Automatic synthesis of reconfigurable instruction set accelerators

Citation for published version (APA):
Kastrup, B. (2001). *Automatic synthesis of reconfigurable instruction set accelerators*. [Phd Thesis 2 (Research NOT TU/e / Graduation TU/e), Electrical Engineering]. Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2001

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

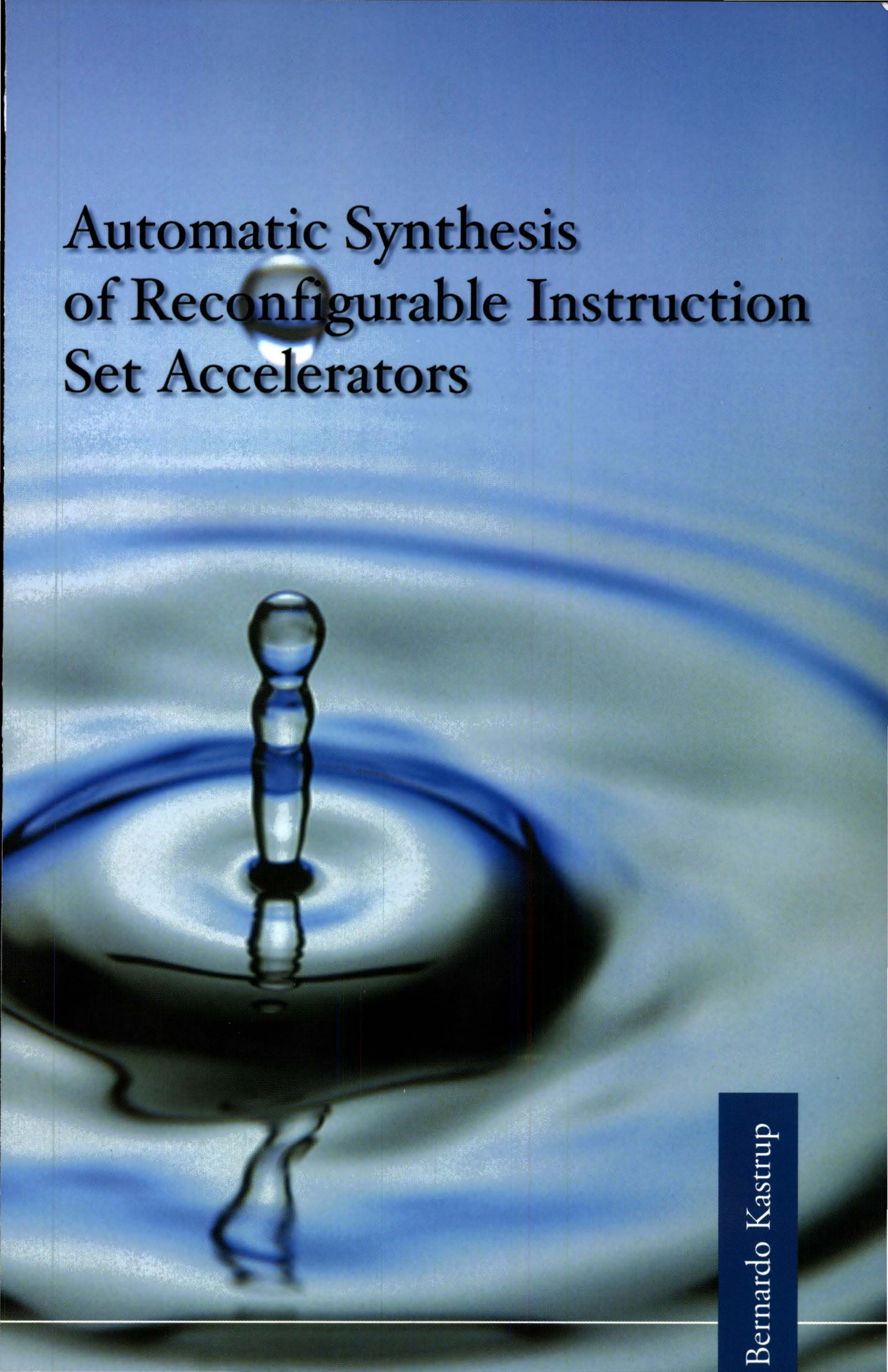
openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Automatic Synthesis of Reconfigurable Instruction Set Accelerators

Bernardo Kastrup



Automatic Synthesis of Reconfigurable Instruction Set Accelerators

Bernardo Kastrup

Automatic Synthesis of Reconfigurable Instruction Set Accelerators

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van
de Rector Magnificus, prof.dr. M. Rem, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen op
dinsdag 22 mei 2001 om 16.00 uur

door

Bernardo de Oliveira Kastrup Pereira

geboren te Niterói, Brazilië

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.-Ing. J.A.G. Jess
en
prof.dr.ir. J.L. van Meerbergen

Cover: Water drop photograph by J. Borowczyk. Licensed for non-commercial use by the author.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

de Oliveira Kastrup Pereira, B.

Automatic Synthesis of Reconfigurable Instruction Set
Accelerators / Bernardo de Oliveira Kastrup Pereira. -
Eindhoven: Eindhoven University of Technology.
Thesis Eindhoven. - With index, ref. - With summary in
Dutch and English.
ISBN 90-74445-50-0
Subject headings: reconfigurable computing, compilers,
high-level synthesis, computer architectures,
field-programmable logic

The work described in this thesis has been carried out at the Philips Research Laboratories Eindhoven, the Netherlands, as part of the Philips Research programme.

© Philips Electronics N.V. 2001
All rights are reserved. Reproduction in whole or in part is prohibited without the written consent of the copyright owner.

To the memory of my father, who inspired me, from a very early age, to relentlessly pursue a scientific understanding of the world we all live in.

Acknowledgements

The work described in this dissertation has been carried out during the first three years of my employment at Philips Research Laboratories in Eindhoven, as a full-time Research Scientist. I am grateful to the management of Philips Research for giving me the opportunity to use that work in this dissertation. In particular, I am in debt with Joachim Trescher, Menno Treffers, and Eelco Dijkstra, for the backing and moral support to complete this thesis.

Naturally, this dissertation would not have come true without the guidance of my supervisors, prof. Jochen Jess and prof. Jef van Meerbergen. The time I spent with them has been valuable to me.

I am grateful to my colleagues at the Compiler Technology cluster for all the support, *Hoogarden* and *bitterballen* we had together, all the exciting and mind-sharpening chess games, and for all the inspiring philosophical discussions about formal systems, the mind and the brain, Gödel's theorem, quantum physics, and so forth. In particular, I am in debt with Paul Hoogendijk, for his help in finding a suitable notation for the formalisms in Chapter 6, and for his later reviews of that chapter. Jeroen Trum has also been instrumental in the design and implementation of the software. I wish I could help him with his Ph.D. as much as he has helped me. Orlando Moreira has helped implement a large part of the simulation software while working for his graduation project under my supervision. Hans van Gageldonk has had the patience to translate my summary into Dutch. Finally, discussions I had with Willem Mallon have also helped sharpen parts of this thesis. Many thanks to all of you!

Other people have indirectly contributed to the achievement this thesis represents. The scientific background and personal character it takes to reach this academic degree are built over many years. I would not have made it without the influence and guidance of people like José Manoel de Seixas, Enrico Mattievich, and Rudy K. Böck, to mention only a few.

This thesis has been written from beginning to end during my off-work time. I am grateful to my wife Natalia for her love and care, and for her comprehension during the long weekends and evenings in which this thesis stole me from her.

Summary

Field Programmable Logic (FPL) devices are hardware circuits that can be customized after fabrication to perform a certain function. Just as in programmable processors, control data loaded in memory elements of the FPL controls the behavior of the computationally-active circuit elements. The configuration plane of FPL, however, has orders of magnitude more memory bits than the instruction words of typical programmable processors. This allows for a much finer granularity of control over data-path elements, in a true computing-in-space fashion. This way, the intrinsic parallelism of functions can be exploited in a FPL implementation.

There are three different levels of parallelism in a FPL-based, computing-in-space approach:

1. Boolean, bit-level parallelism, in which individual bits of operands are processed independently and in parallel, and boolean logic minimization opportunities can be exploited;
2. Lateral parallelism, or Instruction-Level Parallelism, in which data-independent operations are processed concurrently; and
3. Time parallelism, or loop-level pipelining, in which the computations of multiple iterations of a loop are overlapped.

Modern, general-purpose processors can exploit most of the available lateral parallelism, but are very limited with respect to bit-level and time parallelism.

In this thesis, we introduce ConCISe ("Compiler-driven, CPLD-based Instruction Set accelerator"), a programmable RISC processor that features a Complex Programmable Logic Device (CPLD) as a Reconfigurable Functional Unit (RFU). ConCISe targets embedded encryption applications. Just like the ALU, ConCISe's RFU must execute in a single clock cycle, during the execution stage of the RISC pipeline. However, unlike the ALU, the RFU can implement segments of the application's critical path, where bit manipulations are intense, in a true computing-in-space fashion. The mapping in space exploits the available boolean, bit-level parallelism and improves performance. For this reason, the approach is orthogonal to other processor design techniques like bigger caches or multiple instruction issue slots in VLIWs. We also introduce a general think-model that can provide some basic guidelines for a designer to evaluate the cost-effectiveness of reconfigurable computing approaches like ConCISe.

The ConCISe processor reconfigures its own RFU prior to executing an application, such that a set of application-specific instruction set extensions is available during execution. Unlike other reconfigurable processors, ConCISe does not use run-time RFU reconfiguration, due to its complexity and associated reconfiguration latency overheads. Instead, ConCISe encodes several custom operations in a single RFU configuration. The approach is simpler, more reliable, easier to verify, and eliminates reconfiguration overheads. Tests conducted show that the RFU proposed is large enough to implement most of the custom operations necessary to map the bit manipulations in the applications' critical paths. The price is that more FPL is necessary in ConCISe's RFU than in similar reconfigurable processors. It boils down to a trade-off between the amount of silicon required and the simplicity of design, verification, and programming.

