 6.3: Architecture-specific flow for the single-row accelerator

6.2 Architecture Specific Tool Flow

Figure 6.3 shows the segment of the full tool flow specific to this accelerator design. This new loop accelerator architecture required re-designing parts of the tool flow considerably. The accelerator interface is unmodified relative to Chapter 5, so CRs generation is unmodified. They are placed in *C* containers in a single source code file. The application is recompiled using a custom linker script which places them at a user-defined location. The *injector*'s address table is a Verilog include file containing only the start addresses of the Megablocks and the locations of each respective CR.

The scheduler replaces the CDFG translation step in previous flows. Some of the preprocessing steps of said flows are preserved and additional minor steps transform the data structure for compliance with the scheduler. The preprocessing steps are performed by a stripped down build of the translation tool written in *C*. The configuration words are printed in text format and given as a synthesis tool compliant *MEM* file which is read during accelerator synthesis. The accelerator HDL is a single Verilog include file (unlike previous flows which produced multiple output files). It is similar to the output shown in Listing 3.1, and defines every aspect of the instantiation.

The scheduler is currently fully implemented in MATLAB. The following section explains how the scheduler generates an accelerator instance and provides a complete scheduling example.

6.3 Accelerator Generation and Loop Scheduling

When modulo scheduling [Rau94] for fixed architectures, there are resource and temporal restrictions to take into account: 1) given all types of operations in a loop, the target architecture must contain at least one FU capable of implementing each one; 2) operation parallelism and II still depend on the spatial and temporal availability of FUs, which determine performance. In these situations, when a modulo scheduler cannot schedule a loop with a given (minimum) II, the II is increased until scheduling becomes feasible. Increasing the II makes it possible to schedule a loop onto the minimum set of resources, but leads to decreased performance, especially when the

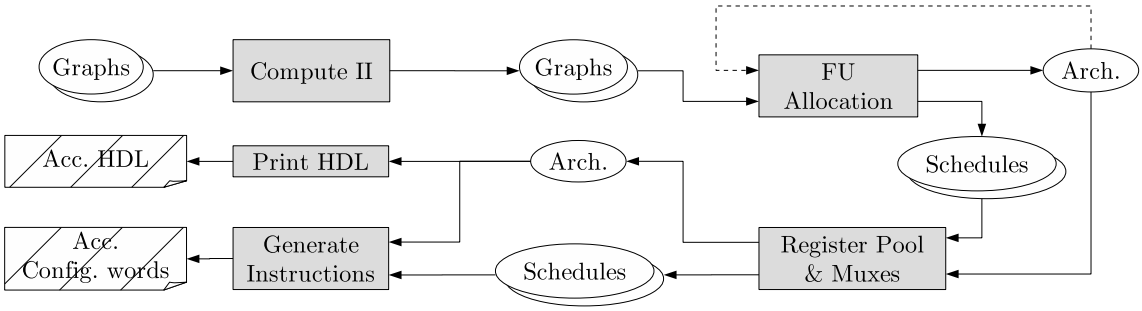


Figure 6.4: Execution flow of modulo scheduling for the single-row accelerator

minimum possible II is low (e.g., increasing the II from 1 clock cycle to 2 approximately halves the performance). The CDFG derived from the extracted Megablock traces typically have low IIs. Given this, the present approach adds the required FUs to each accelerator instance during scheduling to prevent decreasing the performance.

Figure 6.4 summarizes the stages of the modulo scheduler. The scheduler receives each CDFG as a data structure specifying each node's type, inputs and topological level. Also specified are the MicroBlaze registers which contain inputs and hold outputs. The scheduler first determines the II of each CDFG. Each graph is then scheduled independently. The instantiated FUs are fed back into the scheduling process for every subsequent graph. This first stage results in a single row of FU, and annotates each CDFG node with its assigned FU and execution time. The next step takes each annotated graph and: 1) determines how long each computed value needs to be stored for, depending on when nodes are scheduled and on the producer/consumer relationship between them, and 2) determines the length of each FU output register chain. That is, each graph imposes different storage requirements per FU. The final register pool is composed by taking the maximum lengths of each FU output chain throughout all scheduled graphs. This step also determines in which register each computed value resides at a given time. The scheduler aggregates this information for all graphs to determine the connectivity of the input multiplexers, by fetching values from the fully specified register pool. A sequence of configuration words per schedule is then generated. Finally the accelerator specification is produced in HDL, along with a memory initialization file containing the configuration words.

The following section details this process with an example, starting from an extracted CDFG and resulting in a customized accelerator instance.

6.3.1 Scheduling Example

FU Allocation Figure 6.5 shows an example of the type of CDFG the scheduler accepts. The nodes represent GPP instructions, edges represent the data flow between nodes and the inputs (top) and outputs (bottom) represent GPP registers. Input registers shown in dotted lines are only read during the first iteration. The graph itself represents an iteration of the original Megablock loop.

First, the scheduler computes the II, which can be determined by: backwards data edges (i.e., data flow across iterations), resource restrictions (in this case, the two memory ports) or control

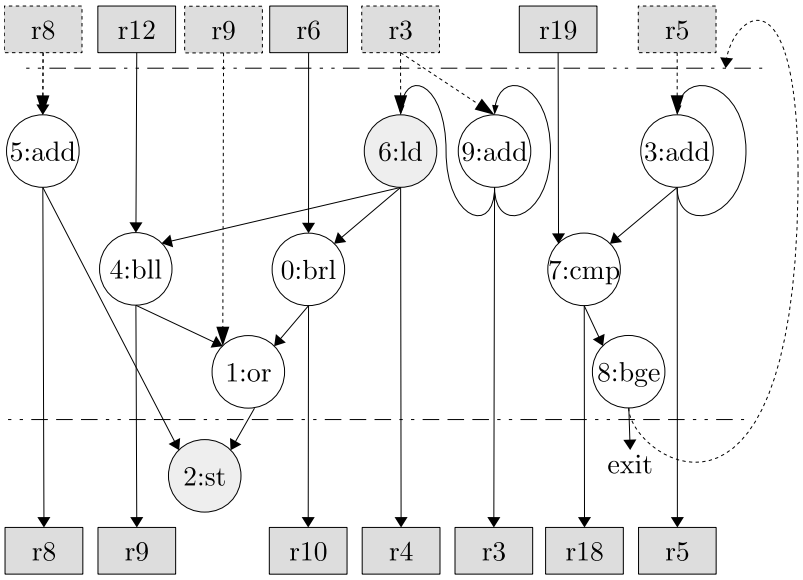


Figure 6.5: Example CDFG, showing operations and a cyclical control dependency which determines the Initiation Interval (II)

edges. It is necessary to consider control edges because the accelerator has no knowledge regarding the number of iterations to execute when invoked. That is, a new iteration can only begin after all exit conditions of the current one are evaluated as false. For this example, the *bge* (*branch if greater-than*) node sets an II of 3 clock cycles, as all nodes are executed by single-cycle FUs.

At the start of the process, no accelerator architectural aspects are defined besides the two memory ports. As nodes are scheduled, FUs are added to the array when necessary. Connectivity is assumed to be unlimited. Figure 6.6 shows the complete schedule for the nodes of Fig. 6.5, placed temporally along the vertical axis and spatially along the horizontal axis. A schedule is composed of prologue, steady state and epilogue.

Nodes are list scheduled individually in topological order. For every node, the earliest (e_t) and latest (l_t) possible schedule times are computed. The e_t of a node is calculated based on its upstream nodes (or input registers), while the l_t is typically unbound. The l_t is only relevant for nodes which originate a backwards edge, since scheduling them too late relative to the other nodes in the closed circuit would violate the II. In this example, nodes 3, 7 and 8 have no slack, nodes 9 and 5 can be delayed up to $t = 3$ and the remaining nodes could be delayed to any indefinite later time without consequence. The scheduler attempts to schedule all nodes as early as possible.

In Fig. 6.6, node 6 was scheduled first at the earliest possible time, $t = 1$. Since the II was computed to be 3, all time slots i for which $i \bmod 3 = 1$ are occupied by future executions of node 6 (shown in grey). When scheduling node 3 an *add* FU is added, and node 3 is scheduled at a time which does not violate the II, $t = 1$. Nodes 5 and 9 can also execute on this FU, since both can be delayed until $t = 3$ at most. If another *add* node existed, and if it could not be delayed to a later time, a new FU would be added and the node scheduled at its e_t.

The steady state of the schedule is a time frame with II time steps. It starts at a time t_s where

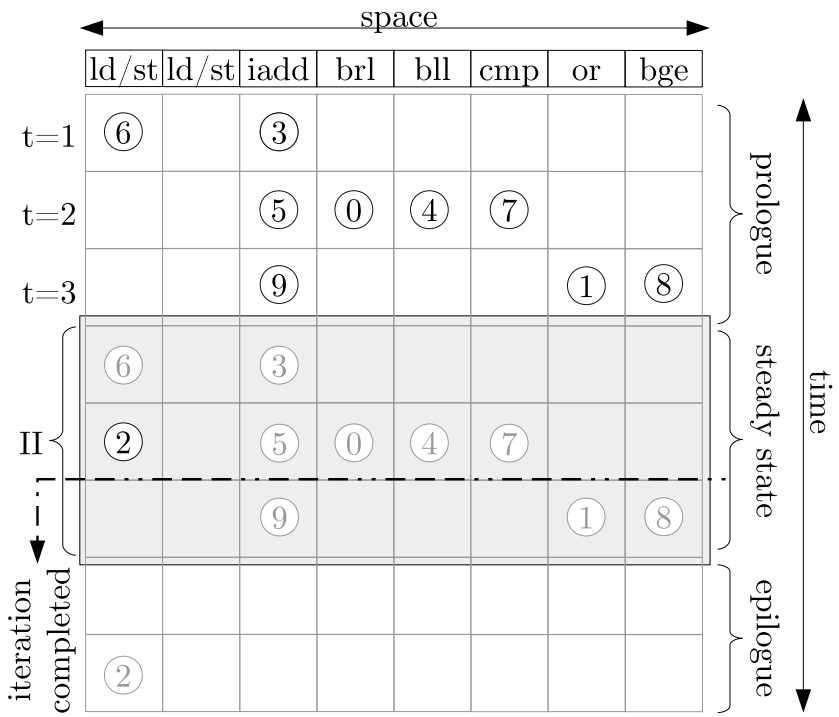


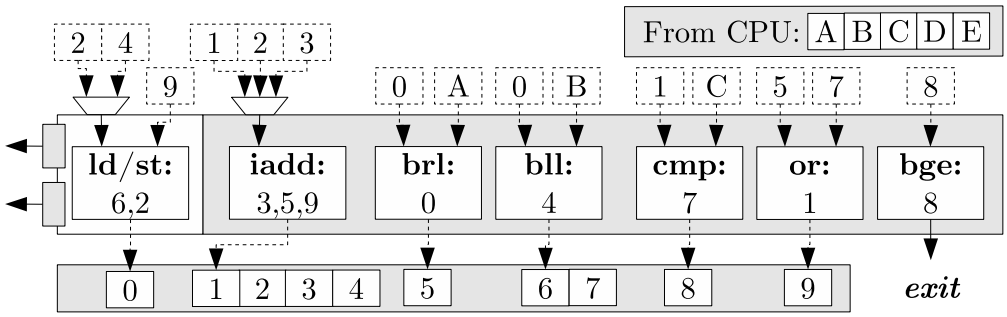
Figure 6.6: Modulo schedule for the example CDFG

$t_s \bmod II = 1$ and contains the time step where the iteration started at the prologue completes. In this case this happens after only one repetition, at $t = 5$. This sequence of timesteps can also be referred to as the Modulo Reservation Table (MRT), and it contains all the nodes of the scheduled CDFG. All time steps prior to $t = 5$ belong to the prologue, and all after $t = 7$ to the epilogue.

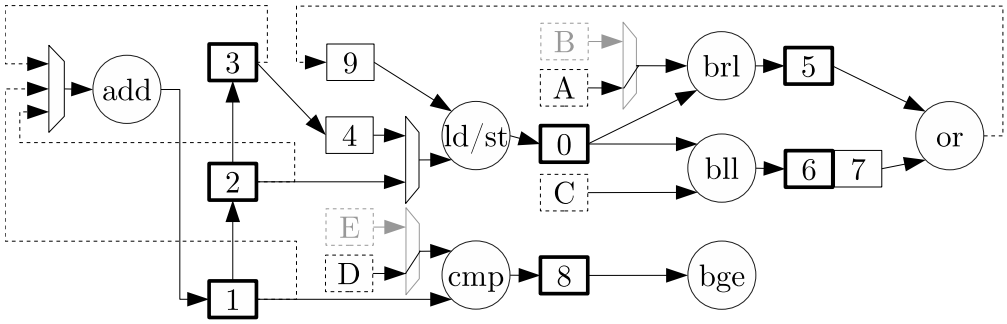
More CDFGs could now be scheduled onto the existing array. As more loops are scheduled, less FUs are added to the accelerator, since FU availability increases. However, there is an increase of the input multiplexer complexity, which depends on the connections between operations. When there is more than one available FU to place a new node, the scheduler attempts to reduce resource consumption by choosing the FU which will result in the least added inter-FU connectivity (based on nodes of previously scheduled graphs).

Architecture Specialization Everything after the scheduling stage is related to generation of the register pool, input multiplexers, output FIFOs and configuration words.

First, the scheduler computes when each value is produced and for how long it must be stored before it is consumed by all downstream nodes which require it. With this information, a chain of registers is built for each FU output. A register is added to the chain whenever a value is produced before the existing value is consumed. When a new value is produced, the existing values are shifted down. However, a value in the middle of a chain may be consumed before values produced in previous time steps. The chain length is shortened as much as possible by having an individual *write-enable* per register and shifting data only up to the point of the first expired datum.



(a) Operation assignments for one loop schedule



(b) Generated accelerator hardware structure

Figure 6.7: Operation assignments and register connections for one loop schedule and actual hardware structure of the single-row accelerator instance

The (simplified) accelerator generated for the graph in Fig. 6.5 is shown in Fig. 6.7a. The *iadd* FU requires four registers to hold results, since most will only be read by the FU in the next iteration. The *bll* FU requires two registers, because the *or* node will consume a value from the input register for the first iteration, and only afterwards will it fetch values produced by the *bll*. Two registers are required to synchronize the data across iterations.

After generating the chains of registers, it is known were each value will reside at every time step. The input multiplexers are then generated based on the location of the desired values in the registers. If a given input receives data from only one pool register, the multiplexer is optimized away. For simplicity, the connectivity is represented by the numbered boxes in Fig. 6.7a. Some FUs receive values from registers that remain constant throughout execution. Literal constants fed into the FUs are omitted in the figure.

Generating Configuration Words The last step is the generation of configuration words. A configuration word contains FU enable bits, hot-bit encoded multiplexer control bits and register pool *write-enable* bits. One instruction is generated for each schedule time step. This includes the prologue, steady state and epilogue. To generate this complete instruction stream it is only necessary to consider the first iteration (i.e., solid black nodes in Fig. 6.6). The scheduler iterates through every FU, starting from time step $t = 1$ (i.e., the prologue) up until the time step where the initial iteration ends, $t = 5$. If there is a scheduled node, the enable bit of the FU and the input multiplexer bits are set. Based on the register pool generation step, the *write-enable* bits for the

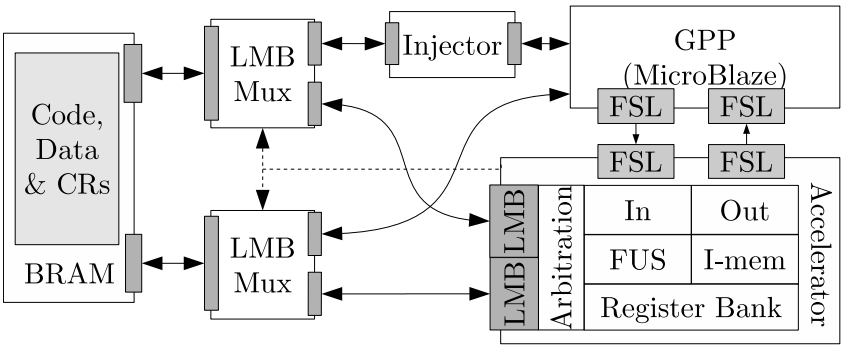


Figure 6.8: System architecture for validation of the single-row modulo scheduled accelerator

required registers in the chain of the respective FU are also set. The resulting set of words is used to create the full configuration word, from prologue to epilogue, sequence, by repeating it every II time steps, up until the time step where the steady state starts. The configuration memory contains one sequence of configuration words (i.e., epilogue, steady state and prologue) per scheduled loop.

Despite the appearance of a horizontal type layout, the dedicated, and in some cases minimal, connectivity between FUs effectively implements circuits with distinct structures. The example in Fig. 6.7 was generated from two graphs. The graph shown in Fig. 6.5 and a second very similar graph. In Fig. 6.7b most of the structure is used for either loop, but some instantiated components are only required for one of them (shown in grey). Registers with a bold outline hold values that will be sent back to the main processor.

6.4 Experimental Evaluation

The performance of this architecture and scheduling approach was evaluated with 13 single-precision floating-point benchmarks from the *Livermore Loops* [Tim92] and 11 integer benchmarks from the TEXAS IMGLIB function library [Tex]. The chosen benchmarks were those with little to no control in the inner-most loop and with little to no operations in the outer loops. That is, perfectly nested loops are preferable.

6.4.1 Hardware Setup

The accelerator was coupled to a local memory system as shown in Fig. 6.8 and similar to the implementation presented in Chapter 5: a single local memory holds all the code and data, a single MicroBlaze processor executes the application from this memory and is augmented with the modulo scheduled accelerator via an FSL. The necessary *injector* and LMB multiplexer modules are present to allow for transparent migration of execution and shared memory access. As before, the *injector* modifies the instruction bus when a Megablock start address is detected and also sends a single configuration word to the accelerator, to select which loop to execute.

The target device for this implementation was a Virtex-7 *xc7vx485* device. For bitstream generation, Xilinx's ISE Design Suite 14.7 was used, the effort policy setting of *speed* was used for accelerator synthesis, and the placement and routing effort to *high*.

6.4.2 Software Setup

For previous implementations, each individual benchmark required modifications to insert *startup* code, defining static input data arrays, retrieving execution times and outputting results for functional verification. Instead, for the evaluation of this architecture, a software harness was developed to streamline the process of retrieving performance results and to increase test flexibility. To understand the execution environment of the tested kernels, the harness is explained in this section.

The harness can be compiled for a desktop machine or for the MicroBlaze processor. It also accepts other compile-time parameters defining whether it uses heap allocation or emulates a heap via a static array. The harness' structure is a significantly modified version of the CoreMark's benchmark for the *Livermore Loops* [EEM15]. The software structure of the harness expects a single source code file containing all kernels, where each is contained in a single function call. This compilation flow is summarized in Fig. 6.9. Also, the harness can also be compiled for compatibility with a VLIW simulation tool suite (detailed in Section 6.6) and ModelSim simulation.

The desktop version accepts call-time parameters that determine: which kernels to call, the amount of data to process (N) and the number of times to repeat the kernel code (L). Also, the input data to be processed is generated at runtime by a pseudo-random generator and placed into arrays allocated onto the heap. The generator seed is also a variable parameter. The purpose of executing the harness on a desktop machine is so that reference data is generated. This reference data is compiled into the harness when targeting the MicroBlaze, so that the functional correctness of the generated accelerators can be verified when executing on the target board. The reference data also includes the used call parameters, so that the embedded versions calls the same kernels under the same conditions. Keeping the kernel functions in a single separate file ensures that no optimizations are made across function boundaries based on the given N and L values (which are statically defined when compiling for an embedded environment).

At startup, the harness creates a bank of random integers and a bank of random single-precision numbers according to the seed. It then calls one or more kernels based on either the call parameters or static compile values. As an example, Listing 6.1 shows the *inner_product* kernel from the *Livermore Loops* adapted to the harness (lines 20 and 21). For each kernel to execute, the harness first calls an initialization function to allocate input data. In this case two arrays are allocated and initialized with the pseudo-random data generated during startup. Then the execution time of each kernel is measured, by enabling the custom timer/counters prior to the function call. To minimize the effect of the call itself on the measurement accuracy, the outermost loop is repeated L times.

To prevent the compiler from optimizing the outer loop (at line 19) away and preserving only one iteration of the inner loop, array z maintains a dependency with the outer loop iterator. This same approach is used throughout the release of the *Livermore Loops*. Some cases are slightly modified according to need and the same technique is applied when adapting the integer kernels

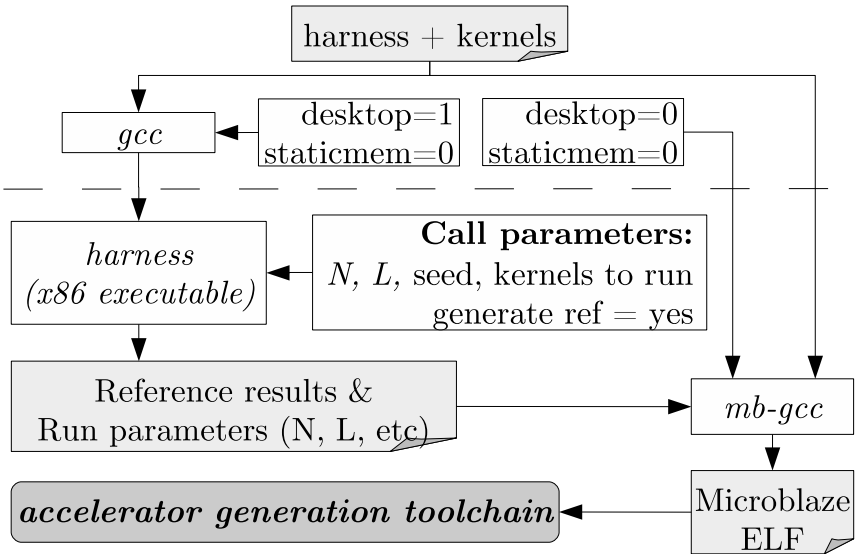


Figure 6.9: Compilation flow of the test harness

to the harness. The use of *-O0* or *-O1* could also be enforced to prevent the optimization. But this would also result in sub-par code for software execution, thereby producing an overly advantageous comparison scenario with the accelerators.