The core of ConCISe is its programming tool-set, capable of automatically partitioning an application into software and hardware, and of automatically synthesizing the hardware partition in the RFU. Since the ease and speed of (re-)programming encryption processors are crucial factors in today's fast-changing Internet world, the tool-set meets an important demand. Unlike prior works, ConCISe uses innovative graph-based techniques during partitioning, trying to maximize performance and the utilization of RFU resources. These techniques are inspired on the field of compiler technology and blended with high-level hardware synthesis concepts.

Some new theoretical basis needed by the techniques employed is also laid down in this dissertation. In particular, an injection that allows one to transfer a partial graph covering problem from the domain of Directed Acyclic Graphs (DAGs) to an abstract domain of directed trees is introduced. The covering problem can be more efficiently and easily solved in the tree domain. The theoretical results are more general than the ConCISe framework itself, and could be a useful tool for DAG-based instruction selection problems normally encountered in compiler technology.

Benchmark results show that the ConCISe approach is a cost-effective investment. For some benchmarks, an RFU-extended MIPS processor exhibits a level of performance more than 1.5 time that of the same processor without the RFU. Given that all recurring costs of ConCISe reside in the RFU, which in turn represents only a small investment in silicon, this is a very promising result. In addition, silicon is an ever cheaper commodity in today's semiconductors world. The core of ConCISe is its programming tool-set, which represents a non-recurring cost. In this context, the benchmark results indicate that the approach can be an attractive solution for high-volume, encryption-enabled, embedded electronics.

The static encoding of several custom operations in a single RFU configuration, ConCISe's basic design principle, unfolds into interesting opportunities at the hardware synthesis and logic optimization levels of the tool-set. The logic corresponding to different operations can be statically cross-minimized by the tools, saving silicon or allowing for more custom operations to be mapped onto hardware. We use techniques of input encoding to demonstrate and quantify the impact of cross-minimization in the hardware partitions mapped onto ConCISe's RFU.

The thesis concludes that the configuration plane of FPL-based instruction set accelerators, like ConCISe's RFU, can be seen as a Hyper Long Instruction Word (HLIW), a natural next step in the exploitation of parallelism currently attempted with VLIW processors. Because HLIWs can map an application both in time and space, their compilers

will embody the convergence of traditional compilation and high-level synthesis techniques. ConCISe's compiler may be an early preview of a future in which the differences between compilers and synthesis tools will no longer be obvious.

Samenvatting

Field Programmable Logic (FPL) componenten zijn hardware circuits die na fabricage geconfigureerd kunnen worden om verschillende functies te kunnen uitvoeren. Net als programmeerbare processoren, besturen de controle-gegevens in de geheugen elementen van de FPL het gedrag van de circuitelementen die actief functies berekenen. Die controlegegevens bestaan echter uit orde-groottes meer bitten in het geheugen dan de instructiewoorden van typische programmeerbare processoren. Dit staat een veel fijnere korrel van besturing over de elementen in het data pad toe, op een ware bereken-in-ruimte manier. Op deze manier kan het intrinsieke parallellisme van functies in een FPL implementatie worden benut.

Er zijn drie niveaus van parallellisme in een FPL-gebaseerde, bereken-in-ruimte benadering:

1. Bit-niveau (of booleaans) parallellisme, waarin de individuele bitten van operanden onafhankelijk van elkaar parallel worden behandeld, en mogelijkheden tot booleaanse minimalisatie kunnen worden benut;

2. Lateraal parallellisme, of instructie-niveau parallellisme (ILP, "Instuction-Level Parallelism"), waarin data-onafhankelijke operaties tegelijkertijd worden uitgevoerd; en

3. Tijd-parallellisme, of lus-niveau parallellisme, waarin berekeningen in verschillende iteraties van een lus tegelijkertijd worden uitgevoerd.

Moderne, algemeen toepasbare processoren kunnen het beschikbare laterale parallellisme meestal uitbuiten, maar zijn gelimiteerd in mogelijkheden als het gaat om parallellisme op bit-niveau en op tijd-niveau.

In dit proefschrift introduceren wij ConCISe ("Compiler-driven, CPLD-based Instruction Set accelerator"), een programmeerbare RISC processor die een complexe programmeerbare logische device (CPLD) als Reconfigureerbare Functionele Unit (RFU) implementeert. Het doel is om met ConCISe embedded encryptie applicaties te implementeren. Net als de ALU moet de RFU van ConCISe in een enkele klokslag kunnen executeren, gedurende de executie-fase van de RISC pijplijn. In tegenstelling tot de ALU echter, kan de RFU delen van het kritische pad van de applicatie, waar veel bit-manipulatie operaties in voorkomen, op een ware bereken-in-ruimte manier uitvoeren. De afbeelding op de ruimte buit het beschikbare bit-niveau parallellisme uit en verbetert de prestaties. Om deze reden staat de gekozen benadering orthogonaal op andere technieken in het ontwerp

van processoren, zoals grotere caches en meerdere instructie-sloten in VLIW architecturen. Wij introduceren ook een algemeen denkmodel dat de ontwerper enige basisregels verschaft om de kosten-effectiviteit van herconfigureerbare berekeningen zoals toegepast in ConCISe te kunnen evalueren.

De ConCISe processor herconfigureert zijn eigen RFU voordat hij een applicatie gaat executeren, zodanig dat een verzameling applicatie-specifieke uitbreidingen van de instructieset beschikbaar zijn gedurende die executie. In tegenstelling tot andere herconfigureerbare processoren voert ConCISe geen run-time RFU herconfiguratie uit, vanwege de complexiteit en de bijbehorende vertraging door de herconfiguratie. ConCISe codeert verschillende applicatie-specifieke operaties in een enkele RFU configuratie. Deze benadering is eenvoudiger, betrouwbaarder, gemakkelijker te verifiëren, en kent geen vertraging door herconfiguratie. Experimenten tonen aan dat de voorgestelde RFU groot genoeg is om de meeste van de specifieke operaties uit te voeren die nodig zijn om de bit-manipulatie operaties uit de kritieke paden van de applicatie af te beelden. De prijs hiervoor is dat er meer FPL hardware nodig is in de RFU van ConCISe dan in vergelijkbare herconfigureerbare processoren. Het komt neer op een afweging tussen de hoeveelheid benodigd silicium enerzijds en de eenvoud van ontwerp, verificatie, en het programmeren anderzijds.

Het hart van ConCISe is de programmeer-omgeving, waarin het mogelijk is om een applicatie automatisch te partitioneren in hardware en software, en het hardware deel automatisch te laten synthetiseren tot de RFU. Omdat het gemak en de snelheid van (her)programmering van crypto-processoren van cruciaal belang zijn in de huidige snel veranderende wereld van het Internet, voldoet de programmeer-omgeving hiermee aan een belangrijke eis. In tegenstelling tot eerder werk maakt ConCISe gebruik van innovatieve graaf-gebaseerde technieken gedurende de partitionering, in een poging om de prestatie en het gebruik van middelen in de RFU te maximaliseren. Deze technieken zijn ontleend aan compiler technologie en zijn gemengd met concepten die bekend zijn van de hoogniveau synthese van hardware.

Dit proefschrift beschrijft ook nieuwe theoretische resultaten die het fundament vormen voor de gebruikte technieken. Een injectieve afbeelding is beschreven die ons in staat stelt om het probleem van de overdekking van deelgrafen uit het domein van gerichte acyclische grafen ("Directed Acyclic Graphs", DAGs) te beschouwen in het domein van gerichte bomen. Dit overdekkingsprobleem kan efficiënter en eenvoudiger worden opgelost in het domein van bomen. De theoretische resultaten zijn algemener dan de ConCISe omgeving zelf en zouden gebruikt kunnen worden als een gereedschap voor DAG-gebaseerde instructie-selectie problemen die men in het algemeen in compiler technologie tegenkomt.

In vergelijking met andere resultaten toont de ConCISe benadering aan een kosten-effectieve investering te zijn. Voor enkele voorbeelden blijkt dat een MIPS processor met RFU-extensie meer dan anderhalf keer zoveel presteert als dezelfde processor zonder RFU. Gegeven het feit dat alle terugkerende kosten van ConCISe zich in de RFU bevinden, die op zichzelf slechts een kleine investering in silicium voorstelt, is dit een zeer bemoedigend resultaat. Bovendien wordt silicium steeds goedkoper in de huidige wereld van halfgeleiders. De basis van ConCISe is de programmeer-omgeving, die op zichzelf een eenmalige investering met zich meebrengt. In dit verband laten de resultaten zien dat de gekozen benadering een potentieel aantrekkelijke oplossing is voor in grote hoeveelheden geproduceerde embedded elektronica voor encryptie.

De statische codering van verscheidene specifieke operaties in één enkele RFU configuratie, het basisprincipe van ConCISe, geeft aanleiding tot interessante mogelijkheden aan de hardware-synthese en optimalisatie kant van de tool-set. De hardware nodig voor de verschillende operaties kan statisch worden geminimaliseerd door de tools, hetgeen silicium bespaart en toestaat om meer specifieke operaties af te beelden op die hardware. We gebruiken technieken van invoer-codering om te laten zien en te kwantificeren wat het resultaat is van deze minimalisatie in de hardware die op de RFU van ConCISe wordt afgebeeld.

Dit proefschrift sluit af met de observatie dat de controle-gegevens van FPL-gebaseerde instructieset versnellers, zoals de RFU van ConCISe, gezien kunnen worden als een Hyper Long Instruction Word (HLIW), een natuurlijke volgende stap in het exploiteren van instructie-niveau parallellisme zoals dat op dit moment geprobeerd wordt door VLIW processoren. Omdat HLIW processoren een applicatie zowel op tijd als op ruimte kunnen afbeelden, zullen de bijbehorende compilers zowel technieken uit traditionele compilatie als technieken uit hoog-niveau synthese gebruiken. De compiler voor ConCISe kan in dit opzicht een blik in de toekomst geven waarin de verschillen tussen compilers en synthese-tools niet langer duidelijk zullen zijn.