Listing 6.1: *Inner product* kernel adapted for test harness integration

```
1  void inner_product_init(test_params *p) {
2
3          // allocate 2 vectors in vector array "iv"
4          p->iv[0] = (int *) calloc((p->N), sizeof(int));
5          p->iv[1] = (int *) calloc((p->N + p->L), sizeof(int));
6
7          // fill vector v[0] and v[1] with random data
8          reinit_ivec(p, p->iv[0], p->N, 0xffffffff);
9          reinit_ivec(p, p->iv[1], (p->N + p->L), 0xffffffff);
10         return;
11 }
12
13 float inner_product(test_params *p) {
14
15         int q = 0, *x = p->iv[0],  *z = p->iv[1];
16         int l, k, n = p->N, loop = p->L;
17
18         // Kernel A -- inner product (integer)
19         for(l = 1; l <= loop; l++)
20                 for (k = 0; k < n; k++)
21                         q += z[k+l]*x[k];
22
23         return (float) q;
24 }
25
26 void inner_product_fini(test_params *p) {
27
28         int i = 0;
29         for (i = 0; i < 2; i++)
30                 free(p->iv[i]);
31         return;
32 }
```

Finally, the harness reads the execution time (and other information) from the loop accelerator after the call returns. The allocated arrays are then freed and the next kernel is called. The desktop

version of the harness outputs one reference value per kernel (by generating a checksum from all the produced data). The embedded version outputs the measured execution times of the kernels (running on the accelerator or MicroBlaze), and a comparison with the reference data.

In summary, the harness allows for: easily testing multi-loop accelerators by choosing which combination of kernels to call, testing the effects of overhead by varying the amount of data to process, testing the functional correctness of the accelerators by varying the input seed, and therefore the input data. However, it is not geared towards testing kernels which require structured data.

For this evaluation, a single *C* file contains all used kernel functions, from the *Livermore* and IMGLIB, where each is enclosed within a function call in the explained manner. The loop traces from each kernel were extracted by running MicroBlaze versions of the entire harness through the Megablock extractor. The compiler used was *mb-gcc 4.6.4* and compilation flags enabled the use of floating-point, integer multiplication and division, barrel-shift, and comparison operations.

6.4.3 Performance vs. MicroBlaze Processor

Table 6.1 summarizes the characteristics of the accelerators and the corresponding kernel speedups: the results for the floating-point set are on the top half, and the ones for the integer set on the bottom half. The last two integer kernels (*innerprod* and *matmul*) are not from IMGLIB, but are direct conversions of the corresponding floating-point versions. The averages shown are geometric for speedup and arithmetic for the remaining metrics. The third column shows the II of the schedules. The average II is given for the cases where multiple traces were accelerated. The fourth column shows the average executed IPC, which measures the parallelism exploited by the accelerator.

The next two columns show the number of FUs and the number of registers in the register pool. They are indicative of the complexity of the accelerator. The last two columns show the speedups obtained when executing each kernel $L = 1000$ times for two different amounts of data (specified by the parameter N). All speedup results include the overhead of passing data from the MicroBlaze processor to the accelerator and retrieving the results. The benchmarks *f5*, *f6* and *f11* where run only for $N = 1024$, because the system does not have sufficient on-chip memory to support the $N = 4096$ case. For instance, *f5* requires 15 arrays of N floating-point numbers. All systems ran at 110 MHz, both for software-only and accelerator-enabled runs.

As is noted in Table 6.1, only one candidate Megablock was extracted for each kernel, save for the cases of *f5*, *f9* and *f11*. The average number of instructions in each trace was 34 when considering all kernels. Considering the sets separately, the floating point set has on average less instructions than the integer set: 26 versus 45. However, the integer average is inflated by the cases of *i8* and *i9*, whose accelerated Megablocks contain the most instructions out of all those implemented, 142 for both. For the traces in the floating point set, an average of 7.2 instructions are floating-point instructions, which represents 27.6 % of total trace instructions.

Taking into account the number of instructions in each implemented loop over the number of clock cycles that the MicroBlaze requires to execute it, we find the MicroBlaze executes the equivalent of 0.38 IPC for the floating-point set and 0.67 for the integer set. As for the IPC achieved on the accelerator, $\mathrm{IPC_{HW}}$, it is computed as the number of instructions in a Megablock

Table 6.1: Generated accelerator characteristics and achieved speedups

ID	Kernel	II	IPC_{HW}	# FUs	# RP Regs.	Speedup	
						$N = 1204$	$N = 4096$
f1	cholesky	3.0	6.67	11	30	7.25	9.80
f2	diffpredict	10.0	4.40	10	52	6.78	7.86
f3	glinrecurrence	3.0	4.33	9	20	3.65	4.58
f4	hydro	3.0	5.67	9	28	12.47	12.84
f5	hydro2d[1]	16.3	3.68	12	71	6.60	(6.60)*
f6	hydro2dimp	6.0	6.17	9	38	11.18	(11.18)*
f7	innerprod (fp)	4.0	2.50	6	9	4.43	4.48
f8	intpredict	10.0	4.10	9	54	7.99	9.46
f9	linrec[2]	11.0	0.95	9	12	2.30	2.31
f10	matmul (fp)	3.0	5.00	10	25	4.84	7.06
f11	pic1d[1]	5.5	3.73	12	30	1.47	(1.47)*
f12	statefrag	5.0	8.40	13	49	18.47	18.98
f13	tridiag	3.0	4.00	8	21	6.81	6.93
	mean	6.4	4.58	9.8	33.8	5.96	6.60
i1	quantize	3.0	3.33	7	15	3.22	3.99
i2	conv3x3	10.0	6.40	12	39	6.75	7.01
i3	perimeter	3.0	7.00	13	34	7.35	7.81
i4	boundary	4.0	6.00	12	25	2.92	3.59
i5	sad16x16	2.0	6.50	9	13	2.31	2.31
i6	mad16x16	2.0	6.50	9	13	2.30	2.30
i7	sobel	5.0	8.00	14	46	7.99	8.14
i8	dilate	20.0	7.10	13	77	5.31	5.23
i9	erode	22.0	6.45	14	72	5.43	5.24
i10	innerprod (int)	3.0	3.33	6	8	3.93	3.98
i11	matmul (int)	3.0	5.00	8	23	3.82	5.44
	mean	7.0	5.97	10.6	33.2	4.27	4.61
	total mean	6.7	5.22	10.2	33.5	6.07	5.60

[1]Three loops accelerated; [2]Two loops accelerated. For all other cases, one loop was accelerated. *values taken from run with $N = 1024$

over the II of the respective schedule. Table 6.1 shows, despite the similar average IPC_{HW} values for both sets, that the average speedup for the integer set is lower than that of the floating-point set. This is expected, since the IPC_{SW} for loops with floating-point instructions is lower.

As an example, consider the *matmul* kernel, which contains 15 binary instructions for both integer and floating-point cases. One iteration of the integer kernel requires 25 clock cycles on the MicroBlaze, but the floating-point version requires 33 cycles. However, both benchmarks execute on accelerators with the same II, resulting in better speedup for the floating-point version. Execution on the accelerator effectively mitigates the latency of floating-point operations due to

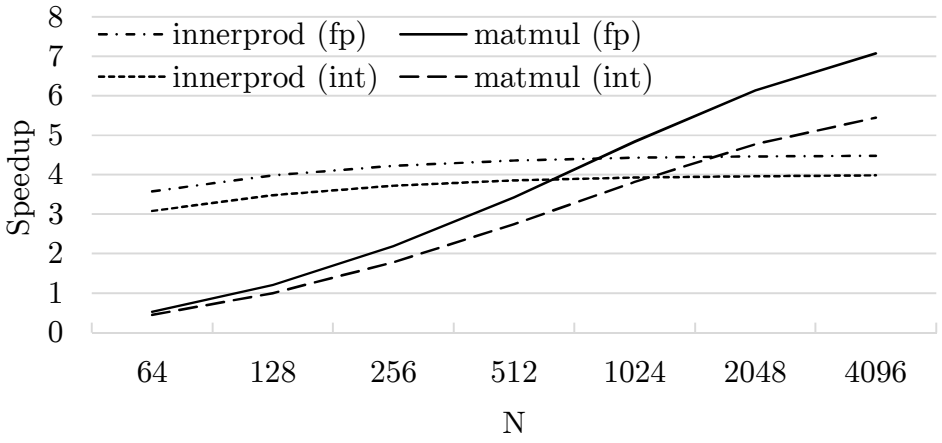


Figure 6.10: Speedup as a function of input/output data array sizes for integer and floating-point versions of *innerprod* and *matmul*

the availability of fully pipelined floating-point units and the use of an efficient static schedule. Consider as another example *i7* and *f12*, whose Megablocks contain 40 and 42 instructions, respectively. They were scheduled at equal IIs and therefore their IPC_{HW} is very similar. The difference in speedup is due to the 16 floating-point operations in the accelerated Megablock for *f12*. As a result, one iteration in software requires 139 clock cycles, versus the 54 cycles for *i7*.

As with all other implementations, the speedup is also affected by the communication overhead between the MicroBlaze and the accelerator. This impact can be seen when comparing the mean geometric speedup for the two values of N. For some benchmarks, (e.g., *i7* and *i4*), the speedup increases with N, indicating that the constant invocation overhead is amortized over a larger number of iterations. For other benchmarks (e.g., *f9, f12, f13, i5, i6*) the effect of increasing the value of N is negligible, meaning that the communication time was already small compared to the processing time (*f9, f12, f13*) or that the communication time also scales with N (*i5, i6*).

To illustrate the first case consider the two versions of *innerprod* and *matmul*. The corresponding speedups for several values of N are shown in Fig. 6.10. The speedup for the two versions of *innerprod* increases slowly and then stabilizes, showing that the communication overhead is negligible for $N > 256$. The speedup for the two versions of *matmul* increases over the entire range of N. In this case the impact of accelerator invocation is still noticeable for large N. The speedup gap between the two version increases too, again showing that, in comparison with the MicroBlaze processor, the accelerator supports floating-point operations more efficiently.

The *i5* and *i6* benchmarks demonstrate that, in some cases, overhead that cannot be amortized, For both these cases, the accelerated Megablock corresponds to a inner loops which always iterates 16 times per call. Adjusting the N parameter determines the outer loop bounds alone. One way to address this would have been to unroll the inner loop, which for such a small number of iterations (and especially given the resource efficiency of this accelerator design) would have been a viable options. Otherwise, the overhead scales alongside the total number of iterations to perform.

The highest overheads occur for *f3*, *i5* and *i6*. The overheads are 17 % for the former case and

51 % for the latter two (for $N = 4096$). For *f3*, each accelerator call performs an average of 63 iterations, and for *i5* and *i6* only 16 iterations are executed. In contrast, the average overhead for all benchmarks is 8 %, and the average number of iterations per call is approximately 1433.

Dual-Clock Domain Estimation The accelerator synthesis frequency is often higher than the MicroBlaze operating frequency. It would be possible to further increase speedups by relying on two clock-domains. To estimate speedups for such a scenario, the systems for all benchmarks were re-generated but now instructing the synthesis tools to maximize frequency.

According to the timing reports, all the critical paths were due to internal MicroBlaze logic, or due to paths between the MicroBlaze and memory controllers. Note that, although the MicroBlaze instances used in these experiments target *speed*, the extended floating-point unit has been enabled, as well as the barrel shifter, comparison and integer multiplier units. As a result, the average operating frequency given by the timing reports is of approximately 150 MHz, which is below the average synthesis frequencies of the accelerators, 221 MHz.

Given this, a speedup can be estimated for a scenario where the MicroBlaze operates at 150 MHz and the accelerator operates at its reported synthesis frequency, on a per-case basis. The resulting geometric mean speedup is of 6.98× for the floating-point set, and of 7.72× for the integer set. The increase is marginal for the floating-point set given the small difference between the average synthesis frequency of the accelerators relative to 150 MHz.

Note that the dual clock domain scenario would require another design iteration on the BRAM sharing mechanism, to allow for the memory ports to be fed by two different clocks. If the clocks are chosen to be integer multiples, then a clock gating method could be used.

6.4.4 Resource Requirements & Operating Frequency

The approach discussed here favours performance, because it does not impose resource limitations on the architecture beyond the restriction to two memory ports. This may come at the cost of resource requirements due to boundless instantiation of FUs and the associated increase in size of the configuration words required to control the accelerator. Figure 6.11 shows the FPGA resources used by each system (after placement and routing). The stacked bars represent the resources used by each accelerator (upper half) and by the remaining system components (lower half). The line represents the clock frequency of the accelerator as given by the synthesis reports.

In order to measure the accelerator resource requirements, consider a MicroBlaze processor (without caches and with an FPU), which requires 2291 LUTs and 1503 FFs. The average accelerator requires 1.13× more LUTs and 1.83× more FFs. Additionally, the number of slices required by the accelerator can serve as a measurement of required area. It requires 1.12× the number of slices that the MicroBlaze requires, which is 860. As a general rule, the accelerators for the integer loops require less resources and also achieve higher operating frequencies relative to the floating-point cases. The average number of FUs required for the floating-point and integer sets is 9.8 and 10.6, respectively. The average size of the register pool is also similar, 33.5 registers.

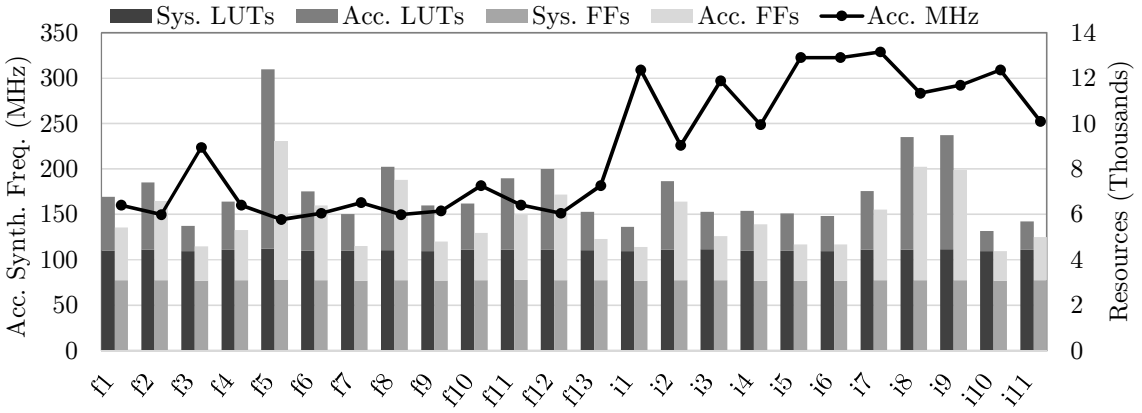


Figure 6.11: Resource requirements for the entire generated systems and accelerators, and accelerator clock frequency

The increased complexity of the floating-point units justifies the lower synthesis clock frequency for the floating-point set, whose average is 164 MHz, which is significantly lower than the average clock frequency of 290 MHz for the integer-only accelerators. In all cases, the critical path in the accelerators with floating-point operations includes the *fadd* FU; the sole exception is benchmark *fp3*, where the critical path is determined by the *fmul* unit (since there are no floating-point additions). Floating-point FUs make up 24 % of all FUs per accelerator. For the integer set, the drops in frequency seen in *i2*, *i4* and *i11* are due to critical paths between the instruction memory, multiplexers and integer multiplication FU. System *i10* is the only other with an integer multiplier, but does not suffer from the same drop in frequency because the inputs of the multiplication FU receive operands only from one other FU each and therefore have no input multiplexer.

There is no clear relation between accelerator size and maximum operating frequency for the implementations shown in Fig. 6.11. Because loops are scheduled for their minimum IIs, more FUs are instantiated, each of which has fewer operations scheduled to it. This leads to more customized connectivity. Increasing the II and thereby instantiating less FUs would require more complex input multiplexers. This is shown in the next section, where accelerators for multiple loops of similar size suffer small decreases in frequency due to this effect.

On a final note, the synthesis frequency for every accelerator is above the 110 MHz clock frequency used for the implementations. In many cases, using a higher frequency might be possible: this would reduce the absolute time taken by the benchmarks, but would not change the speedup values for a system with a single clock domain. The speedups presented in this section are due to the combined effects of the scheduling method and the accelerator architecture, which minimize the II and increases the IPC_{HW}.

6.4.5 Power and Energy Consumption

The power and energy consumption for the single-row accelerator architecture were also determined for a subset of the benchmarks, via actual on-board measurements of power, using Texas

Table 6.2: Power and energy consumption for software-only and accelerated runs

	Accelerator-enabled Execution			Software-only Execution		
ID	Power (W)	Runtime (s)	Energy (J)	Power (W)	Runtime (s)	Energy (J)
f1	0.595	2.02	1.20	0.452	17.96	8.11
f2	0.592	8.00	4.74		58.19	26.29
f3	0.514	9.58	4.93		35.15	15.88
f4	0.561	2.50	1.40		34.03	15.37
f5	0.562	35.47	19.93		235.47	106.37
f6	0.627	5.19	3.26		62.11	28.06
f8	0.635	8.78	5.58		74.46	33.64
f9	0.500	51.32	25.63		77.10	34.83
f12	0.653	4.61	3.01		90.95	41.09
f13	0.545	2.68	1.46		21.12	9.54
mean	0.578	13.02	7.11	-	70.65	31.92

Instruments' Fusion Digital Power Designer [Ins16]. Each kernel was executed with $N = 1024$ and $L = 10000$. Using a high value of L increases the runtime, thereby decreasing the measurement noise since more data points are captured.

The power and energy measurements are shown in Table 6.2, and are the result of monitoring the 1 V-rail, which powers the FPGA. On the left-hand side, the table contains measurements relative to a system containing the respective accelerator instance, and on the right-hand side the measurements are relative to a system containing only local memories, a MicroBlaze instance, and other required modules. For the former case, the average power consumption was 0.57 W, and for the latter the average was 0.45 W. However, due to reduced runtimes, the total energy consumed is lower for the accelerator-based systems, which require an average of 7.11 J versus 31.92 J for the software-only system.

6.4.6 Performance and Cost of Multi-loop Support

Multi-loop accelerators can enhance the execution of several loops (from the same application or from a set of related applications) and also result in resource savings when compared to a group of individual single-loop accelerators. In the results analyzed in the previous sections, there are already two systems whose accelerators support more than one loop (*f5*, *f9* and *f11*). To further test multi-loop support, seven groups of kernels were selected and and accelerator was generated to target each group. Groups of kernels were chosen based on the type of calculations performed as well as number of instructions (e.g., generating a single accelerator for *dilate* and *erode*).

Table 6.3 presents speedups and accelerator resource requirements for these cases. In this table, the speedups shown are not the average of the speedups obtained when executing each individual kernel. Since the II of each scheduled CDFG remains the same as in the single-loop case, so does the individual speedup. Instead, a single speedup was computed using the total execution time of all loop kernel calls. That is, the values shown are equivalent to a weighted

Table 6.3: Generated multi-loop accelerator characteristics and speedups

ID	# Megablocks	# FUs	# RP Regs	MRT Occ. (%)	Speedup	
					N=1024	N=4096
f_2_8	2	11	96	48	7.42	8.70
f_3_9	3	11	25	21	2.73	2.94
f_4_5_6	5	13	119	41	7.54	7.54
f_7_10	2	10	28	40	4.83	6.96
i_5_6	2	8	24	68	1.65	2.31
i_8_9	2	18	99	35	5.37	5.82
i_10_11	2	9	13	53	3.82	5.40
mean	2.58	11.4	57.7	44	4.25	5.16

This table refers to systems with accelerators for loops from several kernels. The benchmark names indicate which kernels are supported.

average of the individual speedups: of all the kernels accelerated per system, the one with the greatest execution time will have the greatest impact on the resulting overall speedup of the system.

The resource requirements for the multi-loop accelerators are compared to the sum of resources of individual accelerators in Fig. 6.12. Each accelerator requires only an average of 72 % and 59 % of the sum of LUTs and FFs of the individual accelerators, respectively. If single-loop accelerators were used, the accelerator with the worst critical path would determine the system's operating frequency. For the multi-loop cases it is expected that the frequency would decrease due to increased accelerator size and complexity. Despite this, the average synthesis frequency of the multi-loop cases is only 2.5 % lower than the worse case of the respective single-loop accelerators.

As an example of the frequency decrease due to increased connectivity consider system *i_8_9*, which supports the same loops as *i8* and *i9*. The synthesis frequency is similar for both of these cases, and is determined by a critical path between the instruction memory, a barrel-shift FU and the pool registers. However, for *i_8_9*, the input multiplexers of two *add* FUs are considerably more complex, than those in the individual cases, for the equivalent units. For instance, the input multiplexer for the first operand of one of the units in *i_8_9* has twice as many possible inputs. As a result, the decrease in frequency for *i_8_9* is the highest, for the tested accelerators in Table 6.3.

Another metric of how well the resources are shared is the occupation of the MRT, which measures how efficiently the schedule uses the available resources. For all the cases in Table 6.1, average occupation of the MRT is 53 %. For the systems in Table 6.3 the average is 44 %. This is expected because accelerators with multiple configurations may have FUs that are used only by a subset of them, which decreases the occupation. The occupancy of the MRT for a set of loops could be a target metric while scheduling, to balance the number of FU and the II.

6.5 Performance Comparison with ALU Based Accelerators

Many existing approaches frequently make use of architectures which allow for some programmability. For instance, mesh arrays of multiple-function FUs with rich interconnects (such as nearest

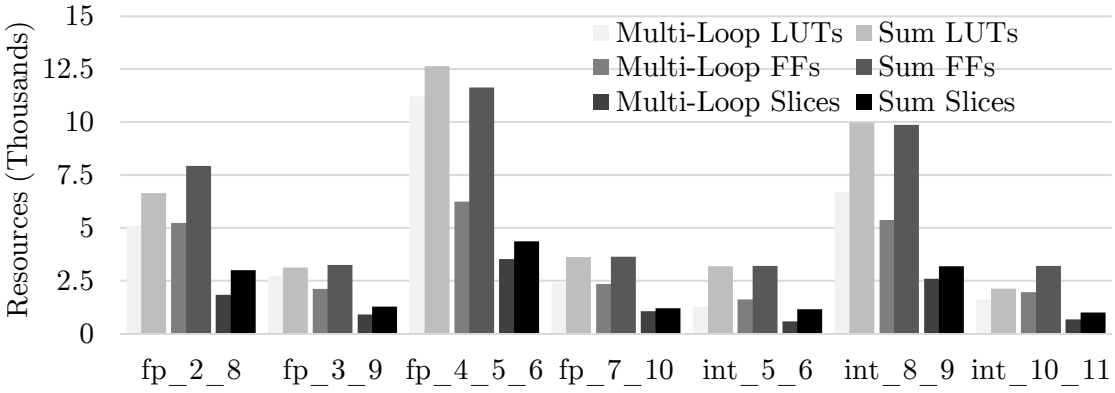


Figure 6.12: Resource requirements for multi-loop accelerators vs. sum of resources for individual loop accelerators

Table 6.4: Accelerator generation scenarios

Scenario	Description
a1	Allocation of any number of resources of any type
a2	Allocation of any number of ALUs (+ other units)
b1	2 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)
b2	4 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)
b3	8 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)

neighbour connections aided by less numerous longer connections to distant units). Architectures such as these are usually designed once, possibly by a quantitative approach, to be flexible enough so that future sub-graphs can be successfully mapped and executed. That is, these designs must be rich enough in terms of resources and especially interconnection capability in order to increase applicability. On the other hand, this may incur considerable resource costs.