Table of Acronyms

Acronym	Meaning
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
BFU	Basic Functional Unit
ConCISe	Compiler-driven, CPLD-based Instruction Set Accelerator
CPLD	Complex Programmable Logic Device
CTR	Compile-Time Reconfiguration
DAG	Directed Acyclic Graph
DES	Data Encryption Standard
DISC	Dynamic Instruction Set Computer
DPGA	Dynamically-Programmable Gate Array
FPGA	Field-Programmable Gate Array
FPL	Field-Programmable Logic
HDL	Hardware Description Language
IFU	Interconnected Functional Unit
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
IXC	Instruction eXecution Count
LFSR	Linear Feedback Shift Register
LUT	Look-Up Table
MaxMISO	Maximal Multiple Inputs Single Output graph
MISO	Multiple Inputs Single Output graph
PAL	Programmable Array Logic
PHDL	Philips Hardware Description Language
PLA	Programmable Logic Array
PRISC	Programmable Reduced Instruction Set Computer
PRISM	Processor Reconfiguration through Instruction Set Metamorphosis
PSI	Program-Specific Instruction
PSIP	Program-Specific Instruction set Processor
PT	Product Term
RaPiD	Reconfigurable Pipelined Data-path
RC	Reconfigurable Computing
RDS	Reconfigurable Data-path Segment
RFU	Reconfigurable Functional Unit
RTL	Register Transfer Level
RTR	Run-Time Reconfiguration
SRAM	Static Random Access Memory
ST	Sum Term
XPLA	eXtended Programmable Logic Array

Contents

1 Introduction

"Consider the surface of a pond, which can support many different types of waves and ripples. The hardware - namely the water itself - is the same in all cases, but it possesses different possible modes of excitation. Such software excitations of the same hardware can all be distinguished from each other."
Douglas Hofstadter, in "Gödel, Escher, Bach: An Eternal Golden Braid".

1.1 Setting the stage

In a programmable processor, the Instruction Set Architecture (ISA) is the interface between software and hardware. The ISA defines how well an application program can be mapped onto the processor architecture. Hennessy and Patterson [42] model the CPU performance as:

$$CPUtime = \frac{Seconds}{Program} = \frac{Instructions}{Program} \times \frac{ClockCycles}{Instruction} \times \frac{Seconds}{ClockCycle}$$

The ISA largely defines how many instructions a program requires and the number of clock cycles necessary to execute each instruction, to a large extent determining the first two terms on the right-hand side of the equation above. This way, the ISA is a crucial parameter for performance.

In traditional programmable processors, the ISA is defined by a set of fixed, hardware-based instructions, often called native instructions. The set of native instructions is chosen as to minimize cost and maximize performance for an entire range of potential applications. For that matter, programmable processors can be *domain-specific*, when the range of potential applications belongs to one particular domain (e.g. video processing, graphics, encryption, embedded control, etc.); or *general-purpose,* when the range of potential applications is extended to include applications from any domain (e.g. general purpose CPUs in PCs). In either case, the ISA must be general enough to allow for efficient mapping of its range of potential applications.

The price of generality, though, is often sub-optimal cost, performance, and power dissipation for any given, specific application program. There are at least two situations in the embedded computing domain in which this is an unnecessary cost:

1. When only one application is going to be mapped onto the programmable processor throughout its lifetime, in which case generality is not needed; or

2. When arbitrary, Program-Specific Instructions (PSIs) can be added to the ISA.

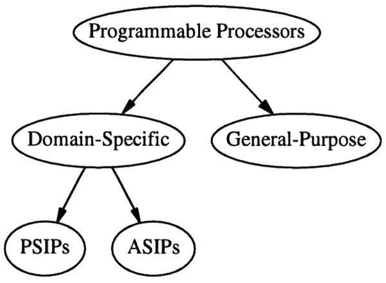


Figure 1.1: Simplified taxonomy of programmable processors, as defined in this dissertation.

In this work, we will define PSIs in the following way:

Definition 1.1.1. *Program-Specific Instructions* (PSIs) are instructions derived *from* a particular application program, and not designed beforehand.

This way, PSIs can exploit the particularities of the program code itself, not only those of the application domain.

The first situation often happens in the Semiconductors industry when time-to-market pressure is high, and/or when some degree of post-fabrication flexibility is required. Naturally, it is much faster, easier, and reliable to program an existing domain-specific processor than to design a new Application-Specific Integrated Circuit (ASIC). In addition, the system can always be reprogrammed. These domain-specific processors are, for times, designed with particular applications (representative of their domain) in mind, therefore being often called Application-Specific Instruction set Processors (ASIPs)[1]. ASIPs, however, represent silicon, performance, and power dissipation overhead when compared to ASICs.

If PSIs, as in definition 1.1.1, are used in an ASIP context, the resulting processor would considerably reduce that overhead. In fact, it would be more properly referred to as a Program-Specific Instruction set Processor, or PSIP. Therefore, PSIPs do not preserve the more general nature of ASIPs, and cannot be efficiently used with different application programs. See Figure 1.1.

To make the difference between ASIPs and PSIPs clear throughout, we will use both acronyms in this text.

The problem with PSIPs resides in the very characteristic that makes them superior to ASIPs: because they are highly optimized for a single program, they have very limited flexibility. The production volumes of a given PSIP must be high enough to justify

[1]ASIPs are considerably easier to instantiate from automatic design tools than ASICs, therefore permitting faster time-to-market [15]. Although most often used in the literature, the name "ASIP" may be misleading. Current ASIPs preserve a degree of generality that permits their efficient re-use for a number of different application programs in the domain, therefore not being truly *application-specific*. Having made this remark, and to be consistent with the literature, I will continue to use this same terminology throughout this work.

non-recurring engineering and mask costs. However, if the hardware-based optimizations employed in PSIPs could be electronically "reconfigured" for different application programs, just as the processor itself is reprogrammed, flexibility could be preserved.

Until the mid 1980's, however, reconfigurable hardware was technologically impractical. Only with the advent of Field-Programmable Logic (FPL), it became feasible to configure hardware circuitry after chip fabrication [1]. By using FPL within a programmable processor, hardware-based PSIs can be programmed in the same way native instructions are loaded in the processor's instruction memory. A processor featuring FPL resources for computing is called a *reconfigurable processor*. Although a reconfigurable processor could be entirely implemented with FPL, in this dissertation we will consider only reconfigurable processors in which hard-wired circuitry is augmented with FPL resources. Techniques that use FPL as a computing medium are referred to as part of the Reconfigurable Computing (RC) field.

This work presents and elaborates on new concepts and techniques to automatically generate PSIs to tune a reconfigurable processor to one particular application program. The objective is to enhance the performance and reduce the power dissipation of the processor for that program, while preserving the ease and reliability of design-by-programming. The target environment is high-volume, embedded computing. I will defend the point of view that PSIs must be generated automatically by a tool set. The reason is two-fold: the hand-crafting of PSIs would contradict the very reason why PSIPs, under certain circumstances, are superior to ASICs, i.e. ease and speed of programming; and for the more flexible reconfigurable processors, it would not be cost-effective (if at all possible in today's market reality) to hand-craft PSIs for every program the processor would run.

RC is a relatively recent field of research. For this reason, the next two sections will contain an introduction to RC and its enabling technology, FPL. A taxonomy of RC approaches is presented in Section 1.3.4.

1.2 Field-Programmable Logic (FPL)

Field-Programmable Logic, as the name suggests, is logic circuitry that can be programmed[2] in the field. The basic principle of FPL is that a completely pre-fabricated device contains customizable characteristics. Early FPL devices contained circuit connections that could be either open or closed, depending on the state of an anti-fuse attached to it. Initially open connections could be selectively closed by blowing their anti-fuses, therefore configuring the device's logic function. The problem with anti-fuse-based logic is that, once the anti-fuses are blown, the configuration is permanent. To overcome this limitation, other technologies were used to control the configurable con-

[2]The word *programming* is confusing in the world of RC, as it will be evident in the next chapters. For the sake of clarity, from this point on I will make the following differentiation: *programming* will be used in the context of loading program code into a processor's instruction memory; the word *configuring*, on the other hand, will be used in the context of loading circuit configuration data into FPL. The only exceptions to this convention will be acronyms like "FPL" itself, in order to preserve consistency with the literature.

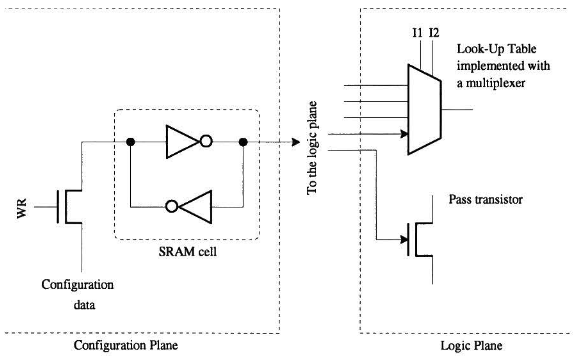


Figure 1.2: A typical FPL configurable point based on an SRAM cell. The value loaded in the cell at configuration time is used to control the circuit at run time. This can happen in different ways. For instance, the cell can control a pass transistor, typically in a configurable connection (therefore working similarly to an anti-fuse). Alternatively, it can be used as a memory element in the circuit itself, like in a look-up table used to implement combinatorial logic functions. Configurable points are scattered throughout the FPL, locally controlling their respective circuitry.

nections, like Erasable Programmable Read-Only Memories (EPROMs) and Electrical-Erasable PROMs (EEPROMs) [70]. More recently, FPL devices based on SRAM became available, allowing for multiple and fast (re)configuration of its logic. In this work, for reasons that will become evident later, we will be primarily interested in SRAM-based FPL. Therefore, from now on, every time we refer to configuration memory, it will be assumed to be SRAM-based, unless said otherwise.