The results presented so far are relative to fully customized accelerators, that is, instances with any number of operation-specific FUs. The objective of generating fully customized designs is to maximize performance and to decrease resource usage. In this section the advantages of this customization are evaluated by comparing fully customized accelerators with instances containing a fixed number of ALUs, to establish a comparison with existing static resource accelerators.

Table 6.4 summarizes the five types of accelerators generated for this experiment using the developed scheduler. Scenario *a1* was presented in the previous section: fully customized generation of accelerators with boundless resource allocation. Scenarios *b1*, *b2* and *b3* contain a fixed number of resources. To do this, the scheduler was tuned so that scheduling starts with the given units, and so that allocation of more FUs is not performed. Instead, as per typical modulo-scheduling approaches, the II is increased until a valid schedule is possible. An additional scenario, *a2*, uses boundless allocation of ALUs.

Comparison with fixed resource scenarios The employed ALU essentially contain instances of each type of individual FU, with the required additional control logic. Supported operations

include all integer arithmetic (except division and multiplication), logic, and comparison operations. In order for the synthesis tools to not optimize the ALU logic on a per-instance basis (due to constant propagation of the instructions feeding each ALU) a black-box instance was synthesised. This means that each ALU represents a fixed cost of approximately 640 LUTs and no FFs.

In addition to the ALU, these cases also use a single branch unit (which evaluates all types of exit conditions) and a single integer multiplier, also instantiated as black boxes. For the benchmarks of the *Livermore* set, a single Floating Point Unit (FPU) is also added, which includes all floating-point arithmetic, comparison and float/integer conversion. The ALU has a latency of 1 clock cycle and the latency of the FPU varies according too the issued operation, but it is still possible to pipeline operations. Like the ALU, the FPU is constructed from one instance of each type of floating-point unit, and a black-box was used to instantiate it. The cost of the FPU is 1460 LUTs and 525 FFs. Note that although the number of units is fixed, the interconnections between them are still specialized by the scheduler based on dataflow between scheduled operations.

Figure 6.13 shows the speedups for these cases. The speedup in scenario *b3* equals that of scenario *a1* for all integer cases. In other words, the Megablocks that can be detected for these benchmarks can be executed at the minimum possible II with 8 ALUs, and in most cases only 4 ALUs suffice. Inversely, we can state that fully customized accelerators perform equivalently to generalized accelerators with 4 or 8 ALUs. There is only a noticeable difference between *b1* and *b2*, where the average IIs are 19.6 and 10.3, respectively. Regardless, for the integer cases, an accelerator with only 2 ALUs still achieves a mean geometric speedup of 2.08×.

However, since the accelerators in these scenarios contain only 1 FPU, this means the speedup decreases for loops with floating-point operations. For 6 out of the 13 floating-point benchmarks, the speedup decreases to approximately half on average, whilst the remaining decrease marginally. The average II of the former 6 cases increases to 17.2 (for *b1*, *b2* and *b3*), versus the average II of 7.2 for scenario *a1*. The highest decrease in speedup occurs for *f12*, whose accelerated loop contains 16 floating-point operations, scheduled onto 4 floating-point units for the accelerator in scenario *a1*. With only one FPU, the speedup decreases by approximately four times, from 18.9× to 4.8× (regardless of the number of ALUs). Although the accelerator for *f5* in scenario *a1* contains 5 floating-point units, the speedup only decreases by half in the remaining scenarios. This is because three loops are accelerated, only one of which uses all 5 floating-point units frequently.

This can be analysed in terms of resources in this way: in order to schedule the benchmark loops with minimum II, an average of 2.3 floating point FU are instantiated for the accelerators of scenario *a1*. This means that, for a fixed resource accelerator, at least 2 FPUs would be required to prevent increasing the II. In other words, a fixed resource accelerator with 4 ALU and 2 FPU would incur a cost of 7707 LUTs and 3697 FFs. In comparison, the accelerators for the floating point set in scenario *a1* require an average of 2829 LUTs and 2863 FFs.

As a summary, Fig. 6.14 shows the average required resources for all scenarios, distinguishing between floating-point and integer sets. The values are normalized to the resource requirements of one MicroBlaze. Considering that the number of slices represents area on the device, the accelerators in scenario *a1* are roughly 0.62×, 0.47× and 0.32× smaller than those of cases *b1*, *b2* and

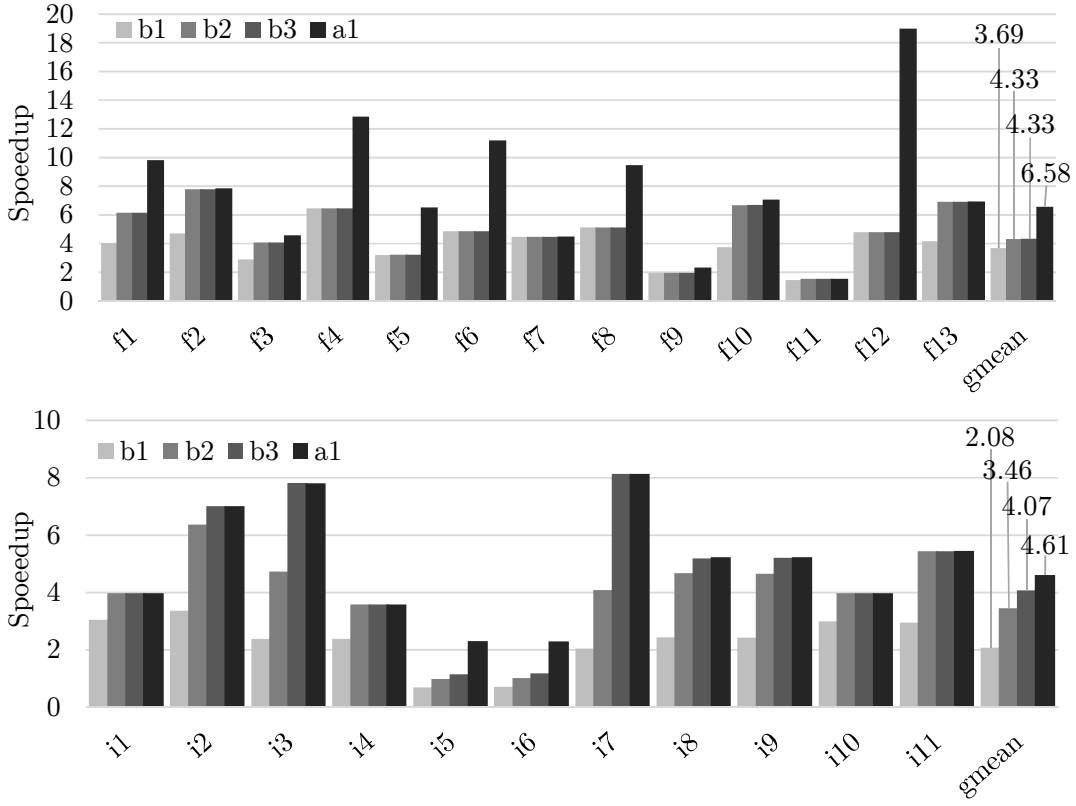


Figure 6.13: Speedups for several types of accelerators vs. a single MicroBlaze processor

b3, respectively. The number of required FFs varies little, since the same amount of data needs to be transported between FU regardless of their type. For the floating-point cases, the reduction in the number of LUTs is higher due to the higher cost of each FPU.

In short, a specialized accelerator performs on par with a fixed-resource accelerator and when floating-point support is required it allows for better exploitation of ILP versus deploying a single fully fledged FPU, and is less costly in terms of resources versus employing two FPUs.

The configuration word memory size also varies between scenarios due to the dependence of the word width on each particular instance. For the sake of brevity, consider only that the average memory size is 8.73 kB, 15.88 kB, 12.48 kB, and 17.30 kB for scenarios *a1*, *b1*, *b2* and *b3*. The size of the code word memory of the accelerators depends on three factors. The first is the number and type of computational resources. Specialized FUs receive at most one or two configuration bits per schedule step. The ALUs, FPUs and branch unit on the other hand require 18, 13 and 8 bits. Secondly, the connections between units affect the multiplexer complexity and therefore the number of bits in a configuration word required to control them. Lastly, a larger number of loop operations (i.e., processor instructions) to schedule generally means longer schedule lengths.

Consider then that scenario *a1* instantiates more units, but each with less control bits, and that more units allows for shorter schedule lengths and therefore fewer total configuration words (approximately half relative to the remaining three cases). For the fixed-resource cases, neither the number of configuration words nor the word width vary in a predictable fashion with the of number

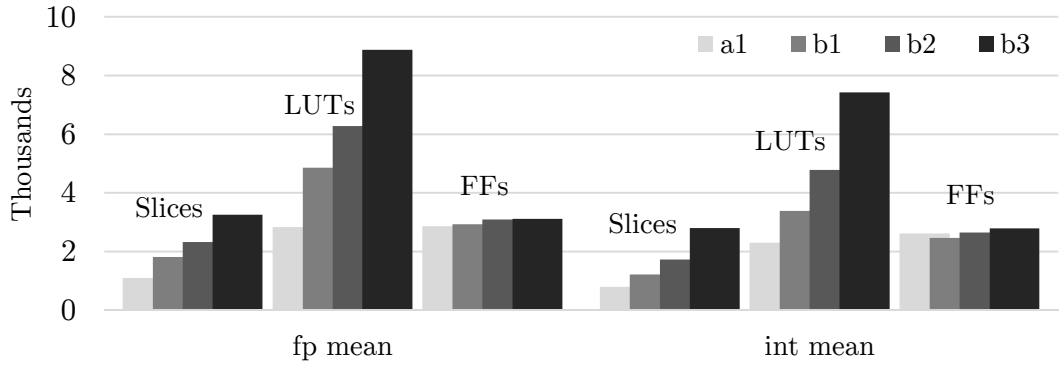


Figure 6.14: Average accelerator resource requirements for the several scenarios

Table 6.5: Average cost of accelerators per scenario, normalized by the cost of a single MicroBlaze

	floating-point			integer		
Scenario	LUTs	FFs	Slices	LUTs	FFs	Slices
a1	1.23	1.90	1.28	1.01	1.74	0.93
b1	2.12	1.95	2.12	1.48	1.64	1.42
b2	2.74	2.06	2.70	2.09	1.76	2.01
b3	3.87	2.08	3.79	3.24	1.85	3.26

of ALUs. Determining the reasons for these particular widths and lengths requires analysing the individual assignment of operations to ALUs and the resulting multiplexer widths. This analysis has not currently been performed but note that the average configuration memory size for scenario *a1* is 70 % that of the smallest memory size for the remaining cases, *b2*.

Comparison with ALU allocation In an additional scenario, *a2*, accelerators were generated by instantiating any number of ALUs required to schedule the loops at minimum IIs. In other words, this scenario is equivalent to the full-custom scenario, but resorting to ALUs instead of custom FUs. This determines how many ALUs, and therefore resources, a generalized loop accelerator would require to achieve the same performance as a specialized instance.

The resulting average number of instantiated ALUs is 3.7 for the floating-point set and 5.0 for the integer set. For this scenario, an accelerator requires an average of 2165 slices and 5766 LUT, which is an additional average of 1210 slices and 3189 LUTs relative to scenario *a1*. In other words, these accelerators require approximately 2.49× the slices, 2.55× the LUTs and 1.10× the FFs relative to a specialized instance (while still benefiting from customized connectivity) in order to achieve the same execution performance.

6.6 Performance Comparison with VLIW Architectures

The MicroBlaze processor is a single-issue processor and therefore it is not surprising that considerable speedups can be attained by exploring, even if only partially, any latent ILP or other

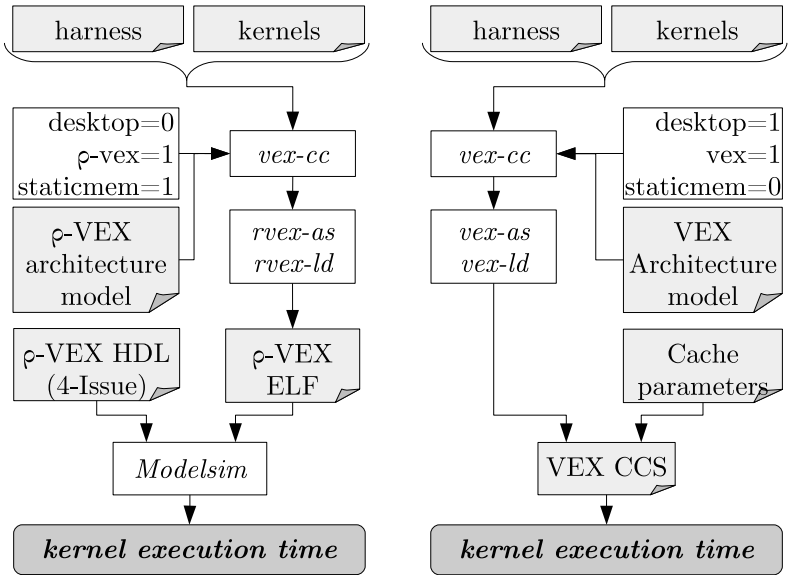


Figure 6.15: Simulation flow for ρ-VEX processor and other VEX architecture models

hardware specializations (e.g., specialized units). Therefore, to have a comparison with VLIW architectures, this section contains performance results obtained by using Hewlett-Packard's VEX toolsuite [FFY05] as well as the ρ-VEX processor implementation [SABW12].

The VEX toolsuite is a compilation chain which targets a user-specified VLIW architecture. Architecture parameters include issue width, and the number, type and latency of units. These parameters are given at compile-time to generate a Compiled Code Simulator (CCS) which runs on a desktop host machine. The ρ-VEX architecture is a synthesis-time parametrizable VLIW processor which implements the VEX instruction set. The ρ-VEX release used was version *3.3*, which includes the open-source ρ-VEX processor (written in VHDL) itself and also a build of the *gcc* tool chain targeting the architecture. Additionally, it also possible to use HP's VEX compiler for the compilation step, and use ρ-VEX's tool chain for the assembly and linking steps.

Using these tools the following experiments were conducted: 1) execution on three VEX CCSs, each using a different architecture model and 2) simulated execution of a 4-issue version ρ-VEX processor in ModelSim. Since the VEX architecture does not support floating-point natively, all comparisons in this section are limited to integer benchmarks.

Figure 6.15 shows the compilation flow used to generate the mentioned test cases. The test harness was compiled with HP's VEX compiler, targeting three different architecture models (right-hand side). The software harness was compiled once per model, generating three VEX CCSs with all the capabilities of the harness (as explained in Section 6.4.2). Each kernel was executed on each CCS, to determine the execution time of the entire kernel function call. For simulation of the ρ-VEX processor, one ELF is compiled per kernel, and then loaded into the processor and executed (left-hand side). As explained previously, the harness can be compiled to receive no call-time parameters and instead read statically defined parameters and reference data. Since we would need to compile the standard *C* library for the ρ-VEX target in order to use dynamic mem-

Table 6.6: VEX simulator and accelerator models comparison

Parameter	r1	r2	r3	a1	b2
Issue width	2	4	8	*N/A*	
#ALUs	2	4	8	Variable	4
#MULs	2 (16x32)			Variable	1 (32x32)
#Registers	64			Variable	
Memory access	1 Load/Store Unit			2 Memory Ports	
ALU Latency	1			1	
MUL Latency	2			2	
Load Latency	3			2	
Store Latency	1			1	
Memory/Cache	Simulated without cache			Local BRAM memory	

ory allocation, the harness was adapted so that only statically allocated memory is used. This does not impact the execution time of the kernels, since their source code is not modified and only the kernel function call is measured. When measuring execution time on the ρ-VEX the cache access stall cycles were not counted (emulating a local memory performance for fairness of comparison).

Table 6.6 shows the model parameters used to compile each VEX CCS. In order to ease the comparison between these cases, the parameters for *r1*, *r2* and *r3* are equal to the parameters used when compiling for the ρ-VEX processor (in terms of number of registers, delays of operations, etc). That is, the CCSs approximately simulate three variants of the ρ-VEX with different issue widths and number of ALUs. These cases are compared with two of the implementation scenarios presented in the previous section: *a1* (customized accelerators) and *b2* (4-ALU accelerators).

As a side note, consider that although presented here as a comparison, using VLIWs for acceleration and deploying binary accelerators, are not mutually exclusive approaches to increasing performance in embedded systems. The augmentation of a smaller 2-issue VLIW with a customized loop accelerator is not infeasible. In such a scenario, the VLIW would provide a moderate performance increase throughout most of the code, exploiting the relatively low but relevant ILPs, whilst the accelerator would target highly pipelinable loops.

6.6.1 Performance Comparison

Deploying VLIWs to explore ILP involves two steps: design of a multiple issue processor and design of a sophisticated compiler. Exploring ILP is therefore a compile-time task, performed statically on high-level source code. The processor can be simpler, since it expects statically scheduled code, and using a compiler allows for exploration of powerful optimization steps. However, not all the code in an application contains ILP to exploit. This means an underutilization of VLIW hardware, and instructions populated with *nops*, which still represent a cost in terms of memory size. Even if a portion of code is highly optimized for the available issue width, there is no guarantee it will execute frequently, which exacerbates the resource cost of deploying a VLIW.

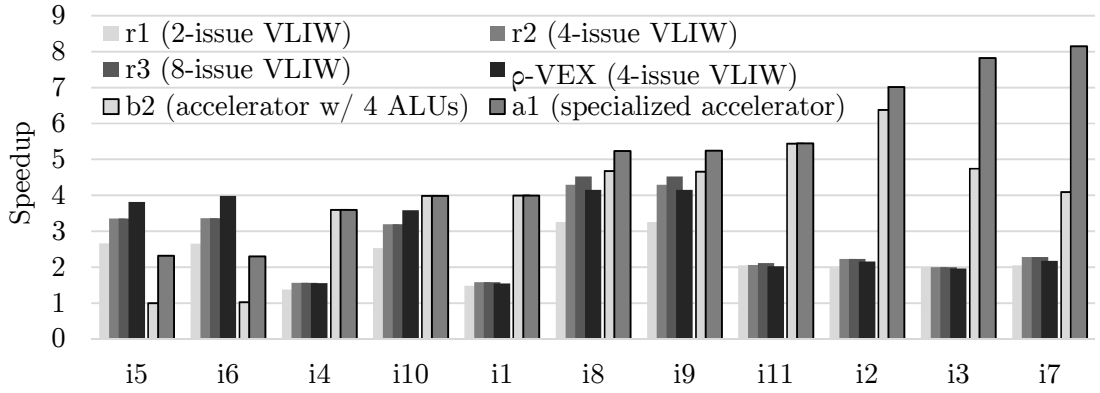


Figure 6.16: Speedups for different VLIW models and two of the presented accelerator scenarios

In contrast, the approach presented in this work relies on post-compilation information. Only frequently executing code is migrated to hardware, and acceleration is achieved by exploiting both intra-iteration and inter-iteration ILP of loop path traces. Although new tools are required to support the present approach, none are as complex as an architecture-specific compiler and also allow for the application binary to remain compatible with a non-accelerated system.

The speedups for the tested integer kernels are shown in Fig. 6.16. Speedups were calculated considering the same operating frequency for all cases and for the baseline. The geometric mean speedups over a MicroBlaze processor for *r1*, *r2*, *r3* and the ρ-VEX execution is 2.22×, 2.58×, 2.61×, and 2.63×, respectively. Considering the geometric mean speedup for *a1*, 4.61×, this means that fully customized accelerators are 1.79× faster than a 4-issue VLIW processor.

Figure 6.16 shows the speedups sorted according to the speedups for *a1*. Three groups of kernels are noticeable: 1) for *i4*, *i1*, *i11*, *i2*, *i3* and *i7*, the accelerators outperform VLIW execution; 2) for *i10*, *i8* and *i9*, the performance is roughly equivalent; 3) for *i5* and *i6*, the VLIW approaches are more efficient. The accelerators effectively execute MicroBlaze instructions. This means a direct correlation between the number of executed IPC and the speedup. However, several factors are relevant when comparing the VLIW cases to the accelerators and to the MicroBlaze baseline.

Firstly, due to different instruction sets, the number of instructions in the VLIW kernel loops differ from the MicroBlaze equivalents, which also means the IPC values for each case cannot be directly used as a comparative measure. As an example, consider *i8* and *i9*. For *r2*, the loops for these cases contained 38 VEX instructions. Given the issue width, these instructions contain more than one elementary operation. The CCS for this model reports an IPC of 2.4 which means a total of 97 VEX operations implement the computations of each loop. The respective MicroBlaze loop traces contained 148 instructions, whose execution on the accelerators achieved a similar performance to *r2*, but required executing approximately 6.5 IPC.

Secondly, the VLIW cases may be able to partially overlap loop iterations. If the compiler employs loop unrolling, operations from different iterations may be bundled into the same single-cycle instruction word targeting the entire VLIW width. It is difficult to determine how much of the ILP possible for the VLIWs is due to this, and how much is due to latent intra-iteration

parallelism. It is noticeable, for the cases where VLIW execution is outperformed, that an increase in VLIW issue width does not lead to an increase in performance. This means there is no more ILP to explore, or that the compiler is unable to do so. In contrast, the accelerators rely on heavily exploiting loop pipelining. For the mentioned kernels, the developed modulo scheduler reports that the average number of instructions which belong to the same iteration and are executed in parallel is 1.42. This is comparable to the average IPC of the VLIWs for the same kernels, 1.50. By exploiting the minimum II the IPC while executing on the accelerators is increased to 5.97.

Thirdly, the accelerators performs better when the number of iterations to perform per accelerator call is high, since this mitigates overhead. The present approach targets innermost loops. This means that for a nested loop which represents a large portion of computation, it is desirable that the inner loop represent most or all of it. Alternatively, the inner loop must iterate such a small number of times that unrolling it would not significantly increase the size of the outer loop trace, thus making it a viable candidate for acceleration. This has been previously shown for *i5* and *i6*, where VLIW execution outperforms the accelerators, as the accelerated inner loops iterate only 16 times. Execution on a VLIW suffers no such effects and can apply loop optimization techniques. In fact, for the two mentioned cases the VEX compiler partially unrolls the inner loop.

Dual-Clock Domain Estimation This evaluation considered that both the VLIW processors and the MicroBlaze operated a the same frequency. In Section 6.4.4 a short analysis presented the possible speedups if the accelerators were to operate at their maximum frequencies, while the MicroBlaze operated at its own maximum frequency, 150 MHz. For this evaluation, and for the ρ-VEX processor specifically, this comparison is moot, since its maximum operating frequency is also 150 MHz [SABW12]. In fact, the MicroBlaze is penalized since the maximum of 150 MHz is due to the FPU. Without it, the MicroBlaze can operate at 200 MHz, as the comparison in [SABW12] demonstrates. Consider then this frequency for the MicroBlaze, the maximum frequency of 150 MHz for the ρ-VEX, and the per-case maximum frequency of each accelerator in scenario *a1*. If each component operates at the respective maximum, the geometric mean speedup for the integer set would be 1.97× for the ρ-VEX, and 6.18× for the fully customized accelerators.

6.6.2 Resource Usage Comparison

In terms of resources, the developed accelerators are compared to the reported resources requirements for the ρ-VEX processor in [SABW12]. When targeting a Virtex-6 XC6VLX240T, the 4-issue version of ρ-VEX requires 1046 FFs, 12899 LUTs and 16 BRAMs. A fully custom accelerator instance (i.e., *a1*) requires approximately half as many LUT and 5× as many FFs. The fact that BRAMs are not used for data storage justifies this higher register requirement.

Finally, consider the memory size required by these approaches. By inspecting the ρ-VEX processor assembly, it was determined that the kernel functions required 1.07 kB on average. The average configuration memory size for an accelerator in scenario *a1* was 7.47 kB. Note that the scheduler currently generates configuration words which are unoptimized for size, but are stored in read-only distributed memory which allows for synthesis-time optimizations.

6.7 Concluding Remarks

This chapter presented a design iteration of a automatically generated accelerator design for the purpose of transparent binary acceleration. A single-row model of customizable FUs and connectivity simplifies both the resulting architecture instances and the supporting tools, but requires a sophisticated scheduler. The modulo scheduler creates specific accelerator instances based on a set of CDFG to execute and assures that the speedup attained for each CDFG is maximized for each. The applicability of the accelerator vastly increases due to floating-point support. The only resource restriction are the two memory ports, which still prove to be adequate enough to efficiently exploit any existing data parallelism within the target CDFGs. Relative to the multi-row implementations, this design is also more resource efficient. The proposed architecture achieves a geometric mean speedup of 5.60× over a single MicroBlaze for a set of 24 kernels. Relative to the accelerator implementation in the previous chapter, the average number of LUTs and FFs required by the accelerator, according to synthesis reports, is approximately 0.78× and 0.83×, respectively.

The additional cost of supporting floating-point operations on the accelerator is not very significant since single-operation FUs are employed, versus fully-fledged FPUs. This was demonstrated by the resource requirement comparison between the fully customized accelerators and fixed-resource models based on ALUs and FPUs. The performance obtained versus state-of-the-art VLIW models and generic designs representative of existing accelerator architectures further highlight the efficiency of the design especially regarding a cost/performance trade-off.

Since the accelerator model is relatively simple, a future potential development direction is the porting of the scheduling tools for on-chip execution, further increasing the self-adaptability of heterogeneous computing systems. As is, the accelerator design is capable of accelerating large loop traces of MicroBlaze executables, supporting nearly all instructions of its instruction set. As such, the single-row accelerator is a fully-fledged co-processor design with a complete supporting tool chain which was implemented and thoroughly validated with a commercial FPGA board.

Chapter 7

Dynamic Partial Reconfiguration of Customized Single-Row Accelerators

The previous chapter presented and evaluated an accelerator architecture which proved to be very efficient in terms of performance and resources. Generating customized accelerators via modulo scheduling for the minimum possible Initiation Interval of all chosen Megablocks allows for performance maximization, and customizing the connectivity reduces the required resources.

However, despite this specialization, the resource requirements and synthesis time start to be considerable when the number of supported configurations increases. The amount of required connectivity increases, which leads to increased circuit complexity and therefore decreased operating frequency. Additionally, the size of the configuration memory increases as well. That is, to implement the *reconfiguration* capabilities required to support a large number of configurations, the circuit becomes increasingly less *customized*.

The introductory chapters referred that balancing these two aspects was a motivation for this work. This chapter addresses this by presenting a proof-of-concept for a system which relies on the same single-row accelerator design, this time augmented with reconfiguration capabilities based on Dynamic Partial Reconfiguration (DPR). The accelerator is re-designed slightly by partitioning it into a static region and a single reconfigurable region. The static region contains interface logic, the register pool, and both *load/store* units. The reconfigurable partition includes all other Functional Units (FUs), as well as their input multiplexers and the configuration memory. Each Megablock to accelerate is used to generate a single partial *bitstream* file. The accelerator's is reconfigured via the Internal Configuration Access Port (ICAP) as part of the migration mechanism which all implementations presented so far relied on.

As the experimental evaluation in this chapter demonstrates, this approach has two main advantages relative to the implementation in the previous chapter: (i) the total area required by the accelerator is smaller by relying on DPR to switch between accelerator configurations, versus multiplexer based configuration logic, and (ii) the synthesis time of the accelerator decreases considerably. Additionally, the motivation to implement this system also came from the notion that an accelerator architecture with DPR-based reconfiguration capabilities is an important stepping

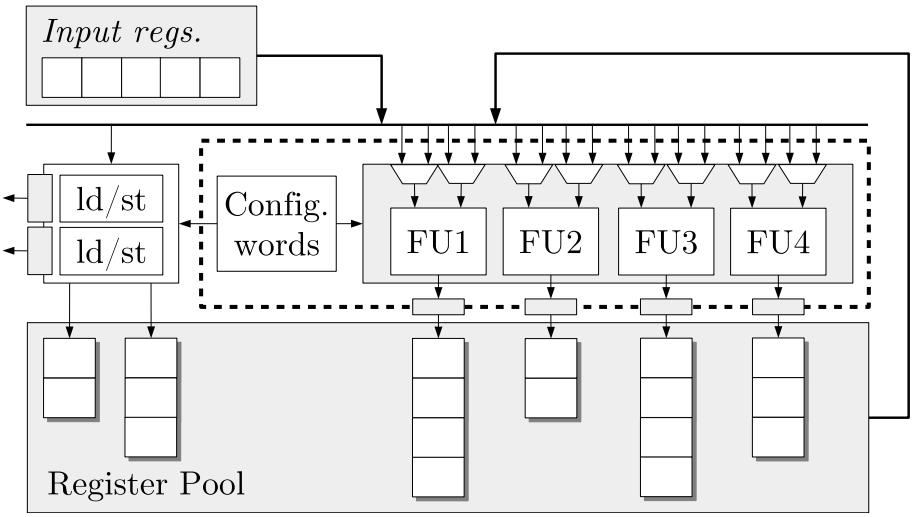


Figure 7.1: Single-row accelerator architecture partitioned for DPR. The reconfigurable partition, includes all FUs, input multiplexers, and the configuration memory.

stone in developing a system capable of runtime self-reconfiguration.

This chapter presents the partitioned accelerator architecture in Section 7.1, and the tool flow modified to support the generation of partial bitstreams in Section 7.2. The experimental evaluation in Section 7.3 discusses the overheads introduced by the runtime partial reconfiguration, as well as the resource savings achieved. Finally, Section 7.4 concludes this chapter.

7.1 Accelerator Architecture

Figure 7.1 shows a simplified view of the accelerator architecture. Some details are omitted as it is largely similar to the customizable architecture template shown in Chapter 6 (page 106).

There is only one row of FUs, customized by the modulo scheduler based on the set of Megablocks to accelerate. The accelerator supports all integer and single-precision floating-point operations, including divisions by non-constant dividers. All FUs are fully pipelined, with the exception of the non-constant integer division unit. The *load/store* units are capable of performing byte-addressed operations to arbitrary memory locations, since the accelerated traces also implement the address generation operations. When accelerating a Megablock, the accelerator executes an arbitrary number of iterations each time it is called. The execution returns to software as a result of evaluating the respective termination conditions (i.e., *branch* instructions).

The modulo scheduler generates accelerator instances capable of executing the target Megablock CDFGs at their respective minimum Initiation Interval. If this is not possible the II is gradually increased until the CDFG is scheduled. This virtually does not occur, as the only resource limitation in this approach are the two memory ports (all other FUs are instantiated as needed).

To summarize, every functional and architectural aspect is identical to what is presented in Chapter 6, but in this implementation the accelerator is partitioned into a static region and a partially reconfigurable region, which is the portion contained within the dash-lined box of Fig. 7.1.

7.1.1 Static Partition

The static partition includes the *input* registers and *output* registers as well as the multiplexers which drive them with outputs from the FUs (omitted from Fig. 7.1), the two *load/store* units and their input multiplexers, as well as all other interface and control logic.

Given a set of Megablocks, the resulting structure of the static region is very similar to what would be instantiated for the non-DPR version of the accelerator. The length of each register chain in the pool is determined by the maximum required by that chain, throughout all configurations of the partition. Likewise, the input multiplexers that drive the *load/store* units (omitted from Fig. 7.1) are part of the static region, and thus implement all required connectivity throughout all configurations. The same is true for the multiplexers which drive the *output* registers.

These components were placed in the static region since their resource requirements do not scale noticeably (e.g., output register multiplexers or register pool), or do not increase at all (e.g., memory port logic and control logic) with the number of Megablocks to support.

7.1.2 Reconfigurable Partition

The reconfigurable partition contains all FUs, their input multiplexers, and the configuration memory. Apart from some Flip Flops (FFs) needed by the memory module, the reconfigurable region logic is implemented nearly entirely in Lookup Tables (LUTs).

The modulo scheduler shown in Chapter 6 efficiently re-utilizes FUs between configurations. However, in some cases one configuration requires a large number of FU which remain unused by the remaining Megablocks. This has no effect on performance, but leads to FUs being under-utilized and to a larger accelerator area requirement. Also, since more units exist, the width of the configuration word also increases, leading to a larger configuration memory. Finally, as more operations are scheduled onto an FU, the complexity of its input multiplexer increases. These two aspects contributed especially to higher resource requirements and longer synthesis times.

By placing these components in the reconfigurable region, the input multiplexer connectivity is only as complex as required by the chosen Megablock subset, which for one or few Megablocks represent an acceptable resource cost. Regarding the FUs, the total area required when implementing a set of Megablocks will always have to be as large as the largest Megablock demands. However, it might not be necessarily as large as the area which would be required by a multi-configuration instance supporting all Megablocks. Also, by implementing each Megablock (or small subsets of Megablocks) as a single configuration, the same resources can be used to implement different FUs between configurations.

A different solution would be to increase the II of the largest Megablock, thereby decreasing the row width and avoiding the instantiation of FUs which would remain idle for other smaller configurations. However, this would increase the number of configuration words required to execute the Megablock, negating the resource savings achieved by instantiating fewer FUs.

The configuration memory was placed into this region because it was observed that, for a large number of configurations, the memory size increased considerably. Even for a few configurations,

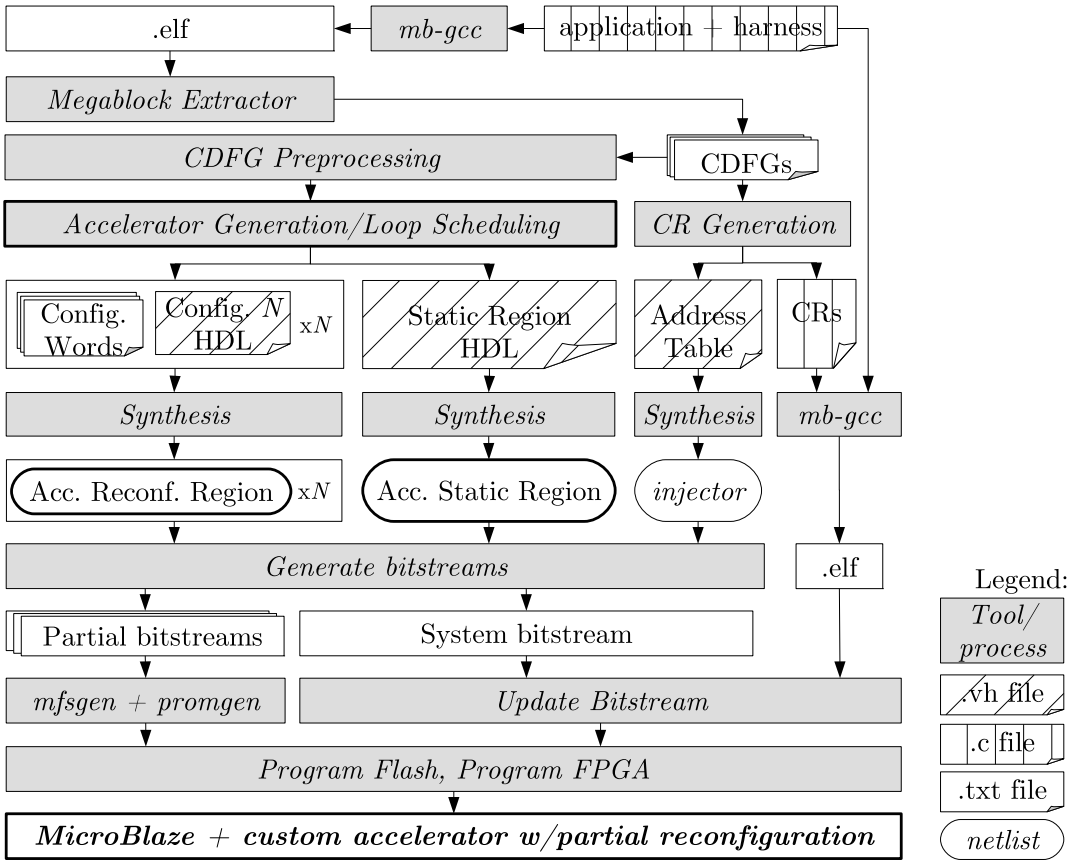


Figure 7.2: Complete tool flow for partially reconfigurable accelerator.

the instruction word width (487 bits on average for the 24 benchmarks in Chapter 6) led to a large number of LUTs being required to implement the memory as distributed RAM.

By placing the configuration memory in the reconfigurable partition, it only needs to contain the words which implement the Megablocks of the respective configuration. This is an efficient re-utilization of LUTs, since the same FPGA resources will implement the memory which will hold different configuration words based on what partial *bitstream* has been written to the reconfigurable area. As Section 7.3 shows, generating one partial *bitstream* per Megablock leads to a very noticeable decrease in the number of required LUTs, relative to a non-DPR implementation, for this reason.

7.2 Tool Flow for Dynamic Partial Reconfiguration

The tool flow for this implementation was extensively re-worked. Figure 7.2 shows the complete flow, from Megablock extraction to implementation. The initial extraction steps are identical to all other implementations, as is the CDFG preprocessing. The generation of the Communication Routines (CRs) and their integration with the application follows the same procedure, but each routine now includes a call to a reconfiguration function which is part of the harness code. This

function checks if the accelerator's reconfigurable area is already configured with the desired partial bitstream. If this is the case, the CR resumes normally. Otherwise, the reconfiguration routine writes the partial bitstream to the reconfigurable area of the accelerator through the ICAP peripheral. The *C* function used to control the ICAP is custom written, to attempt to reduce overhead. By having the partial reconfiguration take place as part of the CRs, the transparency provided by the approach is not compromised.

In previous implementations of the CR, nearly all instructions were direct reads and writes from and to the accelerator. If accelerator execution failed, the contents of the MicroBlaze's register file and stack would not suffer any modification upon return to software. If any iterations were performed on the accelerator, the contents of the register file would match what would be expected by software-only execution. The stack would not be altered in any way.

However, the routines in this implementation include the unconventional call to the reconfiguration function, and to all child functions thereof. As a result, the contents of the MicroBlaze's register file and stack do suffer unwanted modifications, whether the accelerator executes or not. As a first approach, this implementation resorts to saving the entire register file to memory, and recovering it after the partial reconfiguration function executes. The following code excerpt exemplifies this. The *_wrapCaller* auxiliary function is placed at a specific address, according to the linker script and via the respective *section* attribute.

Listing 7.1: Communication Routine with call to partial reconfiguration function

```
unsigned int seg_0_opcodes[87] __attribute__ ((section (".CR_seg"))) = {
    0xf821fffc, // save r2
    // (omitted)
        // save register file (32 instructions total)

    0x3021ff80,
    0xb0000001,
    0xb9fcf3b0,
    0x20a00000,
        // call "_wrapCaller" function

    0x30210080,
    0xe821fffc, // recover r2
    // (omitted)
        // recover register file (32 instructions total)

    // Remainder of CR:
        // (omitted)
};

void __attribute__ ((section(".pr_seg"))) _wrapCaller(unsigned int i) {

    doPR(i);            // reconfigure if needed
    putfsl(1, 0);       // soft reset after partial reconfiguration
    putfsl(1 << i, 0);  // set configuration number
    return;             // return to CR
}
```

The scheduler now generates one accelerator configuration per CDFG, along with a single configuration file which determines aspects relative to the static region (e.g., the size of the register pool and the connections between its registers). Each configuration specification (i.e., the *Configuration N HDL* and *Configuration Words* files in Fig. 7.2) determines the instantiated FUs, their interconnections and the contents of the configuration memory. In this version of the tool

flow, the scheduler does not yet generate a configuration which supports multiple Megablocks. In this scenario, the memory would hold the configuration words for the execution of multiple traces, just as the implementation in Chapter 6.

The bitstream for the base system (i.e, all modules and peripherals, plus the static region of the accelerator) is generated first. Afterwards, each synthesized version of the reconfigurable region imports the base system to generate all partial bitstreams. There are as many partial bitstreams as chosen CDFGs. All bitstreams are placed into an MFS file system, which is a lightweight Xilinx proprietary file system, for which a *C* library is provided. Using *promgen*, a file which can be used to program the non-volatile flash on the used VC707 test board is generated. This file contains only the partial bitstream files. The FPGA itself is configured in a separate step (i.e., it is not programmed from the flash at boot). The exact commands for these two steps are the following:

Listing 7.2: Generating a flash programming file from a file system with all partial bitstreams

```
mfsgen -vcf pr.mfs <partial bitstream files>
        # generate the file system
promgen -b -w -p mcs -c FF -s 4096 -bpi_dc parallel \
-data_width 16 -data_file up 0 pr.mfs -o pr.mcs
        # generate the file for flash programming;
        # options are specific to the flash memory on the VC707
```

The specific area of the FPGA which was used as the reconfigurable region of the accelerator was manually specified. The area encompassed approximately two thirds of one of the lowermost clock regions in the FPGA. It contains a total of 12800 LUTs and 25600 FFs, and corresponds to approximately one fourteenth of the device. The current implementation of the flow does not automatically allocate a reconfigurable region based on the largest resource requirements observed in the synthesis reports of each possible configuration (the reconfigurable area must be specified before the *translate* and *map* stages).

7.3 Experimental Evaluation

This implementation was evaluated with the same set of benchmarks as those used in Chapter 6: 13 single-precision floating-point kernels from the *Livermore Loops* [Tim92] and 11 integer benchmarks from the TEXAS IMGLIB function library [Tex]. The purpose of using the same benchmarks was to determine the resource savings due to DPR for multi-configuration accelerators supporting the same Megablocks.

7.3.1 Hardware Setup

Figure 7.3 shows the system architecture used to evaluate this accelerator implementation. All program code (including the application, harness and partial reconfiguration functions) and data reside in local memories. At boot time, the external memory is initialized with the file system

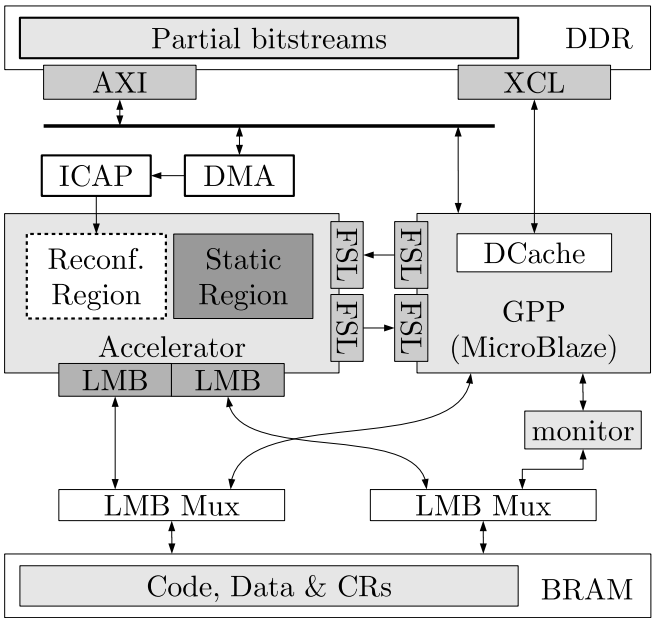


Figure 7.3: System architecture for validation of DPR capable accelerator

containing all partial bitstreams, by copying it from the non-volatile Flash memory. The accelerator is partially reconfigured during the injector-driven migration process. The CRs now contain a call to a function (written in *C*) which, according to the calling CRs, reconfigure the accelerator with the appropriate bitstream. In order to avoid reconfiguration overhead, a DMA module is used to send the partial bitstream to the ICAP. The target platform was the VC707 evaluation board, containing a Virtex-7 *xc7vx485* FPGA. The tools used for synthesis and bitstream generation are from release 14.7 of Xilinx's ISE Design Suite.

7.3.2 Software Setup

The benchmark setup is identical to that of Chapter 6. All kernels are placed into a single file, and compiled along with the test harness which retrieves execution times, compares results with a reference data generated on a desktop machine, and allows for specifying parameters such as the combination of kernels to execute and amount of data to process. Data is allocated at runtime onto the heap, which the accelerator is capable of accessing by receiving live addresses as operands, at the start of execution, from the MicroBlaze. The memory initialization function is called explicitly at boot time as part of the harness, but the partial reconfiguration function is called only by the CR. The harness requires 3.3 kB, and the partial reconfiguration functions require 0.55 kB.

7.3.3 Resource Requirements of Static and Reconfigurable Regions

To determine how the resource requirements of the reconfigurable region scale with the number of supported configurations, a total of 11 accelerators were generated, each supporting one Megablock more than the last. The Megablocks used were those from the integer benchmarks. The accelerator with one configuration supports *i1*, the next supports *i1* and *i2*, etc.

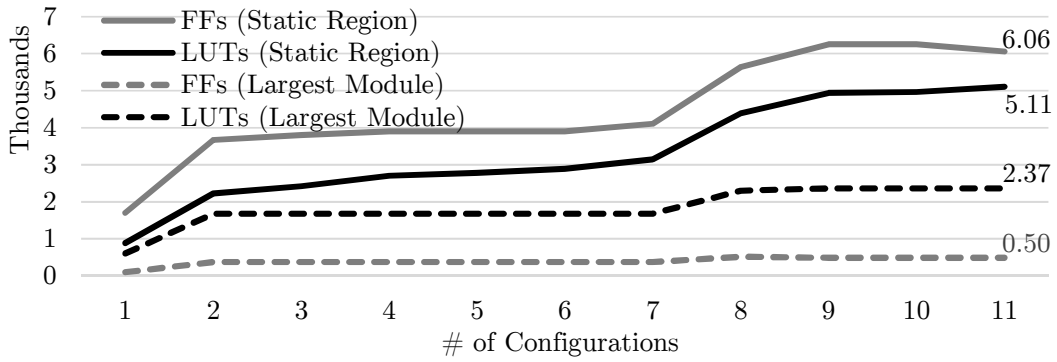


Figure 7.4: Resource requirements of DPR-based accelerator discriminated by static and reconfigurable regions

Figure 7.4 shows the resource requirements of the accelerator with DPR, for the static region, and for the largest module in the reconfigurable region. These resource requirements are as indicated by the synthesis reports. The number of maximum FFs and LUTs required by the reconfigurable region only increases when a new, larger, configuration is added. In contrast, the resource requirements of the static region increase slightly with each configuration, since it includes the input multiplexers of the *load/store* units and output registers. The total of input and output registers increases as a function of the largest number of inputs and outputs amongst all the supported Megablocks. The largest instance (i.e., for 11 Megablocks) requires 6555 FFs and 7474 LUTs. This is equivalent to 2.93× the FFs and 2.82× the LUTs required by a MicroBlaze with an FPU and without a data cache (i.e., the same configuration as the previous chapter). The requirement is considerable, but note that for the implementation in Chapter 6, an accelerator which supported *i8* and *i9* alone required 5405 FFs and 8137 LUT, according the synthesis report.