Figure 1.2 illustrates an FPL configuration point controlled by an SRAM cell. Configuration data can be loaded in the cell through an n-transistor. Once loaded, the cell (holding a *configuration bit*) can be used to control various aspects of the logic circuitry, as pass transistors in a reconfigurable interconnect, or the contents of Look-Up Tables (LUTs) that implement logic functions. It is useful to make a distinction at this point: the memory cells holding the configuration bits, and the associated configuration circuitry, are in the so-called *configuration plane*. The circuitry controlled by the memory cells in the configuration plane is in the so-called *logic plane*. See Figure 1.2. Note that, because SRAM cells are typically made of as many as 6 transistors each, most of the silicon overhead in FPL is due to the configuration points in the configuration plane. The more configuration points a device has, the more flexible it will be, but the more silicon overhead it will carry with it.

Different FPL device architectures use different strategies to define which circuit elements in the logic plane are controlled by configuration bits in the configuration plane. The two main device architectures available today are the Field-Programmable Gate Arrays (FPGAs) and the Complex Programmable Logic Devices (CPLDs) [2].

1.2.1 FPGAs

In a typical FPGA architecture template, as illustrated in Figure 1.3, a number of configurable logic blocks, which implement logic functions, are distributed in a sea of configurable routing lines. Both routing lines and logic blocks are configured by their corresponding memory cells in the configuration plane.

To illustrate how a logic block can be configured to implement certain logic functions, Figure 1.4 shows a didactic example of a logic block structure. The structure in the figure is not extracted from any existing device, nor is it supposed to be optimal. It is, however, realistic and does illustrate some of the most important features of FPGA logic blocks:

1. The logic function is implemented by means of a LUT (a 3-to-1 LUT in Figure 1.4);
2. The logic block is hard-wired, by fixed connections, to a number of horizontal and/or vertical routing lines surrounding it, a subset of which is selected (by means of the configuration bits controlling input multiplexers) as input to the LUT;
3. A flip-flop can be optionally added to the signal path, to implement sequential logic and/or pipelined circuits;
4. The logic block can function as a routing block if the input and output multiplexers are properly configured to do so, by-passing the LUT.

In the case of the routing lines, the memory cells in the configuration plane typically control pass transistors (see Figure 1.2) that can connect two adjacent horizontal lines together, two adjacent vertical lines together, or a vertical line to an horizontal line. This way, the routing lines can be configured according to particular routing patterns, so to connect logic blocks and I/O together and implement the target global function.

A circuit is mapped onto FPL by properly configuring logic blocks and routing lines. The first step is the logic synthesis, which translates a technology-independent circuit specification onto logic equations. These equations are based on hardware primitives that can be implemented in the target device and, because of this, we will refer to this step as *technology mapping*. After it is complete, the two-step *low-level mapping* process is carried out. First, individual logic blocks are chosen to implement each of the hardware primitives in the synthesized circuit, therefore physically distributing the logic through the device. This step is called *placement*. The proper settings of the routing lines are then configured, in a step called *routing*. An efficient placement facilitates the routing process, while an efficient routing optimizes the circuit's critical path by minimizing the number of routing lines in it, as well as choosing the fastest lines available[3]. Therefore, the final

[3]The latency through a routing path depends on the number of configuration points in it, i.e. the number of routing lines connected together to form the path. Going through many configuration points increase the capacitive load in the path, slowing it down. Therefore, a typical FPGA will include hierarchical levels of routing resources, ranging from long distance lines to local neighbor-to-neighbor connections. The *router* negotiates these heterogeneous routing resources, trying to go through as few configuration points as possible.

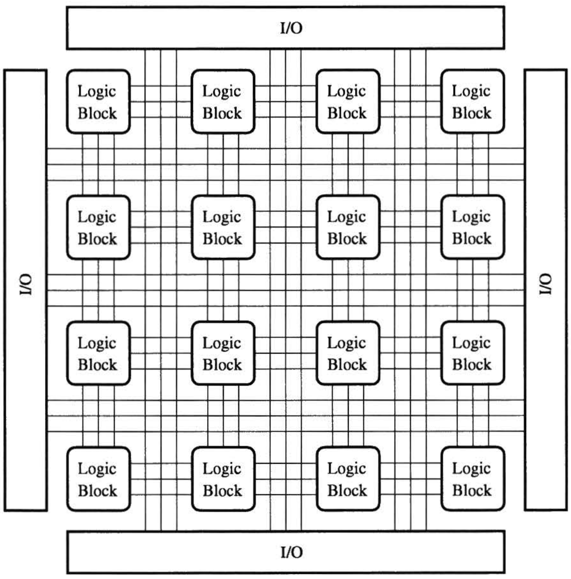


Figure 1.3: Typical FPGA architecture. Vertical and horizontal configurable routing lines surround programmable logic blocks.

circuit latency depends heavily on the results of placement and routing, and is highly unpredictable before the low-level mapping. On top of that, the results of placement and routing can change substantially given slight variations in the circuit or external constraints. This renders FPGAs difficult to use in situations when timing predictability and stability are important.

1.2.2 CPLDs

Unlike FPGAs, CPLDs have a centralized interconnect structure, as illustrated in Figure 1.5 for a typical CPLD. A given logic block can only communicate to another by feeding its outputs back to the interconnect. Input signals from I/O are also passed to the logic blocks via the interconnect.

A typical CPLD logic block can implement more complex logic functions than its FPGA cousins, as illustrated in Figure 1.6 for a PLA-based block. The logic is implemented as sums of products. The number of inputs to a logic block (coming from the interconnect) can be up to a few dozens. Each of these input signals, and its complement, can be used as input to logic gates in a configurable `AND` plane (implementing the Product Terms, or PTs). The outputs of the `AND` plane become inputs to a configurable `OR` plane (implementing the Sum Terms, or STs). The outputs of the `OR` plane then go through a macro-cell, which contain configurable state-holding elements for the implementation of sequential logic and pipelining. See Figure 1.7.

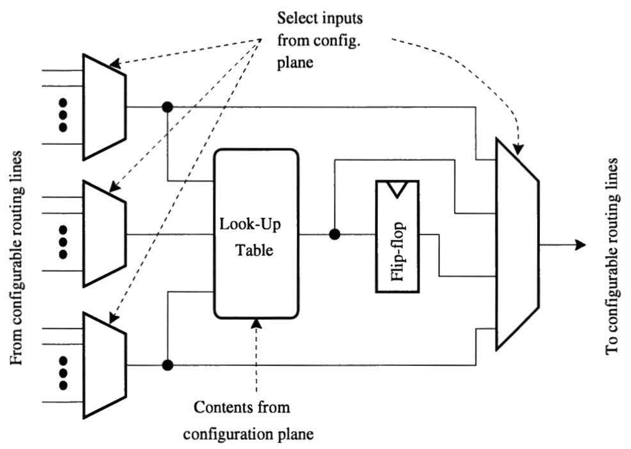

Select inputs from config. plane
Look-Up Table
Flip-flop
Contents from configuration plane
From configurable routing lines
To configurable routing lines

Figure 1.4: An example FPGA logic block. The look-up table contents are loaded during configuration and implement combinatorial functions. The flip-flop is optionally used for sequential logic and pipelining.

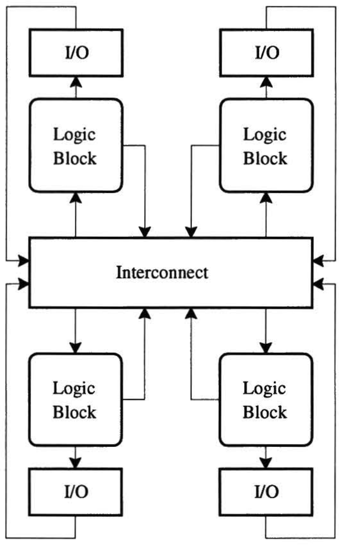


Figure 1.5: Typical CPLD architecture. An interconnect array permits communication among logic blocks and the implementation of multi-level logic circuits.

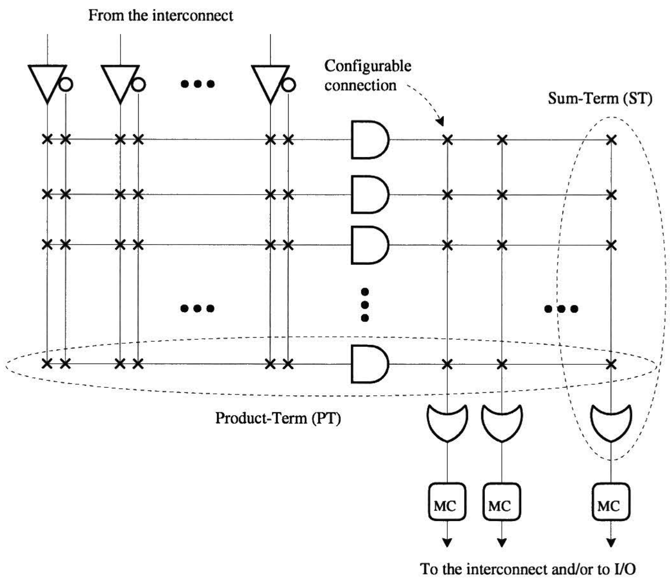


Figure 1.6: Basic CPLD Logic Block using the PLA architecture. The logic is implemented as sums of product-terms. "MC" stands for macro-cell.

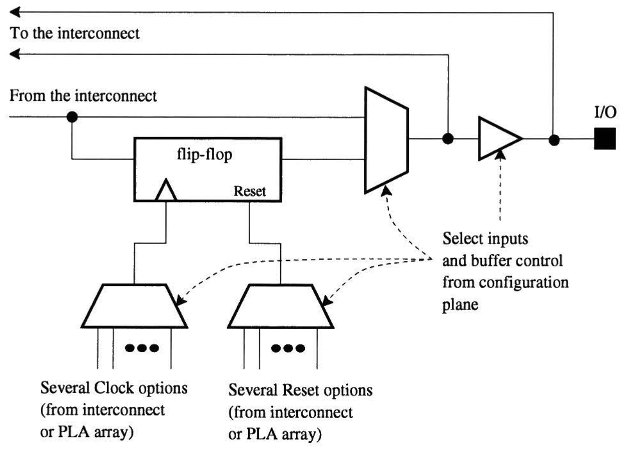

To the interconnect
From the interconnect
flip-flop
Reset
I/O
Select inputs and buffer control from configuration plane
Several Clock options (from interconnect or PLA array)
Several Reset options (from interconnect or PLA array)

Figure 1.7: Typical CPLD macro-cell. Pipelined, multi-level circuits are implemented by storing an intermediate result in the flip-flop, and feeding it back to the interconnect.