7.3.4 Resource Requirements of DPR Accelerator vs. Non-DPR Accelerator

Figure 7.5 compares the resources requirements of the DPR and non-DPR accelerators. The same sequence of supported configurations for Fig. 7.4 is used. The requirements shown were taken from the synthesis reports of both accelerator archiectures. For the DPR accelerator, the resources shown are the sum of the resources of the static region plus those of the largest variant of the reconfigurable region. Note that the curves in Fig. 7.5 depend on which configurations each accelerator, at each index, supports. Despite this, it is possible to compare how each architecture reacts, in terms of resource requirement increase, when a new configuration is added. Also, the order in which configurations are added is irrelevant when comparing the the two architectures for the case with all configurations.

Given this, the number of required FFs is nearly equal for both architectures. It is for the number of LUTs that the use of DPR demonstrates its advantages. For the 10 configuration case, the number of required LUTs is 20273, which is 2.76× more than the equivalent DPR-based version. The case for 11 configurations did not finish for the non-DPR accelerator, due to long synthesis times. Additionally, an accelerator supporting 15 floating-point Megablocks was implemented,

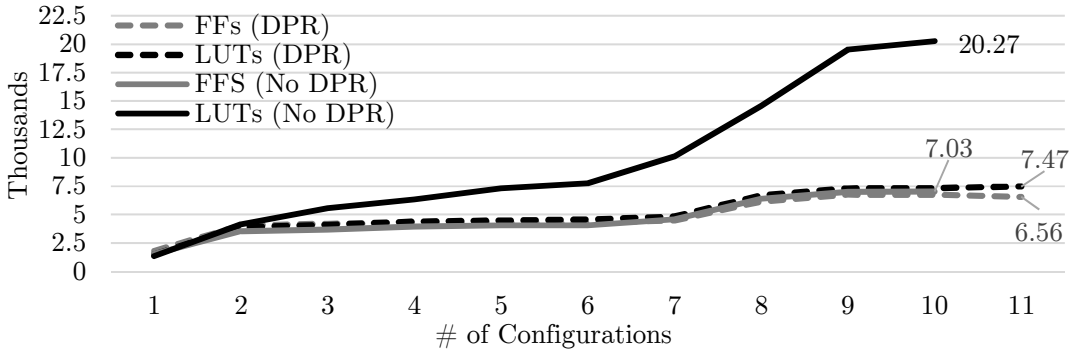


Figure 7.5: Resource requirement comparison between DPR and non-DPR capable accelerators. The number of FFs required by the DPR and Non-DPR versions are nearly identical for all cases.

and compared to an equivalent non-DPR accelerator. The former case requires 7187 FFs and 8100 LUTs, while the later 8326 FFs and 25566 LUTs. The main reason why many LUTs are required on accelerators with many configurations is the size of the configuration word memory, since it is implemented as distributed RAM. Instantiating a single memory, with all configuration words for all Megablocks, increases the total number of words in the memory and also the word width.

The effect of the word width increase can be observed in Fig. 7.5. The marked increase in LUTs for the non-DPR accelerator, which happens for indices 7, 8 and 9, corresponds to adding the Megablocks for *i7*, *i8* and *i9* as configurations of the accelerator. The Megablock for *i7* contained 40 instructions, and the ones for *i8* and *i9* are the largest for the integer set, with 142 instructions each. This leads to a configuration word width of 1436 bits, whereas the width was 784 for the accelerators between indices *1* and *7* (which was imposed by the then largest Megablock, extracted from *i2*). As a result, the size of the configuration memory for the non-DPR accelerator increases from 26 kB to 114 kB, from index *7* to index *9*. For 10 configurations, the memory size is 231 kB.

For the DPR-based accelerator the configuration memory is part of the reconfigurable region. Therefore it is only as large as the largest set of configuration words (in terms of total number of bits required) amongst the configurations. As a result, the DPR based accelerator suffers only a minor increase in number of required LUTs at this point. The size of the configuration memory is determined only by the largest configuration, and is 33 kB for the accelerator of index *9*.

For an additional comparison, consider the same groups of kernels used in Chapter 6 (Table 6.3, page 124) to evaluate the multi-configuration performance and resource requirements. Table 7.1 shows the resource requirements, according to the synthesis reports, for the accelerator design of the previous chapter on the right hand side, and for the present design on the left hand side. For the latter case, the resource requirements were calculated as the sum of the resources reported for the static region, plus the resources required by the largest variant of the reconfigurable partition. The reduction in both LUTs and FFs from the non-DPR case to the DPR case is consistent with the evaluation presented in Fig. 7.5. That is, for up to 3 configurations, the number of LUTs decreases by 0.77× on average, while the number of FFs is nearly the same.

Table 7.1: Resource requirements for several multi-configuration accelerators, normalized to the requirements of a single MicroBlaze, according to synthesis reports

ID	# Megablocks	DPR Based Accelerator (Static Region + Largest Module)		Non DPR Accelerator	
		Flip-Flops	Lookup Tables	Flip-Flops	Lookup Tables
f_2_8	2	2.22	1.97	2.34	2.35
f_3_9	3	0.98	0.91	0.95	1.24
f_4_5_6	5	2.43	2.23	2.77	4.32
f_7_10	2	1.13	0.95	1.05	1.09
i_5_6	2	0.78	0.61	0.72	0.76
i_8_9	2	2.74	2.25	2.70	3.38
i_10_11	2	0.97	0.72	0.88	0.76
mean		1.61	1.38	1.63	1.99

	Flip-Flops	Lookup Tables
Microblaze (w/FPU; no caches)	2236	2654

The resource utilization of the DPR-based accelerator was estimated as presented since otherwise synthesizing the accelerator as a whole (i.e., replacing the black box reconfigurable region with the largest variant) would cause the synthesis tools to perform optimizations across the boundaries of both regions. On the other hand, this prevents the calculation of the synthesis frequency, since the tools cannot analyse signal delays across the region boundaries. So as an estimate, single-configuration instances of the DPR-capable version were synthesised (where the reconfigurable region black box is replaced with an actual instance). Considering the worst synthesis frequency for all cases (e.g., the lowest frequency between the two Megablocks for *f_2_8*) the average synthesis frequency is 209.9 MHz, versus the 205.2 MHz for the non-DPR accelerator.

Note that although the results presented thus far are good indications of resource consumption, the synthesis reports do not take into account the placement constraint which defines the reconfigurable area. So, the multi-configuration cases in Table 7.1 were fully implemented to retrieve more accurate results from the post-map reports. The post-map reports indicate that the average accelerator requires 4074 LUTs, 3220 FFs, and 2051 slices. This corresponds to 1.79× the slices of the MicroBlaze instance used in the same system (i.e., with an FPU and a data cache). But when compared to the MicroBlaze instance used in the last chapter (i.e, without data cache and an FPU), this corresponds to an increase of 2.39×. This is considerably more than the average size of the non-DPR accelerator instances, which required 1.86× the slices the MicroBlaze required.

This occurs because, it is not possible to place the logic of both the static and reconfigurable regions in to the same slices. This means a larger number of slices is required (and to a lesser extent, LUTs) if the number of loops supported is too little to reach an advantageous trade-off. This is true for nearly all cases of this set, save for *f_4_5_6*, which supports 5 configurations.

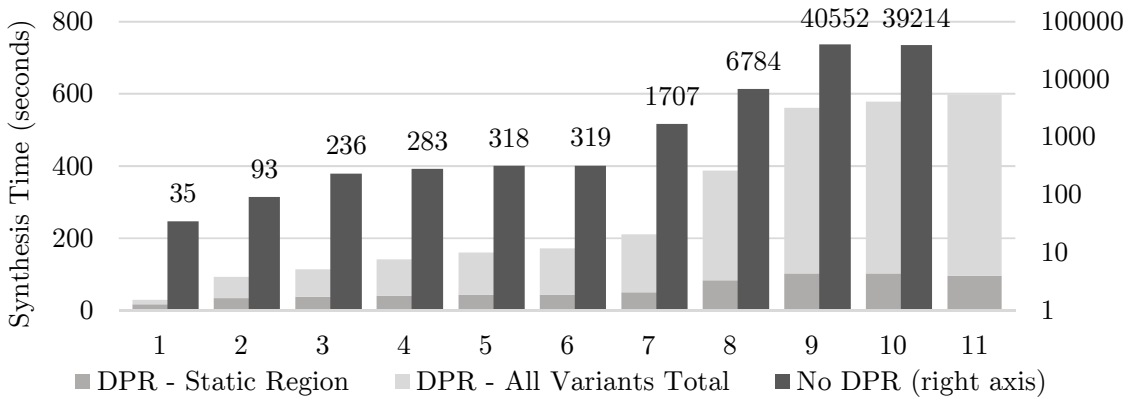


Figure 7.6: Synthesis times for DPR and non-DPR capable accelerators

In this case, the number of LUTs required by the accelerator is 0.51× the number required by a non-DPR accelerator, and the number of slices is 0.88×.

The exact same reduction in the number of LUTs is indicated by the respective synthesis report, since the amount of resources relative to the reconfigurable area outweighs the overhead of partitioning the accelerator. This suggests, especially for numerous configurations, that the synthesis reports, and therefore the comparison in Fig. 7.5, provide good estimates of LUT savings, and also area savings, since the logic in the reconfigurable area outweighs the static region.

In fact, to be more precise, since the DPR capable accelerator can support a (theoretically) unlimited number of configurations in the same area, the actual resource and area requirements it represents is in fact determined by the area reserved for the reconfigurable region. For this evaluation, the area corresponded to 12800 LUTs and 25600 FFs, which was much larger than what was required.

7.3.5 Synthesis Time of DPR-Capable Accelerator vs. Non-DPR Accelerator

Resorting to DPR leads to an additional and very significant advantage: the synthesis time of the accelerator drastically decreases when targeting multiple loops. The synthesis time required for the DPR-capable accelerator is equal to the synthesis time of the static region, plus the synthesis times of all variants of the reconfigurable region. Figure 7.6 shows this, and the synthesis times of the non-DPR version of the accelerator. For the latter case, the synthesis times become prohibitively large for more than 8 configurations. For 10 configurations, the time is close to 11 hours. For the case with 15 floating-point configurations, the non-DPR instance required 59 hours for synthesis, while the DPR design only required 14 minutes.

7.3.6 Effect of Partial Reconfiguration Overhead on Performance

The purpose of this chapter was to demonstrate the resource savings on the accelerator when resorting to DPR. However, the partial reconfiguration process introduces overhead when using the accelerator. For completeness, this section also presents the measured speedups and DPR

Table 7.2: Partial reconfiguration overhead and speedups, for $N = 1024$ and three values of L

ID	Benchmark	Overhead from Partial Reconf. (% of accelerator execution time)			Overall Speedup		
		$L = 500$	$L = 1000$	$L = 2000$	$L = 500$	$L = 1000$	$L = 2000$
f2	diffpredict	42	26	15	3.50	4.42	5.09
f8	intpredict	39	25	14	4.35	5.44	6.23
f3	glinrecurrence	4	2	1	2.05	2.10	2.12
f4	hydro	19	11	6	9.65	10.7	11.3
f5	hydro2d	75	75	75	0.26	0.26	0.26
f6	hydro2dimp	10	5	3	9.18	9.66	9.91
f7	innerprod (fp)	14	8	4	3.69	3.97	4.13
f9	lin_rec	97	97	97	0.08	0.08	0.08
f10	matmul (fp)	<1	<1	<1	3.07	3.08	3.08
i5	sad16x16	<1	<1	<1	1.06	1.06	1.06
i6	mad16x16	<1	<1	<1	1.06	1.06	1.06
i8	dilate	17	9	5	3.84	4.20	4.40
i9	erode	17	9	5	3.84	4.20	4.40
i10	innerprod (int)	18	10	5	3.10	3.41	3.58
i11	matmul (int)	<1	<1	<1	2.41	2.41	2.41
	mean	6	4	2	2.12	2.27	2.35

overheads for the multi-configuration cases shown in Table 7.1. Only these cases were used, as they were readily available. Note that each supported Megablock is still modulo-scheduled at its minimum possible II, so any decreases in performance are due to DPR overhead.

Table 7.2 shows how much of the accelerator execution time corresponds to partial reconfiguration time, for several consecutive calls of the same kernel, which is determined by the parameter L. Since these results were retrieved from complete on-board implementations, this demonstrates the correct functioning of the transparent DPR-based reconfiguration. The averages shown are arithmetic for the overhead and geometric for the speedups.

The time required to reconfigure the accelerator (i.e., to write a partial bitstream to the reconfigurable area) was estimated by considering the time measured by the *injector*. For most cases in Table 7.2, the respective accelerator supports a single Megablock. This means that the DPR overhead is incurred once. For later invokations of the accelerator, the partial reconfiguration is skipped. Measuring the execution time for two values of L is enough to determine which portion of the measured time corresponds to the partial reconfiguration. For all cases the time is nearly identical, 3.5 ms, considering the system clock frequency of 100 MHz.

This leads to a low overhead in most cases, although for *f2* and *f8* the overhead is still significant, since the value of N used, 1024, results in a lower number of iterations in comparison to the remaining cases. However, the effect of increasing L is observed for all cases, as the impact of overhead diminishes. For a large enough value of L the DPR overhead would become insignificant, since it only occurs once. In a realistic scenario however, the accelerator would be reconfigured often if it supported several Megablocks.

This can be observed for *f9*, where two Megablocks were accelerated. Each Megablock represents a small loop which iterates N times. Since the harness repeats the execution of both loops L times, this means that a partial reconfiguration of the accelerator takes place a total of $2 \times L$ times. As a result, the execution time inscreases drastically, since each kernel repetition requires two partial reconfigurations. The same is true for *f5*, which supports 3 Megablocks.

The reconfiguration overhead could be reduced in several ways. Firstly, the size of each partial bitstream file is 592 kB. Reducing the size of the file would shorten the time required for the DMA to transfer it to the ICAP. Compressing the bitstream files was attempted, but the compression ratio was too small to result in any significant difference. Instead, reducing the reconfigurable area to the minimum required would be the most straightforward way to reduce reconfiguration overhead for the current system implementation. All of the used partial bitstream files have the same size, since this aspect is not determined by the logic they contain, but by the size of the reconfigurable area. Secondly, existing approaches have shown that DPR overhead can be reduced by, for example, greatly increasing the ICAP operating frequency [HKT11].

However, the overhead of utilizing a DPR capable accelerator in this approach can be reduced due to the capacity for mixed-grain reconfiguration. That is, the overhead can be reduced if a single circuit configuration (i.e., partial bitstream) is used to support multiple Megablocks using multiplexing logic, as the previous approaches have shown, instead of implementing each in a separate partial bitstream. Determining which Megablocks to support in a single circuit configuration for the reconfigurable area allows for balancing the aspects of overhead and circuit complexity.

7.4 Concluding Remarks

This chapter presented a proof-of-concept system relying on a single-row accelerator design, augmented with Dynamic Partial Reconfiguration (DPR) capabilities, to implement the transparent acceleration of binary traces. The approach relies on generating customized instances of the accelerator, and in using DPR for the purpose of switching between supported configurations.

The DPR based reconfiguration, as opposed to implementing all reconfiguration capabilities via in-module logic, allows for very significant resource savings, as well as reduced synthesis times for the accelerator instances. The approach is advantageous when wishing to support a large number of Megablocks. For instance, for a set of 7 accelerator instances, each of which supported 2 to 4 Megablocks, the number of LUTs required by the DPR-capable accelerator, according to post-map reports, is 1.13× that of the non-DPR version, and the area (indicated by number of slices) is also larger. However, for one accelerator in this set which supported five Megablocks, the DPR-based accelerator requires 0.51× the LUTs and 0.88× the slices relative to an equivalent non-DPR instance. As another example, the number of LUTs required by an accelerator supporting 10 Megablocks can be reduced by 2.76× by relying on DPR.

As expected, the DPR overhead has a noticeable effect on speedups. However, the size of the reconfigurable area was larger than what was required, meaning the reconfiguration time could be reduced. Additionally, other works have proven that the DPR time can be greatly reduced

relative to the vendor-provided flow (as per Section 2.3.2), either by adopting similar DMA-based solutions or custom ICAP controllers entirely, capable of higher throughput [VF12, EBIH12]. Finally, the ICAP module could also be driven at a higher clock frequency [HKT11] than the 100 MHz used for this evaluation.

Also, the current version of the scheduler does not perform any kind of intelligent decision regarding which Megablocks should be grouped to generate a partial bitstream. This evaluation relied on generating one configuration per Megablock. Instead, the most efficient approach would be to generate partial bitstreams which support multiple Megablocks. In this scenario, some of the reconfiguration capabilities come from completely modifying the accelerator circuit via DPR, and some from relying on multiplexing logic as the previous approaches have shown. This has the potential to greatly reduce the required partial bitstream storage capacity and especially the reconfiguration overhead, since DPR would not be required as frequently.

Regardless, this fully functional implementation of an accelerator with DPR capabilities as presented in this chapter is very a significant step towards self-adaptive systems. The framework of the accelerator is a viable starting point for development of and embedded tool for automatic acceleration of frequent traces by on-chip generation of configurations for the reconfigurable region [SF12, SPA08, BFD08]. On its own, this approach provides a high degree of transparency for the application programmer, allows for fast generation of accelerator hardware, achieves considerable speedups, and exploits the yet underutilized technological feature of DPR. This accelerator design and evaluation demonstrated, by relying on existing vendor tool chains alone, that this capability, unique to FPGAs, allows for significant improvements in terms of resource efficiency. In the context of this work specifically, it allows for generation of compact accelerators supporting large numbers of configurations, which do not sacrifice performance for the sake of flexibility.

Chapter 8

Conclusion and Future Work

This thesis presented a transparent binary acceleration approach, based on creating a customized accelerator for the execution of hot spots. The hot spots are represented by Megablocks, which for the purposes of this work were obtained offlina via simulation. The main focus was to devise a methodology for embedded application acceleration which was lightweight, fast, and efficient.

The motivation was two-fold. Firstly, to further advance the already developing field of automated HW/SW partitioning approaches. Existing approaches are useful for developers comfortable with embedded development and tools. The tools and methodology developed in this work can be adopted by this type of user, but an effort was also made to make it more approachable by developers less familiar with these environments, by keeping the tools on the back-end.

Secondly, the approach was meant to efficiently exploit any inter- and intra-iteration instruction parallelism, to maximize performance. An accelerator architecture template is automatically customized to execute a given set of Megablock traces, focusing on a trade-off between *specialized* single-configuration circuits and *reconfigurable* circuits supporting multiple loop traces.

8.1 Characteristics of the Developed Approach

The developed approach relies only on binary trace information that is produced after compilation by a simulation step. No source or binary code modifications are required, either manually or during compilation via a custom compiler. An accelerator specification is quickly generated, usually in the order of seconds, from a set of chosen instruction traces to accelerate. The tools also generate the necessary communication primitives between host processor and accelerator, and the transparent migration makes it unnecessary to modify the application post-partitioning to implement the communication between components.

In Chapter 4, the first implementation of the accelerator architecture and tools was presented. This first design served mostly to validate the approach at a system level, demonstrating that execution remained functionally correct while being transparently migrated to the accelerator by the *injector* and CR method. Execution of Megablock on the accelerator did not support memory accesses nor exploit inter-iteration parallelism, but multiple configurations were supported via

interconnection configuration at the register-level. The applicability of the approach was vastly improved in Chapter 5 with the addition of up to two concurrent memory accesses. Supporting acceleration of traces with memory access operations enables the approach to target realistic workloads. In Chapter 6, the area/performance trade-off is vastly improved with a compact architecture that also exploits intra-iteration parallelism. Loops are modulo-scheduled, taking into consideration the two memory port limitation and its latency. If relying on local memories with a constant known latency, execution on the accelerator never idles waiting for a memory access. Additionally, the applicability was considerably increased by supporting floating-point operations. Finally, in Chapter 7 the accelerator architecture of the previous chapter is modified to support Dynamic Partial Reconfiguration (DPR) . The accelerator is partitioned into a static and a reconfigurable region. Instead of supporting multiple configurations by increasing interconnection complexity, one partial bitstream is generated per Megablock to support. This way, multiple configurations are supported without increasing circuit complexity while also saving resources and area.

To summarize, the developed tools and accelerator designs benefit from the following features and characteristics:

1. No source or binary code modification is required, before or after partitioning
2. Automated generation of specialized accelerators capable of executing multiple-loops
3. Support for floating-point operations on the accelerator, via fully pipelined units
4. Support for concurrent data memory accesses (including heap allocated memory)
5. Transparent migration of execution from software to hardware
6. No costly data transfers between processor and accelerator when relying on local memories
7. Exploitation of intra-iteration ILP
8. Exploitation of inter-iteration ILP via modulo-scheduling at the minimum possible IIs
9. Runtime reconfiguration of the accelerator via DPR

The experimental validations presented, for the several accelerator designs, showcased fully functional transparent binary acceleration systems. With each accelerator implementation, the performance improvements and area efficiency increased considerably. The earlier implementation in Chapter 4 was only capable of targeting integer kernels, and the mean geometric speedup resulting from exploiting intra-iteration ILP was of 2.08× for a mean resource cost approximately 2.5× that of a single MicroBlaze. The single-row implementation in Chapter 6 decreases this cost to 1.12×, while increasing the performance to a geometric mean speedup of 5.61×. In Chapter 7, the use of DPR to reconfigure the accelerator lowers its resource requirements and synthesis time. This is especially noticable for large numbers of configurations, allowing for acceleration of numbers of Megablocks which for a non-DPR accelerator would entail a very high resource cost.

The vision of a fully autonomous self-adaptive system motivated some aspects of this work, but the presented approach, which relies on the described offline/online flow, is useful on its own right. It may be employed by designers even when only binary executables are available, for a quick and transparent detection of the critical application portions. The fast generation of custom accelerator circuits delivers a significant performance increase with very little effort.

This is aided by the fact that the accelerator relies on standard Xilinx interfaces. For instance, the use of FSL connections to communicate to the host processor is compatible with Xilinx's intended use of the interface regarding the use of co-processors. Also, relying on unmodified binary means the vendor compilation flow is undisturbed. An entire accelerator-augmented system can be instantiated using Xilinx's embedded development environments, as was the case for all experimental scenarios in this work. The user only needs to generate the accelerator specification prior to bitstream generation. So, this would be the primary use case for the developed approach and architectures. Integration of the developed tools into the vendor would also be possible.