FPGAs	CPLDs
Distributed, heterogeneous routing resources.	Centralized, homogeneous interconnect structure.
Many, relatively simple logic blocks with relatively few inputs.	Few, complex logic blocks with many inputs.
Timing is unstable and difficult to predict before low-level mapping.	Stable and predictable timing model.
Complex low-level mapping tools (placer and router).	Simpler low-level mapping tools (*fitter*).
Optimized for multi-level logic.	Optimized for 2-level logic.
Higher logic capacity.	Lower logic capacity.

Table 1.1: FPGAs versus CPLDs

Though the logic blocks in a CPLD are typically more complex than their FPGA counter-parts, because the CPLD interconnect is centralized (and not distributed, as in FPGAs), there can be considerably fewer logic blocks in a CPLD than in an FPGA. As a consequence, one can typically map circuits of higher logic density onto FPGAs than onto CPLDs.

Another interesting comparison relates to multi-level logic circuits. The centralized interconnect in a CPLD limits the number of logic levels that can be mapped onto it (i.e. the number of feedback paths from output to input of logic blocks, through the interconnect). The distributed routing of FPGAs, on the other hand, is more efficient for multi-level logic.

Finally, regarding the predictability and stability of the timing model, CPLDs have considerable advantages when compared to FPGAs. First, because the CPLD structure is centralized and homogeneous, the placement and routing problem is much simplified, usually being referred to as *fitting*. Variations of a circuit can typically be mapped with no changes in the timing. In addition, the latency through the interconnect and through a logic block are predictable. The latency of a circuit can typically be evaluated by simply counting the number of times signals in the critical path loop through the set interconnect-plus-logic-block before becoming outputs.

1.2.3 Brief comparison between FPGAs and CPLDs

Table 1.1 contains a short overview of the main differences between FPGAs and CPLDs.

1.3 Reconfigurable Computing (RC)

This section is a brief introduction to the field of RC, within which this dissertation is positioned. One of the main aspects of RC is the early concept of computing-in-space, elaborated upon in the next section.

1.3.1 Computing-in-time versus computing-in-space

Probably, there are nearly as many definitions to the term Reconfigurable Computing as there are authors in the field. An useful definition is based on the realization that, typically, standard general-purpose and domain-specific processors compute by establishing instruction and data connections in time:

> A reconfigurable computer is a device which computes by using post-fabrication *spatial* connections of compute elements [14].

Although modern VLIW and Superscalar processors blur the borders of this definition, it is important to the extent that it brings up the fundamental topic of *computing-in-time* and *computing-in-space*, conceptually illustrated in Figure 1.8. There are two major reasons why computing-in-space is faster than computing-in-time. These are what we call *lateral parallelism* and *time parallelism.*

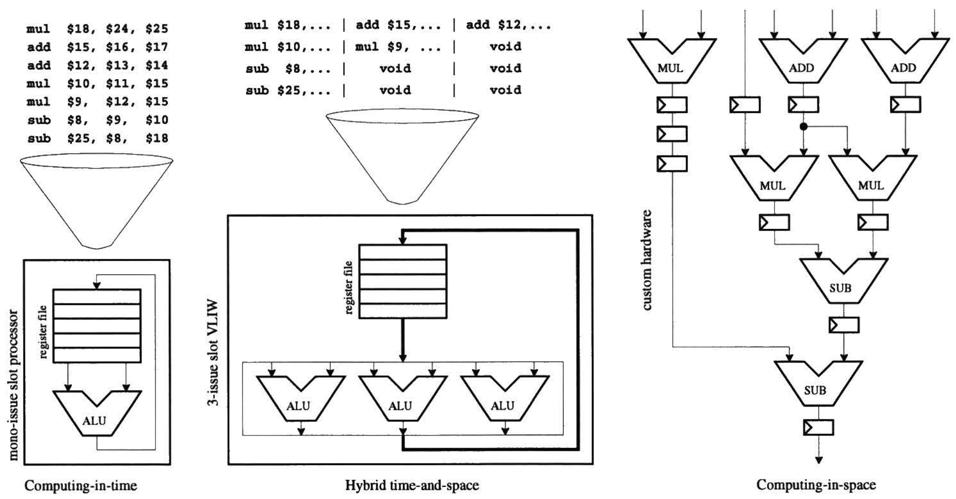
Lateral parallelism is the ability to execute data-independent operations concurrently, as illustrated by the 3 data-independent additions and 2 data-independent multiplications executed in parallel by the 3-issue slot VLIW of Figure 1.8. A mono-issue slot processor must re-use its computing-active element, the ALU, for each operation, therefore rendering lateral parallelism infeasible. Within the framework of modern VLIW and Superscalar processors, lateral parallelism is called *Instruction-Level Parallelism* (ILP).

Let a *data-flow stage* be a set of data-independent operations in a data-flow graph that, apart from resource constraints, can be scheduled together. This way, the program segment of Figure 1.8 is a 4-stage data-flow graph. Although Superscalar and VLIW processors are capable of lateral parallelism, they typically re-use their ALUs for consecutive data-flow stages, therefore computing in time to a big extent. Besides being capable of lateral parallelism, properly pipelined custom hardware can also execute every data-flow stage concurrently, as illustrated in Figure 1.8, optimizing data throughput. As the reader might have already concluded at this point, *loop-level pipelining* is the most popular term for what we mean by time parallelism. By means of loop-level pipelining, consecutive *time* samples of an input data stream can be processed in *parallel*, therefore the term "time parallelism". Note that this clearly differentiates both sorts of computing-in-space parallelism. ILP processors can typically compute the next time samples in an input data stream only after completing the computation of the previous ones.

Some VLIW processors do allow for the exploitation of limited amounts of time parallelism, as illustrated in Figure 1.9 for a simple data-flow graph of three operations. The processor's registers must allow for concurrent reads (of a value previously stored) and writes (of a new value). For a VLIW processor to be capable of completely pipelining a loop body, it must:

1. Have as many issue slots as there are operations in the data-flow graph that represents the loop body;

2. Have as many functional units (of the proper type) as there are operations in the graph;



```
mul  $18, $24, $25
add  $15, $16, $17
add  $12, $13, $14
mul  $10, $11, $15
mul  $9,  $12, $15
sub  $8,  $9,  $10
sub  $25, $8,  $18
```

```
mul $18,... | add $15,... | add $12,...
mul $10,... | mul $9, ... |   void
sub  $8,... |    void     |   void
sub $25,... |    void     |   void
```

Figure 1.8: Computing-in-time versus computing-in-space

3. Enough registers in the register file to store every intermediate result;
4. Every functional unit typically reads two registers and writes one. The register file must have as many read and write ports as required by all functional units operating concurrently.

Therefore, except perhaps for the simplest loops, to exploit loop-level pipelining with a VLIW processor would take a processor so rich in functional units, issue slots, registers, independent access ports to these registers, and interconnections, that the resulting datapath would in fact be a coarse-grained FPGA (i.e. an FPGA in which logic blocks are complex functional units, and whose interconnect is made up of word-wide buses)[4]. One can look upon the instruction word of such a processor (large enough for all issue slots), as the FPGA's configuration plane. Programming such a processor would also be less like compiling for traditional VLIWs, and more like RTL (Register Transfer Level) hardware design.

From Figure 1.8, the weakness of the RC definition given earlier becomes apparent. For the purposes of this dissertation, we will use the following simple and broad working definition:

Definition 1.3.1. Reconfigurable Computing, as a paradigm, is computing on FPL.

Based on this, the definition of a reconfigurable processor can be written in an alternative way:

Definition 1.3.2. Reconfigurable processors are programmable processors that utilize the Reconfigurable Computing paradigm.

[4]Coarse-grained FPGAs will be discussed in Sections 1.3.4 and 3.6.

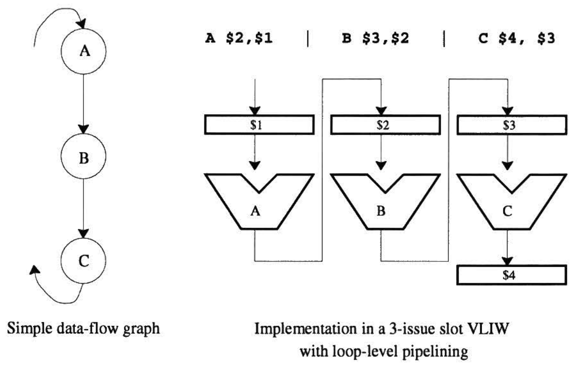

A $2,$1 | B $3,$2 | C $4, $3
$1
$2
$3
$4
A
B
C
Simple data-flow graph
Implementation in a 3-issue slot VLIW with loop-level pipelining

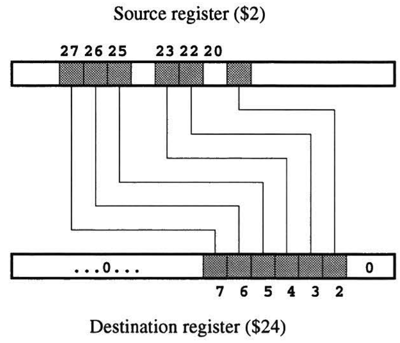
Figure 1.9: Trying to exploit time parallelism in a VLIW processor.

Reconfigurable processors, therefore, use FPL to perform computations. To the extent that FPL allows the mapping of computations in a computing-in-space manner, reconfigurable processors can explore lateral (ILP) and time parallelism (loop-level pipelining), with consequent performance advantages when compared to standard programmable processors.

1.3.2 Bit-level parallelism

In addition to allowing for lateral and time parallelism, reconfigurable processors have one more important advantage. The fixed bit-width data-paths and pre-defined logic operations of standard programmable processors render those devices inefficient when computing bit-level manipulations and sparse logic functions like the ones illustrated in Figures 1.10 and 1.11. In both figures, the original application was compiled by an optimizing compiler. The resulting assembly will take several cycles to execute, and will dissipate more power than the equivalent custom, 1-level logic implementations on the left-hand side of both figures.

Reconfigurable processors, on the other hand, can map computations onto FPL resources at the granularity of individual bits. This allows for the collapsing of a number of (potentially data-dependent) native instructions into a single custom instruction, in which all the original bit manipulations are incorporated in parallel. It also opens the possibility to optimize some of the original logic, as was the case in Figure 1.11, in which the `andi` at the end kills all but one bit in the result. The more sparse the logic of the original instructions is, the more of those instructions can be collapsed. However, depending on the case, even more complex, carry logic operations like adds or constant multiplies can be collapsed together with other instructions. We will from now on refer to the possibility of collapsing a set of native instructions into a boolean-optimized, bit-parallel implementation of their logic as *bit-level parallelism*. Note that bit-level parallelism can be looked upon as an instance of lateral (and possibly time) parallelism at the granularity of bits, instead of that of word operations, as in Figure 1.8.

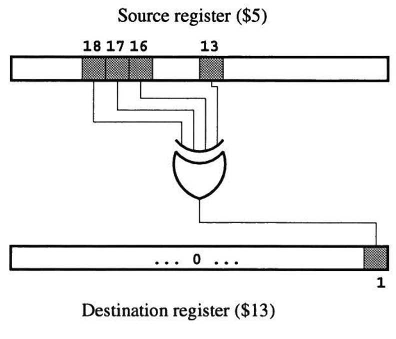


```
srl  $13, $2,  20
andi $25, $13, 1
srl  $14, $2,  21
andi $24, $14, 6
or   $15, $25, $24
srl  $13, $2,  22
andi $14, $13, 56
or   $25, $15, $14
sll  $24, $25, 2
```

Custom hardware implementation (bit wiring only)

Implementation on MIPS native instruction set

Figure 1.10: Bit-level parallelism for bit permutations in the DES encryption algorithm.



```
srl  $24, $5,  18
srl  $25, $5,  17
xor  $8,  $24, $25
srl  $9,  $5,  16
xor  $10, $8,  $9
srl  $11, $5,  13
xor  $12, $10, $11
andi $13, $12, 1
```

Custom hardware implementation

Implementation on MIPS native instruction set

Figure 1.11: Bit-level parallelism for an irregular boolean function in the A5 encryption algorithm.

Summarizing the ideas so far, reconfigurable processors based on the RC paradigm allow for performance improvements w.r.t. traditional programmable processors to the extent that they can exploit parallelism at three levels:

1. Lateral parallelism, or Instruction Level Parallelism (ILP);
2. Time parallelism, or loop-level pipelining; and
3. Bit-level parallelism.

Note that points 2 and 3 above are the ones that, today, can only be fully exploited in reconfigurable processors.

Proposition 1.3.3. *Because traditional VLIW and Superscalar processors can already exploit much of the available lateral parallelism, the true added-value of reconfigurable processors resides in their capability to exploit bit-level and time parallelism, while preserving the flexibility of a programmable device.*

Most of the concepts and results reported in this dissertation will be related to bit-level parallelism.

1.3.3 Is RC truly a new paradigm?

If by now you think that our working definition of RC on page 11 establishes a *clear border* from which one can *unambiguously* tell apart standard programmable processors from reconfigurable processors, you have been fooled.

The bottom line is that both programmable processors and FPL are technologies that allow for post-fabrication customization of their computing resources in order to perform different computations. They both allow for different "software excitations" of the same hardware. In either case, computing resources are "customized" by control (or "configuration") data stored in memory elements. In a standard programmable processor, control data is stored in the instruction register and, through the instruction decoder, controls the computing resources (ALU, registers, data-path multiplexers, status flags, etc.). The amount of program memory limits the size of the applications that can be implemented. In the FPL resources of a reconfigurable processor, on the other hand, control data is stored in the memory elements of the configuration plane and control the computing resources in the logic plane (see Figure 1.2 in page 4). The size of the applications that can be implemented is typically limited by the amount of FPL available.

Therefore, the differences between standard programmable processors and FPL are just relative, and can be illustrated as in Table 1.2. The third line top-down in the table refers to the frequency with which control data changes during the execution of a given application. Although FPL typically computes an application with a single configuration, there are alternatives to this, by means of run-time, dynamic reconfiguration. See Section 1.3.5 ahead.

Although the differentiation between standard programmable processors and reconfigurable processors (page 11) is important from a methodological point of view, the

Standard programmable processors	FPL
Control data interfaces to the computing resources typically through an instruction decoder and further dedicated control circuitry.	Control data typically controls the computing resources directly.
Control data is typically packed in words with relatively small bit width (e.g. 32 or 64 bits).	Control data is typically packed in relatively large configuration planes (typically thousands of configuration bits).
Control data changes frequently, with the load of every new instruction word (intensive use of computation-in-time).	Control data never changes for a given application, or changes less frequently than in a programmable processor (null or weak use of computation-in-time).
Application size limited by the amount of instruction memory.	Application size limited by the amount of FPL.

Table 1.2: Standard programmable processors versus FPL.

reader should always keep in mind that it is, ultimately, artificial. One can look upon both approaches as opposite extremes in a continuous spectrum in which the size of the control word and the degree to which computation-in-time is used varies. It may range from small control words that are reloaded for every primitive operation (e.g. in a mono-issue slot MIPS processor) to very large "configuration" words that are never changed during the execution of a given application (e.g. most commercial FPGAs). Complex VLIW processors with multiple functional units per issue slot (e.g. Philips' TriMedia processor [3]) fall somewhere in between.

1.3.4 A taxonomy of RC approaches

In most applications, there is a large amount of code that is executed relatively rarely. Attempting to map all functionality in space, using FPL, would consume more silicon than necessary. The non-critical parts of the computation can often be multiplexed in time, saving silicon without significant performance deterioration. A more cost-effective balance can be obtained by distributing the computational load between a standard programmable processor (which I consider the ultimate "general-purpose ALU time-multiplexer") and the FPL. Figure 1.12 illustrates this rationale.

Typically, the FPL executes segments of the application's critical path, exploiting boolean and time parallelism, which are not available in traditional VLIW and Superscalar processors. The standard processor is then responsible for program sequencing and control, and other non-critical parts. Because these non-critical segments typically consist of most of the application code, the approach off-loads the FPL and allows the critical path to be computed faster. The integration between FPL and processor must be tight enough to prevent the communication and synchronization overheads between

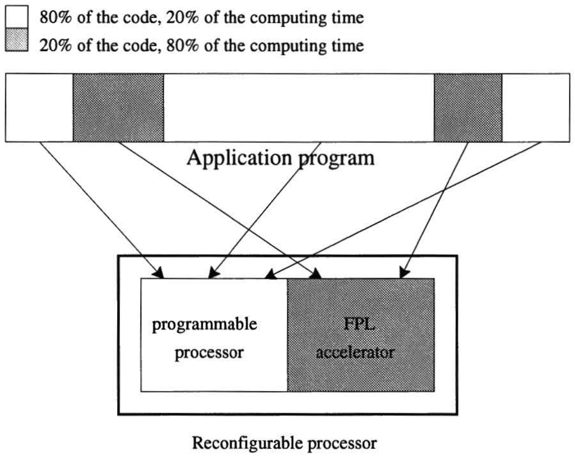

80% of the code, 20% of the computing time
20% of the code, 80% of the computing time
Application program
programmable processor
FPL accelerator
Reconfigurable processor

Figure 1.12: Caricature of the rationale behind reconfigurable processors. The application's critical path is mapped on FPL, exploiting bit-level, lateral, and time parallelism. The remaining code is mapped on a standard instruction set, to benefit from the time-multiplexing of computing resources.

the two from becoming a bottleneck.

Because the combination proposed in Figure 1.12 can strike an optimized balance between computing in time and space, from now we will only consider reconfigurable processors that are made up of two basic components: a standard programmable processor structure, and its FPL accelerator(s).

The way the FPL is integrated within a programmable processor defines a criterion with which to classify RC approaches. Two basic integration alternatives can be identified:

1. Unified instruction stream, when FPL is integrated within the data-path of the programmable processor, henceforth called *host processor*, and is directly controlled by instructions issued in the host processor. The FPL is typically scheduled at compile-time, as are the host processor's native instructions. In the literature, RC approaches using this integration method are often called *closely-coupled* [6];

2. Independent instruction streams, when FPL is integrated as a coprocessor of the host. Typically, run-time scheduling is used to activate the FPL-based coprocessor, which is usually asynchronous with the host and communicates with it via the system bus. In the literature, this integration approach is usually called *loosely-coupled* [6].

Host and coprocessor can be fabricated in a single silicon die or as different chips. In any case, and although an asynchronous coprocessor is not part of the core host proces-

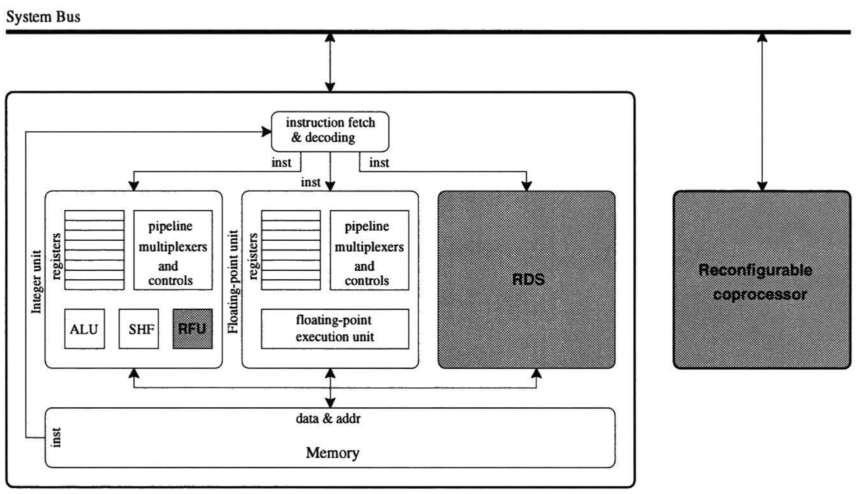

System Bus
instruction fetch & decoding
inst
inst
inst
Integer unit
registers
pipeline multiplexers and controls
ALU
SHF
RFU
Floating-point unit
registers
pipeline multiplexers and controls
floating-point execution unit
RDS
Reconfigurable coprocessor
data & addr
inst
Memory

Figure 1.13: Integrating FPL computing resources with a generic, mono-issue slot MIPS processor. FPL resources are shown shadowed. Examples of the 3 integration categories are illustrated.

sor, we will still refer to the system host-plus-FPL-based-coprocessor as a *reconfigurable processor*.