The described scenario relies on having one accelerator connected to one FSL port (or PLB in earlier implementations). But two additional use cases are possible with very minor modifications the developed flow. The first would be the generation of several accelerators (either single- or multi-configuration), each coupled to its own FSL interface. This would reduce the circuit complexity that a single, larger, accelerator would suffer from, at the cost of additional circuit area. A second case would be the use of one accelerator by two or more MicroBlaze processors in a time multiplexed fashion. The accelerator architecture is abstracted from the processor execution context, so supporting multiple CDFGs originating from different binaries would also be possible. For any use case, using DPR for module-level reconfiguration would be an optional feature.

8.2 Future Work

There are three major aspects to highlight regarding future work. Firstly, some potential improvements to the primary use case of the developed approach, mostly related to tool flow and integration, are listed in the next section. Secondly, the achievable performance would increase if conditional execution was supported on the accelerator. This is explained in Section 8.2.2. Finally, this work was motivated by the concept of a fully autonomous self-adaptive system to be developed in the future. With this in mind, the accelerator progressed towards an architecture which was envisioned to be a suitable target for on-chip configuration generation and reconfiguration. Section 8.2.3 discusses the developments and potential solutions necessary to realize this concept.

8.2.1 Potential Improvements to the Developed Approach

Although fully functional, there are some aspects that could be addressed to considerably increase the applicability of the approach. The following limitations can be identified, and the succeeding paragraphs briefly explain how they could be addressed.

Only the MicroBlaze Instruction Set is Supported Supporting other instruction sets implies the need for a different simulation platform to retrieve traces and possibly the design of additional FUs, such that each instruction in the chosen host processor's instruction set maps to an FU type (save for special register handling instructions). Also, the developed scheduling and translation tools can generate an accelerator from any CDFG, as long as that graph represents a single iteration of a loop path, the nodes are host processor instructions, and the inputs/outputs of the graph represent registers from the processor's register file. That is, support for a different RISC instruction set would not require significant architecture or approach modifications beyond a simulator/platform to extract execution traces.

The Accelerator and Tools are Geared Towards Xilinx Platforms Likewise the accelerator and *injector* rely on the interfaces employed by the MicroBlaze and LMBs. Supporting another processor, or type of memory interface, would require either new interfaces or bridges. The *injector* especially would require modification, as its operation is dependent on the instruction bus behaviour as driven by the MicroBlaze.

Interference by Adding CRs to Application via Re-Compilation The use of a re-compilation step to add the CRs to the binary could be avoided by instead directly modifying the binary using *objcpy*, which is capable of adding sections to the ELF file.

Manual Selection of Extracted Megablocks Automating the selection of Megablocks to accelerate would fully automate the translation flow. It would be relatively straightforward to parse the Megablock information and select those with the highest execution time, or highest number of instructions. However, it would not be as easy to exclude traces which represented processing by functions such as *printf*, or other peripheral communication, without a more in-depth processing of the executable. Accelerating this type of Megablocks is not currently possible, since the accelerator would require access to MicroBlaze's memory mapped peripherals.

Accelerating Software Emulation Sub-Routines Another considerably more complex aspect is the identification of operations such as float-point addition or integer division when they are performed via software emulation. This would allow for executing these operations on the accelerator without enforcing the presence of an FPU or integer divider on the MicroBlaze, which is necessary to ensure that the respective trace instructions appear. Again, this would require processing of the application's ELF file to determine symbol names. Afterwards, calls to such sub-routines could be replaced in the detected traces with equivalent accelerator supported instructions, if any.

Integration with Vendor Flows Currently the tools lack integration with the vendor development flows in such a way that the partitioning process, from Megablock detection to bitstream generation, is a seamless GUI based process. This aspect and the automated instantiation of the accelerator module and supporting hardware would be the final integration steps.

Implementing Efficient External Memory Access Supporting external memory access allows for a acceleration of larger applications, especially those which deal with large volumes of data. Also, realistically, an embedded system contains multiple layers of memory, meaning the host processor resorts to caches. Sharing, or synchronizing, the accelerator and processor data caches efficiently, and ensuring the data parallelism to exploit is not hindered by the memory latency, are the two main aspects of efficient external memory support. This is explored by the proof-of-concept in Appendix A.

8.2.2 Support for Multi-Path Traces

One significant limitation of the developed approach is that the accelerator cannot execute an entire loop body which contains multiple paths. This is due to the nature of the Megablock trace, which by definition does not capture the multiple execution paths through a loop, and the conditional operations which control assignment of values to registers. One possible solution is to process two or more traces starting at the same address, post-extraction, in order to generate a single CDFG with control nodes representing conditional value assignments. The accelerator would have to be augmented also, by having dynamic control for register write-enables based on results of previous FUs. Support for multi-path traces would significantly improve the accelerated code coverage and would further abstract the software developer from partition limitations.

8.2.3 Runtime HW/SW Partitioning via DPR

The previous section outlined some potential improvements of the developed work. Addressing those issues, especially automated selection of traces and exclusion of certain code regions, would contribute to further automating the approach. However, two approach-level aspects stand out as potential future work towards a self-adaptive system. The first issue is relative to runtime detection of traces to accelerate and their translation into a CDFG format (which involves determining data dependencies), whilst the second is the generation of accelerator configurations.

Runtime Trace Detection and CDFG Generation The first problem is essentially a pattern detection issue. Unlike an offline simulation step, this task is more difficult if performed on-chip. Firstly, a monitoring mechanism is needed to observe both the instruction address and the instruction itself. The *injector* is a step towards this function. However, in order to detect patterns the sequence of observed instructions must be stored and, at every new observed instruction, a number of comparisons must be performed in order to detect a repeating pattern. The work in [BPCF13] proposes some hardware solutions for pattern detection. Implementing a pattern detector in hardware implies a high resource cost, and potentially limits the detectable pattern size.

Although detection of traces could be performed by a dedicated hardware module despite the cost, it would be very difficult to extract the needed CDFGs representations from the patterns. So, an alternative would be to have a software driven pattern detection and CDFG extraction. The work in [LV09] relies on a second MicroBlaze to perform this type of on-chip CAD tasks. But a

different solution can be outlined: use the main processor itself to perform trace pattern detection, by periodically processing a trace (of a maximum specified size) stored in an auxiliary memory.

The left-hand side of Fig. 8.1 demonstrates this concept. An external interrupt driven by the monitoring module sends the host processor to an embedded partitioning routine. The interrupt could potentially be triggered based on counters which register how often a backwards branch address is observed, since this is an indicator of loops. Up to N instructions executed prior to the interrupt would be stored in the auxiliary memory. The cost of doing this is relatively minor. The Megablock traces accelerated throughout the experimental evolutions in this work contained approximately up to 120 instructions, so a memory size of 3 kbit would suffice.

To mitigate the overhead of halting the application to perform partitioning, the interruptions could be triggered only when the accelerator is executing (after at least one configuration is already generated). The embedded tools would also need to generate the CRs, place them into memory and configure the *injector*. As with the developed approach, the binary would remain unmodified, acting as software fallback. By relying on interrupts, the embedded partitioning tools would not have to be manually integrated into any target application by the designer. The library would simply be added to the compilation chain, and interrupts would have to be enabled.

A large part of the effort associated with this solution would be the porting of the existing Megablock extraction and developed translation tools to versions suited for an embedded system.

Runtime Accelerator Generation via Partial Bitstream Manipulation The second issue lies with the generation of accelerator configurations. The solution proposed for the extraction of CDFGs also works for this task: the host processor could be used to generate accelerator configurations. So the specific problem is the choice of accelerator reconfiguration/customization method. The approaches presented in this work relied on offline HDL specifications, which is not an option in an on-chip scenario. So two possible alternatives exist: 1) rely on a static, ALU-based accelerator with configurable interconnects, or 2) rely on DPR to generate custom circuits on-chip by combination of coarse-grained components, stored as partial sub-modules. Keep in mind that the solutions proposed here consider the accelerator architecture shown in Chapter 6 as the target.

The first alternative is the approach typically adopted for existing works. It was also proven to be efficient for the accelerator architecture developed in Chapter 6 by the evaluation of ALU-based instances. In this case, the accelerator would also need crossbar-type connectivity since the configurations would be generated online. The register pool would need to be generic as well.

The second approach would rely on having an accelerator with a static portion and several partially reconfigurable areas. Each supported FU would be stored as a single partial bitstream file, so each area would be as large as the largest FU. Some high-cost units could be made static as well, like the FPU. The right-hand side of Fig. 8.1 illustrates this.

Generating a configuration would entail determining the FUs to instantiate and their connectivity, similar to the developed work, but relying on DPR. The overhead of generating a configuration would only be incurred once. The information as to what combination of coarse-grained modules to place on the single-row as well as the connectivity could be stored for later accelerator calls.

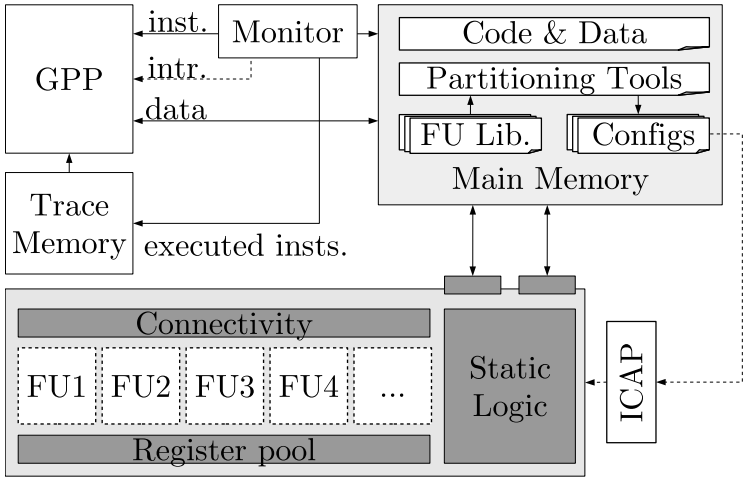


Figure 8.1: Concept for a fully autonomous self-adaptive system based on Dynamic Partial Reconfiguration

The other source of overhead would be the DPR reconfiguration itself as the accelerator is called to accelerate different translated traces. By reconfiguring the accelerator FU by FU some of this overhead could be eliminated, since a given needed unit may already be instantiated when reconfiguring. The translation process could be intelligent so as to maximize the number of matching units per type and position, decreasing the DPR overhead.

Generating the connectivity and register pool dynamically is more difficult. A possible first solution would be to rely on crossbar connectivity and generic register pool, both residing on the static area of the accelerator.

8.3 Concluding Remarks

This work developed a fully functional transparent binary acceleration approach. Specifically, tools relying on binary trace information alone automate the generation of customized reconfigurable accelerators. The most efficient accelerator architecture is, on average, the size of a single MicroBlaze processor, and provides significant performance increases, especially given the small effort required to generate an accelerator instance.

For software developers without hardware expertise the automated hardware generation makes heterogeneous systems more approachable targets. Even for experienced developers, the time saved in hardware design is a considerable advantage. The approach relied on FPGAs as the target devices, as they have evolved to be viable as deployment platforms, especially given that the current trend seems to be towards integrating FPGA fabric as part of a System-on-a-Chip [].

To conclude, the developed approach is useful for an embedded development scenario where the performance needs of the target application can be met with dedicated circuits which balance *specialization* and *reconfiguration*. Finally, it is also a significant stepping stone for future self-adaptive systems.

Appendix A

External Memory Access for Loop Pipelined Multi-Row Accelerators

The implementations presented in the main body of this thesis relied on system architectures containing only local data memories. This simplified the design of the memory sharing mechanism that allows the accelerator to access data memory. This local-memory-based scenario is fully functional and useful for embedded applications which do not need to rely on external data memory. However, some applications may require for data to reside in external memory, either due to the volume of the data to be handled or to make it possible for other peripherals to easily access data as well, for instance. The main objective of this work was to focus on transparent binary acceleration and mechanisms. However, it is important to demonstrate that the approach is viable for such a scenario. Therefore, this appendix presents an analysis of the performance and design of an accelerator design coupled to a customizable dual-port cache, which interfaces with an external memory controller to access data stored off-chip.

The accelerator is an extension of the design shown in Chapter 5. To cope with the variable memory access latency caused by the external memory, the *load/store* Functional Units (FUs) have been adapted into pipelined units, and the rows of FUs now support multi-cycle execution. To reduce the impact of longer memory access latencies, this implementation also exploits loop pipelining by concurrently enabling multiple rows. Finally, an accelerator instance may also contain synchronization logic which allows it to operate at a higher frequency than that of the remaining system. Likewise, if the accelerator complexity lowers its maximum operating frequency relative to the baseline system, this allows for the remaining software execution to be unaffected.

The accelerator architecture is shown in Section A.1, and the developed configurable dual-port cache in Section A.2. Section A.3 presents the experimental results, and Section A.4 concludes this annex.

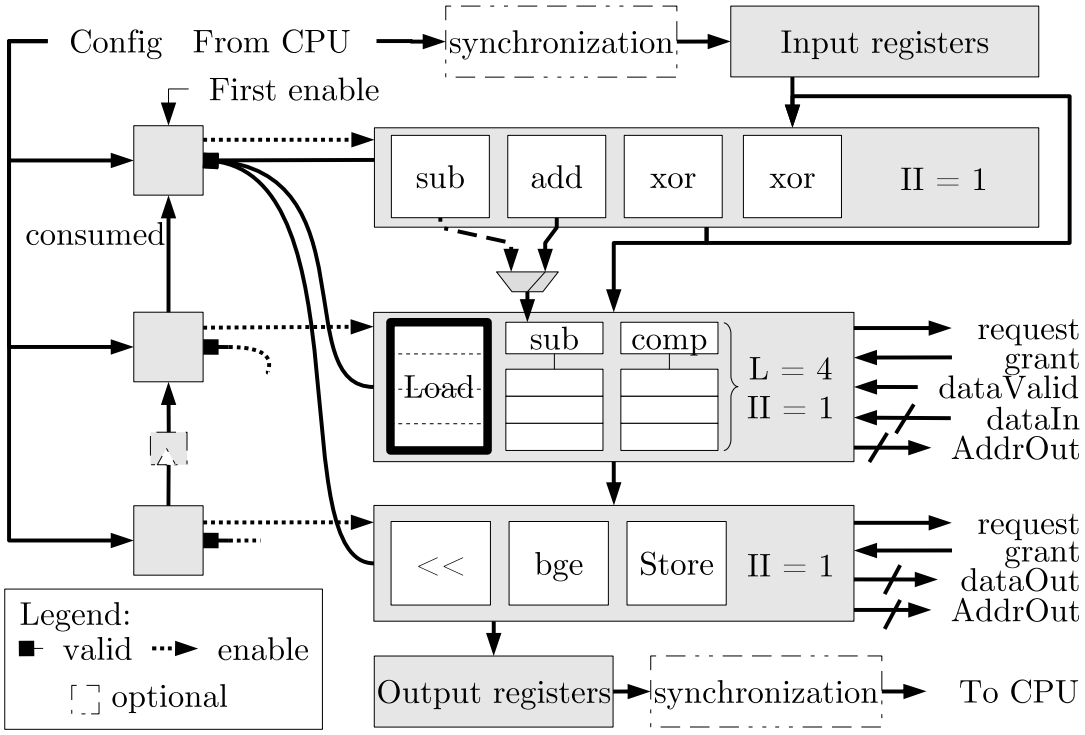


Figure A.1: Pipelined 2D accelerator adapted for higher memory access latencies

A.1 Accelerator Architecture

Figure A.1 shows a synthetic example of the accelerator architecture, which is similar to the multi-row design from Chapters 4 and 5. Like all implementations, each accelerator instance is based on customizing the number, type and location of FUs and their interconnections. The two different features are a more sophisticated backwards interconnection capability between FUs, which allows for direct connections from any row to any other preceding it, and the ability to simultaneously activate multiple rows of FUs.

A.1.1 Structure

The accelerator is composed by rows of FUs which execute in parallel. Independent operations are translated into FUs on the same row, and generally each row corresponds to a CDFG level. Each FU input is fed by a customized multiplexer. This multi-row accelerator design has the capability to support backward connections from any row to any preceding row. For example, the first row feeds data back into itself. In the previous multi-row design, the data had to be propagated to the bottom of the array, and only then fed back. This allows the implementation of the exact, and shortest possible, backwards node connectivity present in the source CDFGs. Together with the capability of simultaneously activating multiple rows, this allows for loop pipelining. This is implemented via the per-row control modules shown on the left-hand side. Memory accesses are handled by a Memory Access Manager (MAM) unit (not shown in Fig. A.1), which interfaces with all rows containing memory operations, as shown on the right-hand side.

The array supports all integer (including division by a constant), comparison, logic operations, and also *load/store* operations. Floating-point operations and integer division by a non-constant divider are not supported. All units are single-cycle, except the constant integer division (2 clock cycles), the multiplication (3 clock cycles), and the *load* operations (4 clock cycles at the least). The latency of *load/store* operations depends on the memory access contention (which is a function of the number of memory units on the array and how frequently they are activated). The behaviour of the cache also influences the execution of these units.

The *load* FUs were re-designed into a four cycle operation, based on the number of cycles required to fetch a datum from the cache when there is a tag hit. These FUs are pipelined, so registers are added at the output of other FUs on the same row as a *load* unit, to synchronize data. Likewise, when a row contains a three cycle multiplication unit, registers are added to the outputs of other FUs. The multiplication unit is fully pipelined, so the resulting inter-row pipelining allows for producing data every cycle. The *store* operations may buffer up to one datum if no memory ports are available. These units only introduce latency if they are activated again before their buffered datum is written to memory.

The synchronization logic, which is automatically instantiated if the system and accelerator clocks differ, is placed at the input/output register interfaces and also at the memory interface to the cache. That is, the cache and accelerator operate at different frequencies, as this facilitates the interface of the cache to the external memory controller port.

A.1.2 Execution

The accelerator is invoked by the same migration mechanism used in all other implementations. When a Megablock start address is detected, the *injector* sends a single configuration word to the accelerator, which sets the number of expected operands and global multiplexer context. If the accelerated loop contains one or more *store* operations, the respective Communication Routine (CR) includes a loop to invalidate the MicroBlaze's data cache. The accelerator's own data cache is invalidated in a single cycle prior to execution.

Any number of *exit* FUs on the array (which represent the *branch* operations of the MicroBlaze instruction set), keep track of the conditions which end the execution. For multi-configuration accelerators, the number of rows which needs to be activated to complete an iteration is equal to each accelerated CDFG's Critical Path Length (CPL), i.e., depth. The output registers can be fed by any FU in any row, just as any FU in any row can be fed by any other.

The per-row control modules on the left-hand side of Fig. A.1 enable their respective rows based on all *valid* signals they receive from all rows. For example, the control for the first row issues an *enable* once a *valid* signal is issued by that same row. Likewise, the second row is enabled once a *valid* signal is issued by the first row. However, a row may produce data consumed by multiple rows. For instance, the first row in Fig. A.1 produces data consumed by itself and by the second row. This means that the first row cannot produce new data before the existing data is read by all consumer rows. Thus, a *consumed* signal is asserted by every row after it activates.

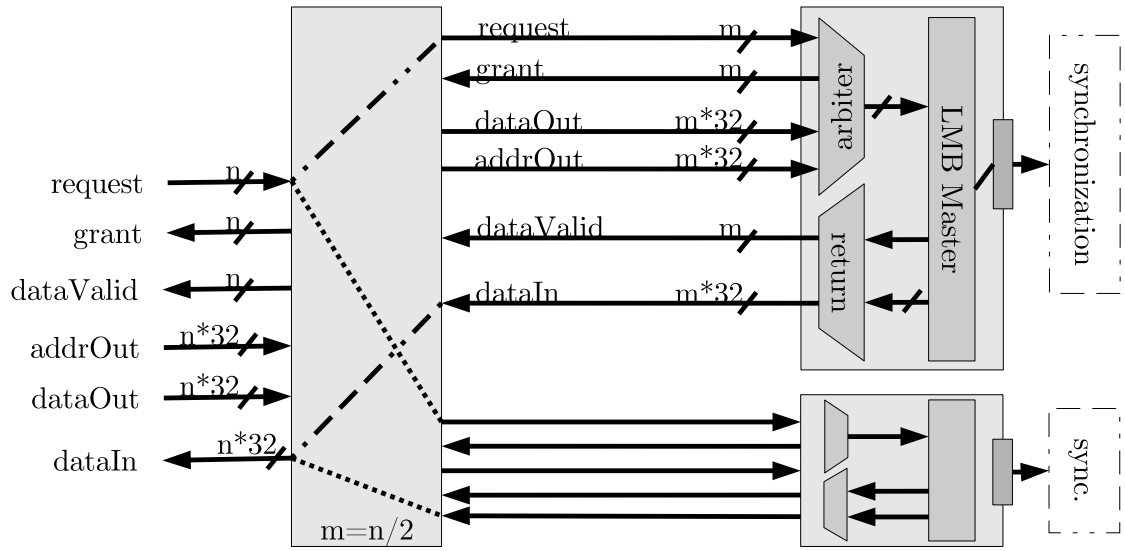


Figure A.2: Memory Access Manager (MAM) for this implementation. Each memory port (right-hand side) manages half of the *load/store* units (*m* is the total number of units on the array).

This kind of control is easy to generate on a per-accelerator basis, implementing each CDFG's different inter-row data dependencies, which sometimes include several backwards connections. Also this control allows for the accelerator to cope with the variable latency of the *load/store* operations. For clarity, the memory port logic is omitted in Fig. A.1, but is nearly identical to the previous multi-row design. The loop pipelining implemented by concurrent row enabling increases performance by exploiting instruction parallelism further, and also by mitigating the impact of external memory access latency, as operations executed while a given row idles waiting for *load/store* operations to complete.

A.1.3 Memory Access

Like previous implementations, *load/store* units receive operands from any other FUs. This means accesses can be to arbitrary addresses. The access requests are handled by a Memory Access Manager (MAM) unit, shown in Fig. A.2, which is similar to the one presented in Chapter 5. As before, memory operations are byte-enabled.

The *load/store* units assert request signals and stall execution if their operation does not complete within a given number of cycles. This implementation relies on single-cycle arbitration logic, and each LMB port has its own arbiter. Both ports receive all signals from all *load/store* units, and the arbiters make mutually exclusive selections. After granting access to a given FU placed on row *n*, an arbiter will only grant it access again after all *load/store* units located on all rows after row *n* are handled as well, to preserve memory access order. To reduce the complexity of the arbitration logic, each of the memory ports may handle only half of the *load/store* FUs in the array.

A.2 Configurable Dual-Port Cache

The dual-port cache used is a custom module which can be configured in terms of line size, block size, and number of blocks. It implements a direct-mapped, write-through no-allocate policy. The cache has the following features: data already present in the cache can be accessed by one port while another port waits on a block load; reads are answered as early as possible when loading a block (i.e., when the request datum is fetched it is sent back to the accelerator before the entire bock is read); both ports can read data from the same block (even while the block is being loaded); data can be written by the accelerator into a block being loaded without loss of coherency.

The accelerator can either connect to the local data BRAMs or to the cache, since the cache relies on the same interface signals. To access the Multi-Port Memory Controller (MPMC) used, the cache has two ports, equal to those of the MicroBlaze. One is exclusively for reading complete cache lines, and the other exclusively for single datum writes (i.e., non-burst). The MPMC uses round robin arbitration by default to manage its multiple master ports, which caused some accelerator-issued accesses to be performed out of order. This is why the cache is write-through and also why the port used for writes is given priority.

The cache operates on the same clock domain as the accelerator, so no synchronization logic exists between the two modules. An additional register stage may be introduced (manually) however, if the joint accelerator and cache complexity introduces long critical paths. Introducing this stage increases the memory access latency, which does have impact on accelerator performance. If the cache operates at a different frequency than the system, the synchronization logic is placed at the interface with the MPMC.

There are some issues which, however, are not necessarily limitations neither of the proposed acceleration approach nor of the accelerator design: namely: the cache has a considerable resource cost; it was designed to interface specifically with the MPMC present in the utilized Spartan-6 FGPA; since only one cache link is used for reads, only one block can be loaded into the cache at the same time; and it is overall too complex relative to the performance it delivers. Keep in mind however that the cache was developed simply to provide a functional external memory access for the accelerator for validation purposes.

A.3 Experimental Evaluation

Figure A.3 shows the system architecture used to evaluate this accelerator design, and to demonstrate that the migration and acceleration approach functions even under an external data memory scenario. The accelerator retains its LMB based interfaces, which couple to the dual-port cache. The cache has two Xilinx Cache Link (XCL) interfaces (equal to the MicroBlaze) to the multi-ported memory controller in order to access external DDR memory, which holds only data arrays.

The MicroBlaze uses local memories for program code, and is configured with a 256-byte write-through data cache, with a line size of 8 words, and no instruction cache (as all instructions reside in local memory). The accelerator's cache is instantiated with the same total and line sizes.

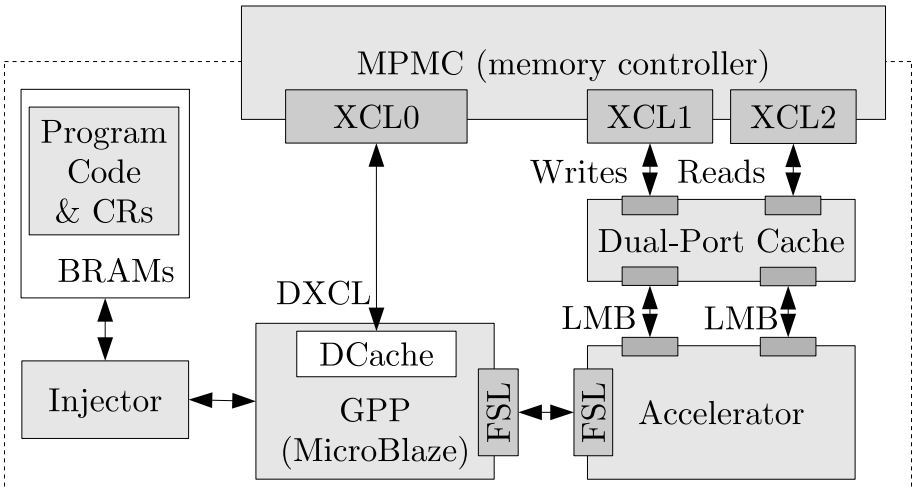


Figure A.3: System architecture for validation of external data memory access by the accelerator

The buffer stage is placed between the cache and accelerator to remove long critical paths. The system, i.e., the MicroBlaze, operates at 83 MHz for all cases, and the accelerator's operating frequency varies per case. The implementation platform was a Digilent Atlys board, with a Spartan-6 LX45 FPGA, and a 128 MB DDR2 memory.

The static data to process is placed on the external memory by copying it from on-board non-volatile flash memory at boot. A custom section in the linker script and a *section* attribute are used when declaring the *C* arrays which contain this data, so the compiler knows their address. The accelerator's data cache is reset before the accelerator executes, since its contents are most likely invalid due to MicroBlaze execution prior to each accelerator call. Likewise, the MicroBlaze's own data cache is invalidated while the accelerator executes, as part of the CR. This invalidation is only performed if the loop executed on the accelerator contains any store operations.

A total of 12 integer kernels were used to validate the correct functioning of the system. The kernels originate from the PowerStone [MMC00], Texas's IMGLIB library [Tex], and WCET [GBEL10] benchmark suites, and from other sources [War02, Jen97]. The *dotprod* benchmark is a simple synthetic kernel. A large 3D path planning application, *griditerate*, was also used [CDP+11]. For bitstream generation and synthesis, Xilinx's EDK 14.6 was used. For compilation, version 4.6.4 of *mb-gcc* was used, along with the *-O2* flag and additional flags to enable the barrel-shift, integer multiplication, and comparison operations.

The following section discusses general performance aspects, and comments on the effects of memory access contention and loop pipelining on performance. Section A.3.4 presents resource requirements and operating frequencies for the generated accelerators.

A.3.1 General Aspects

Table A.1 shows the characteristics of the extracted Megablocks and the generated accelerators. The third and fourth columns show the average number of instructions in the traces, and the minimum possible Initiation Interval (II) of the respective CDFGs, *Min*II. Completing an iteration in

Table A.1: Megablock and characteristics of generated accelerators

ID	Benchmark	Avg. #Inst.	Min. II	Max. IPC	# FUs	# Loads	# Stores	# Rows
i1	blit	10.0	1	10.0	18	1	2	4
i2	bobhash	10.0	5	2.0	11	1	0	8
i3	checkbits	63.0	1	31.5	59	1	1	19
i4	dotprod	12.0	2	12.0	10	2	0	5
i5	fft	39.0	1	19.5	48	6	4	10
i6	gouraud	19.0	1	19.0	16	0	1	6
i7	perimeter	22.0	1	22.0	28	5	1	10
i8	poparray1	27.0	1	27.0	22	1	0	18
i9	griditerate	120.0	5	24.0	121	22	11	16
i10	g3fax	5.4	2	2.7	22	2	0	7
i11	edn	15.3	2	7.6	33	4	0	9
i12	fir	9.0	1	9.0	11	2	0	4
	mean	29.3	2	15.5	33.3	3.9	1.7	9.7

*Min*II clock cycles leads to the highest number of executed Instructions per Clock Cycle (IPC), *Max*.IPC, as is shown in the fifth column. One Megablock was accelerated for all cases, except for *i1*, *i10*, and *i11* (2, 4, and 2 Megablocks respectively). The mean is arithmetic for all cases.

Most of the used kernels in this evaluation were also employed in Chapter 5; *i4* and *i9* are the only exceptions. The later was included here as it contains a deeply nested loop (four levels of nesting) which was considered to be a good target for aggressive loop pipelining due to the number of instructions in the resulting Megablock. Since it also contained a large number of memory operations, it was also a useful stress test for the accelerator's execution model and especially to evaluate the impact of the external memory access latency.

Despite the large number of instructions for *i9*, the average number of instructions per Megablock in this set is approximately 0.70× the average number of instructions for the comparable implementation in Chapter 5 (i.e., memory access capable multi-row accelerator). The traces have fewer operations, so there is less parallelism to exploit per iteration. But since loop pipelining is enabled, additional intra-iteration parallelism is also exploited. That is, it might not pay off to migrate small loops to the accelerator when exploring inter-iteration ILP alone, which is addressed by also resorting to loop pipelining. For example, consider that the benchmarks used here are a subset of the cases evaluated in Chapter 5 (exception for *i4* and *i9*). For these cases, the maximum IPC resulting from intra-iteration parallelism alone is, on average, only 2.03. Considering both intra- and inter-iteration parallelism the IPC becomes 14.8.

However, loop pipelining also means that more memory operations are issued per cycle, meaning more contention despite the fact that the Megablocks in this evaluation have fewer memory operations per iteration than the entire set in Chapter 5. As a consequence of memory access contention and latency, the IPC actually achieved on the accelerator, II_{HW}, is lower than the ideal *Max*.IPC. For execution on the accelerator to achieve this maximum, it would have to be capable

Table A.2: Performance metrics and speedups for the tested benchmarks

ID	II_{SW}	IPC_{SW}	II_{HW}	IPC_{HW}	S_{Kernel}	$S_{Overall}$	$S_{NoOverhead}$	$S_{UpperBound}$
i1	15.0	0.7	7.7	1.3	1.94	1.91	1.93	8.39
i2	15.8	0.6	7.3	1.4	2.14	2.10	2.11	2.04
i3	68.0	0.9	8.2	7.7	8.06	7.58	7.83	18.31
i4	16.6	0.7	9.1	1.3	1.76	1.65	1.72	5.30
i5	121.2	0.3	100.1	0.4	1.19	0.83	1.04	0.95
i6	21.0	0.9	3.1	6.1	6.65	6.39	6.45	14.12
i7	27.5	0.8	14.2	1.7	1.91	1.87	1.91	12.64
i8	35.7	0.7	6.0	4.5	5.21	3.86	4.26	7.52
i9	278.9	0.4	245.7	0.5	1.16	1.15	1.24	7.94
i10	10.0	0.5	7.1	0.8	1.02	0.93	1.13	1.24
i11	23.3	0.6	16.5	0.9	1.22	1.17	1.35	2.45
i12	17.8	0.5	17.6	0.5	0.93	0.96	1.03	2.21
mean	54.0	0.7	37.0	2.2	2.07	1.91	2.06	4.69

of executing all memory operations in an iteration within *Min*II clock cycles (assuming a scenario where the cache always responds in one clock cycle). That is, the accelerator would need to be able to execute an average of 2.3 memory operations in parallel, each with a latency of one clock cycle. The deviation of this average is high however, with *i5*, *i7* and *i9* requiring 5.0, 6.0 and 6.6 parallel accesses per cycle to achieve execution at the minimum II, respectively.

A.3.2 Performance

Table A.2 compares the performance of software and accelerator-enabled execution. The first two columns show how many clock cycles are required to complete an iteration of the accelerated Megablock through software (II_{SW}), and the resulting instructions executed per clock cycle (IPC_{SW}), which are computed given the number of accelerated instructions and the II. The next two columns shows the same values for accelerator-enabled execution. The last four columns show the *kernel* speedup (S_{Kernel}) (i.e., the speedup of the accelerated portion), the speedup of the overall application ($S_{Overall}$) the speedup value which would be achieved if CR overhead were eliminated ($S_{NoOverhead}$) and finally a speedup upperbound ($S_{UpperBound}$) computed considering a scenario where memory access latency was minimal (i.e., one clock cycle), and memory bandwidth was unlimited. That is, this upper bound is the potential speedup if iterations could be completed at rate of one per *Min*II clock cycles (shown in the fourth column of Table A.1). The mean is geometric for speedups and arithmetic for all other columns.

A.3.2.1 Instruction per Clock Cycle and the Effects of Memory Latency

In the experimental evaluations in the main document, the equivalent IPC_{SW} (i.e, the number of instructions the MicroBlaze executes per cycle) was computed by taking into account the number of instructions in the accelerated trace and the latency of each instruction. But since the data are

now on external memory, the IPC_{SW} for this evaluation was computed using the measured number of clock cycles required to execute all Megablock iterations, on the number of executed iterations, and on the number of instructions in the Megablock. The IPC_{HW} is computed in the same manner. Note that although the accelerator operates at a different clock frequency than the MicroBlaze for some cases, all IPC values are computed using clock cycle counts of the system clock.

The accelerator achieves a higher IPC for all cases, although for some, like *i12*, the increase is minimal. The IPC_{HW} value is lower than the ideal IPC almost exclusively due to the memory access latency, but also decreases (very minimally) since it was computed taking into account the number of processor equivalent cycles during which the accelerator executes (which also includes a small overhead and synchronization cycles).

The Megablocks which contain the least number of memory operations are those for which the achieved IPC is closest to the maximum. This happens for *i2*, *i3*, and *i6*, for which the average accelerator IPC is 0.41× the respective average ideal IPC. For the entire set, the achieved IPC is, on average, 0.18× the optimal. Although *i12* also contains few memory operations, it suffers from cache latency due to the access pattern of the operations, which constantly forces the cache to discard freshly loaded blocks. As a result, the resulting IPC is very low relative to the optimal, and there is no gain relative to software only execution.

From the measurements its possible to compute how many clock cycles are added to the execution of a single iteration, relative to a scenario with minimum memory latency. This was computed for both the MicroBlaze and the accelerator. For the MicroBlaze, accessing data residing on external memory adds an average of 24 clock cycles relative to a local memory scenario, where the average number of clock cycles required to complete an iteration would be 30 (this was estimated considering the number of instructions in each trace and their latencies). For the accelerator, 35 clock cycles are added per trace iteration due to memory access, over the ideal *Min*II, whose average is 2 as shown in Table A.1.

Memory accesses performed by the accelerator are more costly since: (i) even though there are two ports between cache and accelerator there is only one port through which data is fetched into the cache from external memory; (ii) the 2-clock-cycle cache latency makes memory accesses more costly for the accelerator. However, the accelerator does not necessarily suffer from twice the memory access latency, because (i) the dual-cache port allows for concurrent accesses, and (ii) the accelerator operates at higher clock frequencies relative to the system clock for all cases.

The purpose of exploiting data parallelism via two memory ports was also to mitigate the access latency. This evaluation also counted how many accesses were performed per accelerator port, since the handling of *load* and *store* FUs relies on a runtime arbitration heuristic. Considering all accelerated Megablocks, one port performs 502 accesses per accelerator call, and the other 329 accesses per call. This demonstrates that considerable data access parallelism was exploited.

A.3.2.2 Speedups

Despite the penalty of external memory access, the geometric mean speedup on the accelerator is 1.91×. However, for 3 cases the use of the accelerator resulted in a slowdown. For *i5*, this is

due to the low number of iterations performed for some of the accelerator calls. The accelerated Megablock for this case also contains the largest number of memory operations (apart from *i9*). But the most influential factor for the slowdown in this case is the number of accelerator calls that take place without any useful work being completed, i.e., not a single iteration is performed on the accelerator. For *i15*, 127 accelerator calls out of a total of 128 terminate without having executed a single iteration. For *i10* and *i12*, the number of instructions in the accelerated traces is small, and coupled with the number of iterations per call is 57 and 32, respectively. For these cases, the overhead has large impact on the total migration time, negating any small parallelism gains.

There are two factors which influence the application speedup, $S_{Overall}$, shown in Table A.2: the achieved IPC due to parallelism exploitation and the accelerator execution model, and the fact that the accelerator operates at higher clock frequencies (the average being 128 MHz). To disambiguate the effect of these two aspects, we can compute the time the accelerator would require to execute the Megablocks if it operated at the system frequency, 83 MHz. For this scenario, the geometric mean speedup is 1.13×, with slowdowns occurring for 5 benchmarks (*i5* and *i9* to *i12*). The minimum latency of two clock cycles for cache access now becomes very detrimental, as it is no longer mitigated by a higher clock frequency. The accelerator's two parallel memory ports effectively operate as a single port with a latency of one clock cycle.

The benchmarks used here are a subset of the cases evaluated in Chapter 5 (except for *i4* and *i9*). The implementation in Chapter 5 relied on local memories for the data and the accelerator did not exploit loop pipelining. As a result, the geometric mean speedup was of 1.62×, versus the 2.03× achieved in this implementation, despite the external memory access latency.

Finally, the geometric mean for $S_{Overall}$ is only 41 % of the geometric mean of $S_{UpperBound}$. Using instead the values of $S_{NoOverhead}$ (which discards CR overhead), the geometric mean is 53 % of the geometric mean of the upper bound. Since $S_{NoOverhead}$ discards CR overhead relative to $S_{Overall}$, and $S_{UpperBound}$ is the estimation of speedup without memory access latency but with CR overhead, this shows that it is the limited memory bandwidth, and the high access latency that greatly decrease performance with this accelerator.

A.3.3 Communication and Cache Invalidation Overhead

The overhead introduced for this implementation includes the transfer of operands and results between the accelerator and the MicroBlaze during Communication Routine (CR) execution, and the additional overhead due to the invalidation of the MicroBlaze's cache during accelerator execution. For all benchmarks, the overhead represents 6.02 % of the migration time (i.e., CR and accelerator execution, plus a final iteration in software).

The cache is invalidated by a small loop which is part of the CR. While the accelerator's cache can be invalidated in a single cycle, the MicroBlaze only allows for selective invalidation of a specific cache line. To invalidate the 256-byte cache used in this implementation, a total of 192 clock cycles are required. This is considerable given that a CR without this cache invalidation step contains an average of 22 instructions and requires approximately the same number of cycles to execute (since the instructions reside in local memory and nearly all operations have a single cycle

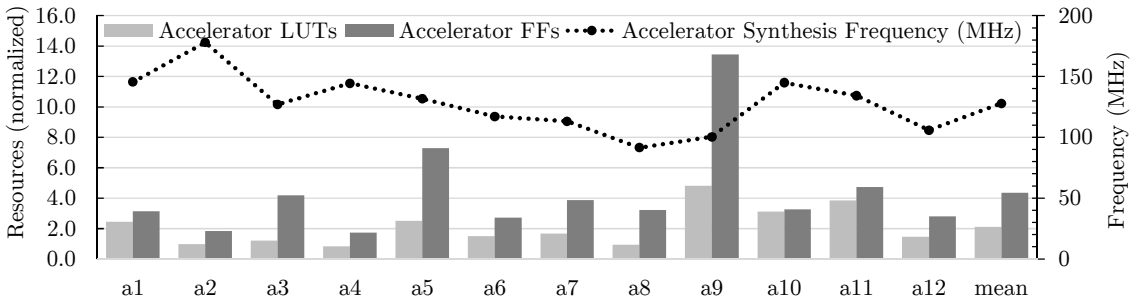


Figure A.4: Resource requirements and synthesis frequency of the generated accelerators with a multiple row architecture capable of loop pipelining

latency). However, since the cache is invalidated during accelerator execution, the invalidation time is hidden if the accelerator executes enough Megablock iterations. This is not the case for one of the Megablocks implemented for *i10*, for instance, where an average of 2.2 iterations executed per call mean that the accelerator finishes executing before the MicroBlaze's data cache has been completely invalidated.

A.3.4 Resource Requirements and Operating Frequency

Figure A.4 shows the resource requirements of the generated accelerators, as well as the synthesis frequency. For these benchmarks, a single MicroBlaze requires 1615 LUTs and 1213 FFs. This means the average accelerator requires 2.21× and 4.35× the LUTs and FFs of a MicroBlaze, respectively. Ten out of the 12 benchmarks used in this implementation were also used in the evaluation in Chapter 5. For this subset, the average number of required LUTs is very similar, but the number of required FFs now increases by 2.21×. This increase can be attributed to the additional per-row control logic, and especially to the need to register *passthrough* units to synchronize data for loop pipelining. The entire system requires an average of 8696 LUTs and 9555 FFs, meaning that the average accelerator represents 37 % of LUTs and 51 % of FFs.

The average synthesis frequency of the accelerator is 128 MHz, and it always operates at a frequency higher than the 83 MHz baseline. This evaluation attempted to increase the operating frequency of the system (and therefore the MicroBlaze), to achieve a more competitive baseline. However, given the size of the device, and all the peripherals needed (i.e, the occupied area of the device, accelerator included), this was not possible. The average operating frequency of the accelerator was of 110 MHz, which was possible by adding the additional register stage between the accelerator and the dual-port cache. Otherwise, the joint complexity would result in longer critical paths, which in some instances lowered the accelerator's operating frequency below that of the baseline. The lowest synthesis and operating frequencies occurs for *i8*, due to the control logic which dynamically enables stages based on inter-row data dependencies. Nearly all critical paths are related to the signals between the MAM and the stage control modules.

Finally, regarding the cache, it requires 1595 LUTs and 1103 FFs when set for 256 bytes and a block size of 8 words, and the synthesis frequency is 139 MHz. The frequency is very similar for

all sizes up to 4096 bytes, and for all block sizes up to 16 words. The only observable decrease to 75 MHz is for a block size of 2 words and a total size of 4096 bytes, due to the larger tag memory and number of comparators required. The resulting resource requirements are also the highest for all combinations of cache parameter values: 15536 LUTs and 9061 FFs.

The cache parameters that ensure optimal performance largely depend on the pattern of memory accesses performed by the accelerator. Although this is an analysis that could be performed based on the simulation step that extracts the Megablocks, it is not within the scope of this work. However, the resource requirements of even the smaller cache instances are still considerable relative to those of the average accelerator instance. As a whole, the accelerator and cache combined require on average 3.1× the LUTs and and 5.3× the FFs of a MicroBlaze containing its own 256-byte data cache.

A.4 Concluding Remarks

This appendix presented a brief proof-of-concept for an accelerator augmented system capable of external memory access. It was never expected that this implementation outperformed the cache-enabled MicroBlaze; the objective was to demonstrate that the Megablock translation and execution migration mechanisms are also applicable in this scenario.

To allow for the data memory to be shared between the MicroBlaze and the accelerator, each of these modules resorts to its own data cache. The MicroBlaze's data cache is instantiated conditionally based on a configurable parameter, and the accelerator cache is a custom designed module. The accelerator capability to perform two concurrent memory accesses is preserved since the cache is also dual-ported. This system architecture introduces the need to occasionally invalidate either the MicroBlaze's or the accelerator's data cache, depending on which module has executed and which has expired data. Additionally, the multi-row accelerator design is capable of exploiting loop pipelining by simultaneously enabling multiple rows.

However, the evaluation found that the the memory access latency, and especially access contention, have a great impact on performance. This is because the execution model of the accelerator is a direct translation of CDFGs. For CDFGs without memory operations, this translation method works efficiently: each type of FU is self-contained, with a low (usually one clock cycle) and constant latency. Supporting backward inter-row connectivity allows for exploiting loop pipelining by implementing the recursion in the CDFGs, but the translation process does not take into account that the memory ports are a limited resource, both in terms of the number of ports and in terms of amount of accesses per cycle. As a result, the execution stalls frequently while memory accesses are handled. The fact that iterations are overlapped by loop pipelining increases the number of memory accesses per cycle. This causes an under-utilization of the instantiated resources due to the stall time while accessing the cache. Also, the cache architecture used in this evaluation does not benefit from multi-way associativity or pre-fetching. The memory access pattern of the accelerated traces would need to be analysed to implement any kind of optimization pertaining to this point, but this falls out of the scope of this work.