Closely-coupled approaches can be further sub-divided according to the degree of integration in the data-path. For the purposes of this thesis, only one more level of classification will suffice. Within the framework of a RISC host processor, closely-coupled reconfigurable processors can have Reconfigurable Functional Units (RFUs) or entire Reconfigurable Data-path Segments (RDSs) implemented with FPL. In the first case, an RFU is added in the execution stage of the pipeline. In the second, more than one stage of the pipeline can be by-passed, and the control flow diverted to added FPL resources. All categories are illustrated in Figure 1.13.

Note that the unified instruction stream approach is the one that allows for extensions to the host processor's native ISA. This can be achieved by means of implementing new, program-specific circuitry in the configurable FPL resources. A closely-coupled integration, therefore, paves to way to program-specific instruction set acceleration and, as such, is the focus of this thesis.

Another useful classification criterion for RC approaches is the granularity of the FPL itself. Depending on the degree of configurability it possesses, FPL can be classified in:

1. Fine-grained FPL, which is configurable at the level of individual bits. Therefore, it requires many configuration points (see Figure 1.2 on page 4). These will be the focus of this dissertation;

2. Coarse-grained FPL, which is configurable at the level of individual words (nibbles,

bytes, a pair of bytes, etc.). Therefore, it requires less configuration points than fine-grained FPL, and the reduction is directly proportional to the size of the word;

Examples of fine-grained FPL are illustrated throughout Section 1.2. Examples of coarse-grained FPL will be described in Chapter 3, but the reader may check [31], [33], [34], [35], [36], or [37] for some early pointers. Typically, the logic blocks in coarse-grained devices are entire, hard-wired ALUs, adders, or multipliers.

Note that because CPLDs have large and complex logic blocks when compared to FPGAs, they are often referred to as "coarse-grained" devices. According to the criterion established here, though, CPLDs are fine-grained devices, because every bit line in the circuit can be configured independently of any other line. See Figures 1.5 (page 7) and 1.6 (page 8).

The granularity of the FPL fine-tunes it to specific application domains. Fine-grained FPL is suited for irregular, random logic, due to its configuration flexibility. However, the many configuration points increase the silicon overhead w.r.t. hard-wired silicon. The use of less configuration points and largely hard-wired logic blocks in coarse-grained FPL leads to reduced silicon overhead and increased performance in domains where systolic, ALU-like computations prevail (e.g. image processing). However, its reduced flexibility limits its application. For instance, coarse-grained FPL typically cannot exploit bit-level parallelism. Chapter 2 contains a discussion about the choice of the right FPL architecture for the right application domain.

Although the literature contains examples of more elaborate classification systems, the taxonomy presented in this section will suffice for the purposes of this dissertation.

1.3.5 Static versus dynamic reconfiguration

The FPL resources within a reconfigurable processor can be configured statically, prior to application execution, and remain unchanged throughout the computing time. This is often called *static reconfiguration*, or Compile-Time Reconfiguration (CTR) in the literature [17]. Note that, in this case, there must be enough FPL resources to map the hardware-based part of the application completely, in space.

Because of the flexibility of FPL, however, other alternatives are also possible. For instance, the FPL can be reconfigured *during* application execution. This is often called *dynamic reconfiguration*, or Run-Time Reconfiguration (RTR) in the literature [17]. In this case, the hardware-based part of the application can be divided into temporal partitions, each implemented in a different configuration. Note that, here, computing-in-time is used in the FPL itself. However, because the loading of a new configuration plane typically cannot be done in parallel for all the bits, dynamic reconfiguration usually incurs in a reconfiguration latency overhead that must be negotiated at compile-time by the programming tools, or at run-time by special reconfiguration circuitry[5].

[5]Multiple configuration planes can be distributed in the FPL and have all their bits (re-)loaded in parallel (e.g. DPGA [14] or MorphoSys [33]). However, each new configuration plane adds to the already existing silicon overhead of FPL relative to hard-wired silicon (as mentioned in Section 1.2, the configuration plane is the main responsible for the overhead). Yet, if this is done, much of the rationale behind instruction cache efficiency can also be applied (e.g. [26]).

A third alternative exists for dynamic reconfiguration, in the form of *partial reconfiguration* [17]. In this case, only certain portions of the FPL resources are reconfigured, while the remainder can continue to operate. By efficiently negotiating the (partial) reconfigurations, some of its overhead can be hidden.

Dynamic reconfiguration (partial or not) can increase the efficiency of FPL utilization by using some degree of computing-in-time. However, to properly use this opportunity, major partitioning and reconfiguration management problems must be solved, either statically or dynamically, increasing the complexity of the approach.

1.3.6 Capabilities required in RC research

To the extent that RC blurs the borders between hardware and software, it is a multidisciplinary field that requires diverse capabilities. From high-level programming tools, like optimizing C compilers and hardware/software partitioning algorithms, down to FPL architecture design itself, all capability layers are part of an integral RC project, which must be approached in a true codesign fashion. Because of the unprecedented level to which these different layers are connected to each other in RC, approaching them in isolation would potentially lead to failure.

Figure 1.14 shows a summary of the basic three different layers of capabilities. They are as follows:

1. High-level tools design. RC projects will typically involve compiler design, but a compiler that is capable of HW/SW partitioning in order to decide which parts of the application program are to be mapped onto the FPL resources. We will refer to such a compiler as a *smart compiler*;

2. Low-level tools design, i.e. tools like hardware synthesis tools, placers, and routers, capable of performing the steps of technology and low-level mapping for the hardware partition;

3. Platform design, which includes the microarchitecture specification and the design of the FPL itself.

All three levels of capabilities will be addressed in this dissertation, in a true codesign approach.

1.4 Overview of this dissertation

Chapter 2 discusses the cost-effectiveness of the RC paradigm for high-volume embedded computing. The discussion aims at being objective and pragmatic, and will form the basis for justifying the applicability and relevance of the concepts and ideas presented later in the thesis. Chapter 3 briefly reviews the prior work on reconfigurable processors. The examples reviewed are classified according to the taxonomy presented in Section 1.3.4, and analyzed under the light of the think-model elaborated in Chapter 2. Chapter 4 introduces the architecture of ConCISe, a closely-coupled RFU approach that aims

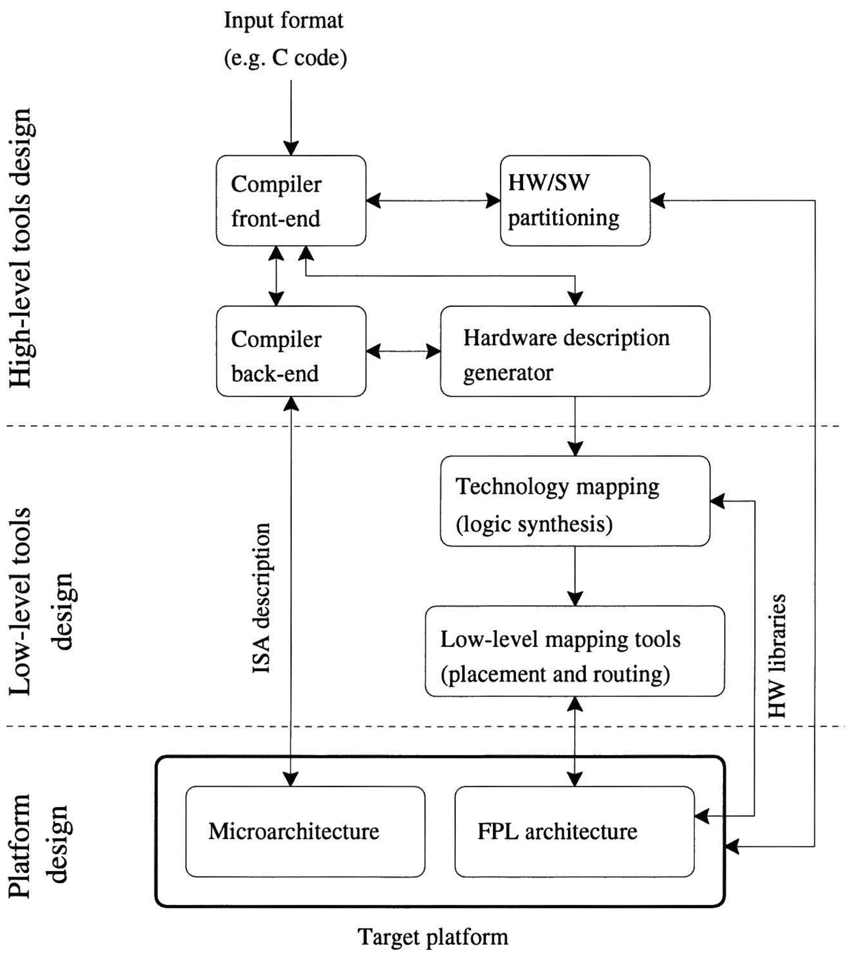


Figure 1.14: Different layers of capabilities in RC research, based on a generic RC design flow. Rectangles represent design modules, while arrows represent the main design dependencies. Double arrows are codesign dependencies.

at a reconfigurable processor optimized for bit-level parallelism (as introduced in Section 1.3.2, page 12). The discussion belongs to the platform design layer. The core of ConCISe, however, is its smart compiler, introduced in Chapter 5 and addressing the high-level tools capability layer. A formalism to describe ConCISe's hardware/software partitioning techniques and algorithms is introduced in Chapter 6. Chapter 7 presents and discusses benchmark results that evaluate the performance and cost-effectiveness of ConCISe. Chapter 8 deals with related logic synthesis aspects and, therefore, addresses the low-level tools capability layer. Finally, Chapter 9 concludes this dissertation.