As a result, accelerator-enabled execution achieves a geometric mean speedup of 1.91× over MicroBlaze-only execution, whereas the potential upperbound possible with loop pipelining is 4.69×. Additionally, the average accelerator is costly in terms of resources; the average number of required LUTs and FFs is 2.11× and 4.35× the number required by a MicroBlaze, which is a considerable cost, and also higher than the average for the non-pipelined multi-row architecture.

It was this insight regarding the impact of memory access latency, and especially the contention which occurs when loop pipelining is exploited, that guided the accelerator design into the architecture presented in Chapters 6 and 7, which implements loop pipelining more efficiently and requires less resources.

References

[AD11] José Carlos Alves and Pedro C. Diniz. Custom FPGA-Based Micro-Architecture for Streaming Computing. In *2011 VII Southern Conference on Programmable Logic (SPL)*, pages 51–56, April 2011.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 177–189, New York, NY, USA, 1983. ACM.

[AMD] AMD. APU 101: All about AMD Fusion Accelerated Processing Units. `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/apu101.pdf`. Accessed on 4th January 2016.

[AMD13] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 51:1–51:10, New York, NY, USA, 2013. ACM.

[APTD11] Giovanni Ansaloni, Laura Pozzi, Kazuyuki Tanimura, and Nikil Dutt. Slack-aware scheduling on Coarse Grained Reconfigurable Arrays. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2011.

[ATPD12] Giovanni Ansaloni, Kazuyuki Tanimura, Laura Pozzi, and Nikil Dutt. Integrated Kernel Partitioning and Scheduling for Coarse-Grained Reconfigurable Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(12):1803–1816, December 2012.

[BC10] João Bispo and João M.P. Cardoso. On Identifying Segments of Traces for Dynamic Compilation. In *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pages 263–266, August 2010.

[BDM+72] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[BFD08] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs. In *17th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 178–192, Berlin, Heidelberg, 2008. Springer-Verlag.

[BFM+07] M. Boden, T. Fiebig, T. Meissner, S. Rulke, and J. Becker. High-Level Synthesis of HW Tasks Targeting Run-Time Reconfigurable FPGAs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1 –8, March 2007.

[BGDN03] Nikhil Bansal, Sumit Gupta, Nikil Dutt, and Alexandru Nicolau. Analysis of the Performance of Coarse-Grain Reconfigurable Architectures with Different Processing Element Configurations. In *Workshop on Application Specific Processors, held in conjunction with the International Symposium on Microarchitecture (MICRO)*, 2003.

[Bis12] João Carlos Viegas Martins Bispo. *Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units*. PhD thesis, Instituto Superior Técnico, 2012.

[Bis15] João Bispo. Megablock Extractor for MicroBlaze v0.7.14. `https://sites.google.com/site/specsfeup/`, February 2015. Accessed 21st December 2015.

[BKT12] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go Ahead: A partial reconfiguration framework. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, 2012.

[BPCF11] João Bispo, Nuno Paulino, João M. P. Cardoso, and João Canas Ferreira. From Instruction Traces to Specialized Reconfigurable Arrays. In Peter M. Athanas, Jürgen Becker, and René Cumplido, editors, *ReConFig*, pages 386–391. IEEE Computer Society, 2011.

[BPCF12] João Bispo, Nuno Paulino, João M.P. Cardoso, and João Canas Ferreira. Generation of Coarse-Grained Reconfigurable Processing Units for Binary Acceleration. *VII Jornadas sobre Sistemas Reconfiguráveis*, pages 11–19, February 2012.

[BPCF13] João Bispo, Nuno Paulino, João M. P. Cardoso, and João Canas Ferreira. Transparent Runtime Migration of Loop-Based Traces of Processor Instructions to Reconfigurable Processing Units. *International Journal of Reconfigurable Computing*, 2013.

[BPFC13] João Bispo, Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Transparent Trace-Based Binary Acceleration for Reconfigurable HW/SW Systems. *IEEE Transactions on Industrial Informatics*, 9(3):1625–1634, August 2013.

[BRGC08] A.C.S. Beck, M.B. Rutzig, G. Gaydadjiev, and L. Carro. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In *Design, Automation and Test in Europe (DATE)*, pages 1208–1213, March 2008.

[But07] M. Butts. Synchronization Through Communication in a Massively Parallel Processor Array. *IEEE Micro*, 27(5):32–40, September 2007.

[Cal] Calypto. Catapult Overview. `http://calypto.com/en/products/catapult/overview/`. Accessed on 29th December 2015.

[CBC+05] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In *Proceedings 32nd International Symposium on Computer Architecture (ISCA)*, pages 272–283, June 2005.

[CDP+11] João M. P. Cardoso, Pedro C. Diniz, Zlatko Petrov, Koen Bertels, Michael Hübner, Hans Someren, Fernando Gonçalves, José Gabriel F. Coutinho, George A. Constantinides, Bryan Olivier, Wayne Luk, Juergen Becker, Georgi Kuzmanov, Florian Thoma, Lars Braun, Matthias Kühnle, Razvan Nane, Vlad Mihai Sima, Kamil Krátký, José Carlos Alves, and João Canas Ferreira. *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*. Springer, New York, NY, 2011.

[CFF+99] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, 1999.

[CH00] Katherine Compton and Scott Hauck. An Introduction to Reconfigurable Computing. *IEEE Computer*, 2000.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys (CSUR)*, 34(2):171–210, June 2002.

[CHM08] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized Execution Accelerator for Loops. In *35th International Symposium on Computer Architecture (ISCA)*, pages 389–400, June 2008.

[Cho11] Kiyoung Choi. Coarse-Grained Reconfigurable Array: Architecture and Application Mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46, 2011.

[CKP+04] N. Clark, M. Kudlur, Hyunchul Park, S. Mahlke, and K. Flautner. Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *37th International Symposium on Microarchitecture (MICRO)*, pages 30–40, Washington, DC, USA, December 2004. IEEE Computer Society.

[CM14] Liang Chen and Tulika Mitra. Graph Minor Approach for Application Mapping on CGRAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21:1–21:25, September 2014.

[CZS+08] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker. A multi-platform controller allowing for maximum Dynamic Partial Reconfiguration throughput. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 535–538, 2008.

[DGG05] G. Dimitroulakos, M.D. Galanis, and C.E. Goutis. Alleviating the Data Memory Bandwidth Bottleneck in Coarse-Grained Reconfigurable Arrays. In *16th IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, pages 161–168, July 2005.

[Dun13] R. Dunkley. Supporting a Wide Variety of Communication Protocols Using Dynamic Partial Reconfiguration. *IEEE Instrumentation Measurement Magazine*, 16(4):26–32, August 2013.

[EBIH12] A. Ebrahim, K. Benkrid, X. Iturbe, and C. Hong. A novel high-performance fault-tolerant ICAP controller. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 259–263, June 2012.

[EBSA+12] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power Limitations and Dark Silicon Challenge the Future of Multicore. *ACM Transactions on Computer Systems (TOCS)*, 30(3):11:1–11:27, August 2012.

[ECF96] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL)*, pages 126–135, London, UK, UK, 1996. Springer-Verlag.

[EEM15] EEMBC - The Embedded Microprocessor Benchmark Consortium. CoreMark-Pro. `http://www.eembc.org/coremark/index.php?b=pro`, 2015. Accessed 1 January 2016.

[ESSA00] J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici. Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration. In *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 165–174, 2000.

[Est02] G. Estrin. Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer. *IEEE Annals of the History of Computing*, 24(4):3–9, October 2002.

[FDP+14] R. Ferreira, W. Denver, M. Pereira, J. Quadros, L. Carro, and S. Wong. A Run-Time Modulo Scheduling by using a Binary Translation Mechanism. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 75–82, July 2014.

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[FISS12] M. Feilen, M. Ihmig, C. Schwarzbauer, and W. Stechele. Efficient DVB-T2 decoding accelerator design by time-multiplexing FPGA resources. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 75–82, August 2012.

[FPKM08] Kevin Fan, Hyun hul Park, Manjunath Kudlur, and S ott Mahlke. Modulo Scheduling for Highly Customized Datapaths to Increase Hardware Reusability. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 124–133, New York, NY, USA, 2008. ACM.

[GAA+07] V. Groza, R. Abielmona, M.H. Assaf, M. Elbadri, M. El-Kadri, and A. Khalaf. A Self-Reconfigurable Platform for Built-In Self-Test Applications. *IEEE Transactions on Instrumentation and Measurement*, 56(4):1307 –1315, August 2007.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks - Past, Present and Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[GC08] H. Gu and S. Chen. Partial Reconfiguration Bitstream Compression for Virtex FPGAs. In *Congress on Image and Signal Processing (CISP)*, volume 5, pages 183–185, May 2008.

[GRV05] Ann Gordon-Ross and Frank Vahid. Frequent Loop Detection Using Efficient Non-intrusive On-Chip Hardware. *IEEE Transactions on Computers*, 54(10):1203–1215, October 2005.

[GSB+99] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, R. Reed Taylor, and R. Reed. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 28 –39, 1999.

[Har01] Reiner Hartenstein. Coarse Grain Reconfigurable Architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 564–570, 2001.

[HFHK04] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffrey P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):206–217, February 2004.

[HGNB10] M. Hubner, D. Gohringer, J. Noguera, and J. Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[HKT11] S. G. Hansen, D. Koch, and J. Torresen. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 174–180, May 2011.

[HW97] John R. Hauser and John Wawrzynek. Garp: a MIPS Processor with a Reconfigurable Coprocessor. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 12–21, 1997.

[Ima12] Nethra Imaging. Am2045 product overview. `http://www.ambric.com/products_am2045_overview.php`, 2012.

[Ins16] Texas Instruments. Fusion digital power designer. http://www.ti.com/tool/fusion_digital_power_designer, February 2016. Accessed 14 April 2016.

[Int96] Intel. Using MMX™ Instructions for Procedural Texture Mapping - Based on Perlin's Noise Function. `https://software.intel.com/sites/landingpage/legacy/mmx/MMX_App_Procedural_Texturing.pdf`, 1996. Accessed on 10th March 2016.

[Jen97] Bob Jenkins. A Hash Function for Hash Table Lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`, December 1997. Accessed on 10th March 2016.

[KHC11] Yangsu Kim, Kyuseung Han, and Kiyoung Choi. A Host-Accelerator Communication Architecture Design for Efficient Binary Acceleration. In *2011 International SoC Design Conference (ISOCC)*, pages 361–364, November 2011.

[KLMP12] Yongjoo Kim, Jongeun Lee, Toan X. Mai, and Yunheung Paek. Improving Performance of Nested Loops on Reconfigurable Array Processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):32:1–32:23, January 2012.

[KLSP11] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Memory Access Optimization in Compilation for Coarse-Grained Reconfigurable Architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(4):42:1–42:27, October 2011.

[KTB+12] D. Koch, J. Torresen, C. Beckhoff, D. Ziener, C. Dennl, V. Breuer, J. Teich, M. Feilen, and W. Stechele. Partial reconfiguration on fpgas in practice - tools and applications. In *ARCS Workshops (ARCS)*, pages 1–12, February 2012.

[LCDW15] Mingjie Lin, Shaoyi Chen, Ronald F. DeMara, and John Wawrzynek. ASTRO: Synthesizing application-specific reconfigurable hardware traces to exploit memory-level parallelism. *Microprocessors and Microsystems*, 39(7):553–564, 2015.

[LEP12] A. Lifa, P. Eles, and Z. Peng. Minimization of average execution time based on speculative FPGA configuration prefetch. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–8, December 2012.

[LFy09] Wang Lie and Wu Feng-yan. Dynamic partial reconfiguration on cognitive radio platform. In *IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, volume 4, pages 381–384, November 2009.

[Liu08] Dake Liu. Chapter 3 - DSP Architectures. In Dake Liu, editor, *Embedded DSP Processor Design*, volume 2 of *Systems on Silicon*, pages 87 – 158. Morgan Kaufmann, Burlington, 2008.

[LP13] Daniel Llamocca and Marios Pattichis. A Dynamically Reconfigurable Pixel Processor System Based on Power/Energy-Performance-Accuracy Optimization. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(3):488–502, March 2013.

[LPV10] Daniel Llamocca, Marios Pattichis, and G. Alonzo Vera. Partial reconfigurable fir filtering system using distributed arithmetic. *International Journal of Reconfigurable Computing*, 2010:4:1–4:14, February 2010.

[LV04] Roman Lysecky and Frank Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*, volume 1, pages 480 – 485 Vol.1, February 2004.

[LV09] Roman Lysecky and Frank Vahid. Design and Implementation of a MicroBlaze-Based Warp Processor. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):22:1–22:22, April 2009.

[MGZ+07] Arash Mehdizadeh, Behnam Ghavami, Morteza Saheb Zamani, Hossein Pedram, and Farhad Mehdipour. An Efficient Heterogeneous Reconfigurable Functional Unit for an Adaptive Dynamic Extensible Processor. In *International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 151–156, October 2007.

[MLM+05] Bingfeng Mei, A. Lambrechts, J.Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design Test of Computers*, 22(2):90–101, March 2005.

[MMC00] A. Malik, B. Moyer, and D. Cermak. A Lower Power Unified Cache Architecture Providing Power and Performance Flexibility. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 241–243, June 2000.

[MO98] Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *IEICE Transactions on Information and Systems E82-D*, pages 389–397, 1998.

[MS09] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18–25, July 2009.

[MSB+07] Gayatri Mehta, Justin Slander, Mustafa Baz, Brady Hunsaker, and A.K. Jones. Interconnect Customization for a Coarse-grained Reconfigurable Fabric. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[MV15] Sparsh Mittal and Jeffrey S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, 47(4):69:1–69:35, July 2015.

[Nag01] Ulrich Nageldinger. *Coarse-Grained Reconfigurable Architecture Design Space Exploration*. PhD thesis, Universität Kaiserslautern, Gottlieb-Daimler-Strasse, 67663 Kaiserslautern, Germany, 2001.

[Nat11] National Research Council. *The future of computing performance. Game over or next level?* The National Academies Press, Washington, D.C., USA, 2011.

[NMIM12] Hamid Noori, Farhad Mehdipour, Koji Inoue, and Kazuaki Murakami. Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization. *The Journal of Supercomputing*, 60(2):196–222, May 2012.

[NMM+06] H. Noori, F. Mehdipou, K. Murakami, K. Inoue, and M. SahebZamani. A Reconfigurable Functional Unit for an Adaptive Dynamic Extensible Processor. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, August 2006.

[NMM+08] Hamid Noori, Farhad Mehdipour, Kazuaki Murakami, Koji Inoue, and Morteza Saheb Zamani. An Architecture Framework for an Adaptive Extensible Processor. *The Journal of Supercomputing*, 45(3):313–340, September 2008.

[OEPM09] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence Cycle Aware Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 21–30, 2009.

[ORK+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware, February 2015.

[Pau11] Nuno Paulino. Generation of Reconfigurable Circuits from Machine Code. Master's thesis, Universidade do Porto - Faculdade de Engenharia, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal, 2011.

[PCC+14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[PCF15] Nuno Paulino, João M.P. Cardoso, and João Canas Ferreira. Transparent Binary Acceleration via Automatically Generated Reconfigurable Processing Units. *XI Jornadas sobre Sistemas Reconfiguráveis*, February 2015.

[PDH11] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, December 2011.

[PFBC15] Nuno Paulino, João Canas Ferreira, João Bispo, and João M. P. Cardoso. Transparent Acceleration of Program Execution Using Reconfigurable Hardware. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1066–1071, San Jose, CA, USA, 2015. EDA Consortium.

[PFC13] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Architecture for Transparent Binary Acceleration of Loops with Memory Accesses. In *Proceedings of the 9th International Conference on Reconfigurable Computing: Architectures, Tools, and Applications (ARC)*, pages 122–133, Berlin, Heidelberg, 2013. Springer-Verlag.

[PFC14a] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. A Reconfigurable Architecture for Binary Acceleration of Loops with Memory Accesses. *ACM Transactions on Reconfigurable Technology and Systems*, 7(4):29:1–29:20, December 2014.

[PFC14b] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 158–165, August 2014.

[PFM+08] Hyunchul Park, Kevin Fan, Scott A. Mahlke, Taewook Oh, Heeseok Kim, and Hong-seok Kim. Edge-centric Modulo Scheduling for Coarse-grained Reconfigurable Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 166–176, New York, NY, USA, 2008. ACM.

[PPM09] Hyunchul Park, Yongjun Park, and Scott Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[QMM11] X. Qin, C. Muthry, and P. Mishra. Decoding-aware compression of FPGA bitstreams. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(3):411–419, 2011.

[Rau94] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO)*, pages 63–74, New York, NY, USA, 1994. ACM.

[RBM+11] Mateus B. Rutzig, Antonio C. S. Beck, Felipe Madruga, Marco A. Alves, Henrique C. Freitas, Nicolas Maillard, Philippe O. A. Navaux, and Luigi Carro. Boosting Parallel Applications Performance on Applying DIM Technique in a Multiprocessing Environment. *International Journal of Reconfigurable Computing*, 2011:4:1–4:13, January 2011.

[RSS08] F. Redaelli, M. D. Santambrogio, and D. Sciuto. Task Scheduling with Configuration Prefetching and Anti-Fragmentation techniques on Dynamically Reconfigurable Systems. In *Design, Automation and Test in Europe (DATE)*, pages 519–522, March 2008.

[SABW12] R.A.E. Seedorf, F. Anjam, A.A.C. Brandon, and S. Wong. Design of a Pipelined and Parameterized VLIW Processor: ρ-vex v2.0. In *Proceedings of the 6th HiPEAC Workshops on Reconfigurable Computing*, page 12, Paris, France, January 2012.

[SAFW11] Ali Asgar Sohanghpurwala, Peter Athanas, Tannous Frangieh, and Aaron Wood. OpenPR: An open-source partial-reconfiguration toolkit for Xilinx FPGAs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 228–235, May 2011.

[SBFC10] Antonio Carlos Schneider Beck Fl. and Luigi Carro. *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques: Automatic Acceleration of Software Execution*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[Seo] Seoul National University. SNU Real-Time Benchmarks. `http://www.cprover.org/goto-cc/examples/snu.html`. Accessed 23 Dec 2012.

[SF12] Miguel Lino Silva and João Canas Ferreira. Run-time generation of partial FPGA configurations. *Journal of Systems Architecture*, 58(1):24–37, January 2012.

[SGNV05] Greg Stitt, Zhi Guo, Walid A. Najjar, and Frank Vahid. Techniques for Synthesizing Binaries to an Advanced Register/Memory Structure. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays (FPGA)*, pages 118–124, 2005.

[SKK15] J. Sarkhawas, P. Khandekar, and A. Kulkarni. Variable Quality Factor JPEG Image Compression Using Dynamic Partial Reconfiguration and MicroBlaze. In *International Conference on Computing Communication Control and Automation (ICCUBEA)*, pages 620–624, February 2015.

[SLL+00] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[SML09] M. Sima, M. McGuire, and J. Lamoureux. Coarse-Grain Reconfigurable Architectures - Taxonomy -. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim)*, pages 975 –978, August 2009.

[SNS+13] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne. Selective Flexibility: Creating Domain-Specific Reconfigurable Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):681–694, May 2013.

[SPA08] J. Suris, C. Patterson, and P. Athanas. An efficient run-time router for connecting modules in FPGAS. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–130, September 2008.

[SRK11] A. Salman, M. Rogawski, and J. P. Kaps. Efficient Hardware Accelerator for IPSec Based on Partial Reconfiguration on Xilinx FPGAs. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 242–248, November 2011.

[SV05] G. Stiff and F. Vahid. New Decompilation Techniques for Binary-Level Co-Processor Generation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 547–554, November 2005.

[Syn] Synopsys. Synphony model compiler. `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/Pages/synphony-model-compiler.aspx`. accessed on 4th January 2016.

[Tei12] J. Teich. Hardware/Software Codesign: The Past, the Present, and Predicting the Future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, May 2012.

[Tex] Texas Instruments. TMS320C6000 Image Library (IMGLIB) - SPRC264. `http://www.ti.com/tool/sprc264`. Accessed 23 Dec 2012.

[Tho80] James E. Thornton. The CDC 6600 Project. *Annals of the History of Computing*, 2(4):338–348, October 1980.

[Tim92] Tim Peters. Livermore Loops coded in C. `http://www.netlib.org/benchmark/livermorec`, 1992. Accessed 3 April 2015.

[Tri15] S.M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015.

[VEWC+09] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus Scheduled Interconnect in Coarse-Grained Reconfigurable Arrays. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 268–275, September 2009.

[VF12] K. Vipin and S. A. Fahmy. A high speed open source controller for FPGA Partial Reconfiguration. In *International Conference on Field-Programmable Technology (FPT)*, pages 61–66, December 2012.

[War02] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[WH95] M.J. Wirthlin and B.L. Hutchings. A dynamic instruction set computer. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, FCCM, pages 99–107, Washington, DC, USA, April 1995. IEEE Computer Society.

[WKMV04] S.J.E. Wilton, N. Kafafi, Bingfeng Mei, and S. Vernalde. Interconnect Architectures for Modulo-Scheduled Coarse-Grained Reconfigurable Arrays. In *Proceedings of the 2004 IEEE International Conference on Field-Programmable Technology (FPT)*, pages 33 – 40, December 2004.

[Wol03] Wayne Wolf. A Decade of Hardware/Software Codesign. *Computer*, 36(4):38–43, 2003.

[Xila] Xilinx. SDSoC Development Environment. `http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`. Accessed on 29th December 2015.

[Xilb] Xilinx. Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics. `http://www.xilinx.com/support/documentation/data_sheets/ds893-virtex-ultrascale-data-sheet.pdf`. Accessed on 6th January 2016.

[Xilc] Xilinx. Vivado High-Level Synthesis. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. Accessed on 29th December 2015.

[Xild] Xilinx. Xilinx UltraScale Architecture for High-Performance, Smarter Systems. `http://www.xilinx.com/support/documentation/white_papers/wp434-ultrascale-smarter-systems.pdf`. Accessed on 6th January 2016.

[Xile] Xilinx. Zynq-7000 All Programmable SoC. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`. Accessed on 6th January 2016.

[Xil12a] Xilinx. Partial Reconfiguration of a Processor Peripheral Tutorial. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/PlanAhead_Tutorial_Reconfigurable_Processor.pdf`, October 2012. Accessed on 9th March 2016.

[Xil12b] Xilinx. UG702: Partial Reconfiguration User Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf`, October 2012. Accessed on 13th January 2016.

[Xil15a] Xilinx. UltraScale Architecture and Product Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf`, December 2015. Accessed on 13th January 2016.

[Xil15b] Xilinx. Xilinx Artix-7 FPGAs: A New Performance Standard for Power-Limited, Cost-Sensitive Markets. `http://www.xilinx.com/support/documentation/product-briefs/artix7-product-brief.pdf`, 2015. Accessed on 13th January 2016.

[YM04] Pan Yu and Tulika Mitra. Characterizing Embedded Applications for Instruction-Set Extensible Processors. In *Proceedings of the 41st annual Design Automation Conference (DAC)*, pages 723–728, New York, NY, USA, 2004. ACM.