2 The cost-effectiveness of RC for high-volume, embedded computing

"*Precisely* because *these firms (...) invested aggressively in new technologies that would provide their customers more and better products of the sort they wanted, (...) they lost their positions of leadership. (...) There are times at which it is right* not *to listen to customers, right to invest in lower-performance products that promise* lower *margins, and right to aggressively pursue small, rather than substantial, markets.*"
Clayton Christensen, in "The Innovator's Dilemma", Financial Times' Best 1997 Business Book Award.

As we will see in the next chapter, some of the prior work on RC overlaps with more traditional microarchitecture and compiler optimization techniques. This reflects the fact that, without a prior, careful consideration of the pros and cons of RC technology, and a clear vision as of what are the bottle-necks, advantages, and necessary trade-offs, one may end up doing redundant work. This chapter tries to provide one such a vision, based on the pragmatic idea of cost-effectiveness. The rationale in it has been initially published in [47]. With the basis laid down here, I hope to demonstrate later that the concepts presented in Chapters 4, 5, 6, and 8 have added-value and are relevant.

Note that this chapter is not intended to be a quantitative modeling work. It contains only qualitative reasoning, based on common-sense and on some insider's knowledge of the electronics industry reality. It aims at providing a check-list of relevant considerations to base design decisions.

2.1 Some ground rules

FPL technology is somewhere in between ASICs and standard programmable processors. It can be customized after fabrication, just like processors, and can implement logic in space with a high degree of bit-level, lateral, and time parallelism, just like ASICs. One can think of two ways to exploit those hybrid characteristics of FPL:

1. FPL can be cost-effective as a replacement for ASICs in low-volume quantities. The rationale behind it is that ASICs have considerable non-recurring engineering and fabrication costs. Masks are becoming more expensive with each new process technology, and VLSI design is getting ever more complex, difficult, and expensive to engineer and validate. The use of a prefabricated FPL device eliminates much

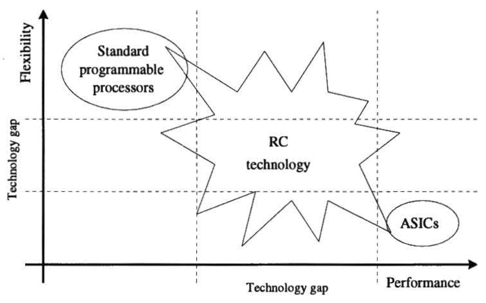


Figure 2.1: RC technology fills the gap between programmable cores and ASICs.

of the non-recurring costs. The flexibility of FPL permits the manufacturer to amortize these costs among a wide variety of customers, therefore lowering the device's price for each individual customer. Note that, in this case, the core feature of FPL as reconfigurable, "general-purpose" hardware is not exploited in the end product where it will be featured. There, it will not be reconfigured, but will behave just like an ASIC. Note also that this rationale does not apply to high-volume quantities, in which case the hardware overhead of FPL would render it less competitive than an ASIC;

2. The hardware reconfigurability of FPL can also be exploited as a core feature of the end product itself, in which case it can be attractive even for high-volume quantities. In this case, FPL would be used to replace or augment a programmable processor, performing more cost-effectively (part of) the processor's task. Note that a processor augmented with FPL (i.e., a reconfigurable processor) can be though of as a more cost-effective replacement for an expensive and power-hungry, high-end standard processor (without FPL).

In this dissertation, we will address the cost-effective use of FPL in high-volume quantities. We will be interested in exploiting the differentiating feature of FPL, its hardware reconfigurability, as an innovative solution for technical challenges, not as an economical convenience. RC will be used as an extra option in the current portfolio of embedded systems solutions, which fills the performance, flexibility, and power dissipation gap between ASICs and programmable processors. See Figure 2.1. Therefore, in the framework of this dissertation, we will be interested only in point 2 above.

2.2 The motivation for hardware reconfigurability

The area overhead of FPL with respect to ASICs is largely inherent to its flexibility. FPL, however, is re-usable, in the context of reconfiguration. The same piece of silicon, re-used repeatedly for different circuit implementations, can justify the area penalty it implies for a single implementation. Re-usability through reconfiguration is the main justification for the silicon overhead of FPL implementations of digital circuits.

For high-volume electronics, however, hardware reconfiguration is not necessary when:

1. The algorithms a computing device must run are fixed and known at design time. In this case, ASICs are more cost-effective (see the discussion in [?])[1];
2. Standard programmable processors can satisfy the performance and power dissipation requirements. If that is the case, programmable processors are a better choice because they are supported by a robust application programming environment. FPL is much behind programmable processors (and their well-developed compiler technology counter-part) in terms of programming friendliness, speed, dependability, and stability.[2]

> To the extent that the core of RC is hardware reconfigurability, it can only be cost-effective if this feature is demanded.

Devices targeted at very specific, individual applications do not demand so. They do not need flexibility, and their customizations can be hard-wired. On the other hand, general-purpose devices need to be able to run algorithms unknown at design-time during their life-time. They are built for generality and, therefore, their performance is sub-optimal for each individual application program. In addition, general-purpose devices need flexibility by definition. Because of these two factors, those devices are the ones that can best benefit from hardware reconfigurability.

> The more a device targets a general-purpose application space, the more it can benefit from hardware reconfigurability.

However, the story does not end here. FPL implies a performance and power consumption overhead when compared to ASICs, and this must be taken into account.

2.3 The right FPL architecture

Once one is convinced of the need for hardware reconfigurability, there are other issues to consider. As the enabling technology of the RC paradigm, FPL is a promising solution for a wide range of problems. A solution, however, that is embodied in many different architectures. The choice of the right architecture for the right problem is not always obvious.

If the target applications are known to be biased towards a certain kind of computation, a suitable FPL architecture can be chosen that performs faster (and with less silicon

[1]Here I am *not* considering time-to-market pressure issues.

[2]The complexity of designing custom hardware is uncomparable to that of programming a processor. A corollary to this is that, the closer a reconfigurable processor is to a standard programmable architecture, the greater are the possibilities of adapting standard compiler technology to make application programming easier, faster, and more dependable. Closely-coupled approaches (Section 1.3.4, page 15) have advantages with respect to it, due to their statically-scheduled, unified instruction stream. This reasoning is one of the main motivations for the architecture of ConCISe, presented in Chapter 4.

overhead) for that particular kind of computation. To elaborate more on this notion, it is necessary to extend the taxonomy of RC approaches presented in Section 1.3.4, page 15. Let a fine-grained FPGA be sub-divided in two categories (this does not apply to CPLDs):

1. Fine-grained, island-style FPGAs [1]. These have emphasis on global interconnect with several arbitrary, long-distance routing lines, and relatively complex logic blocks (the "islands" in the "sea" of routing lines). They are usually general-purpose architectures;

2. Fine-grained, cellular-style FPGAs [1]. These have emphasis on local routing lines interconnecting neighboring logic blocks. The logic blocks themselves are relatively simple and, therefore, can be present in higher numbers in the FPGA than their island-style counter-parts.

This way, island-style FPGAs, like the Xilinx XC4000 family [11], are better suited for complex, irregular logic. The abundance of (global) routing resources in these FPGAs render the devices more flexible and general-purpose. Precisely because of this generality, one could also implement other kinds of circuitry on them, like regular, systolic arithmetics. Still, this comes at the price of a level of flexibility - and overhead - that will not be utilized. In contrast, cellular-style FPGAs, like the Atmel AT6000 series [12], are better suited for highly local, pipelined circuits, such as systolic arrays, and imply less silicon overhead (the routing structure requires less configuration points).

Architectural optimizations that improve FPL performance for regular DSP arithmetic have been developed more extensively in the Academia, in the form of coarse-grained architectures, as described in Sections 1.3.4 and 3.6. These allow for a boost in performance and a reduction in the area overhead for the target applications. The loss in flexibility, in turn, renders coarse-grained devices inefficient for irregular bit-wise computations. Another limitation is that reduction of order is typically no longer possible for constant operands[3], as well as any kind of bit-level parallelism.

> Generally speaking, the FPL architecture can be fine-tuned towards a well-defined target application domain by varying the grain size of the logic blocks and the flexibility of the interconnect. Fine-tuning allows for a better trade-off between cost, performance, and flexibility.

Both the point made above, and the idea elaborated upon in Section 2.2, are illustrated in Figure 2.2. A fundamental dilemma becomes clear in that figure.

Proposition 2.3.1. *RC is most cost-effective when used in an application domain wide enough to justify hardware reconfigurability, while specific enough to allow for proper fine-tuning of the FPL architecture. Different domains may require different FPL flavors and different integration approaches.*

[3]For example, a multiply by a constant 2^n can no longer be order-reduced to a logical shift of n bits to the left.

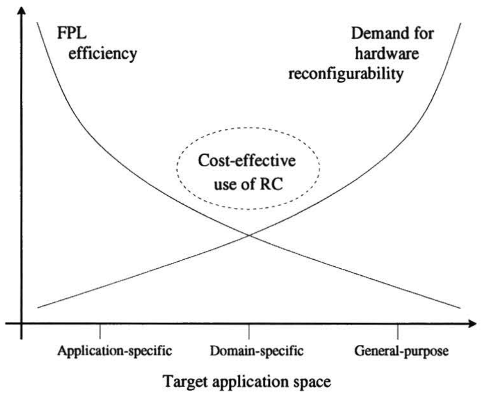


Figure 2.2: Finding the proper application space for RC.

Naturally, in reconfigurable processors, an efficient hardware/software partitioning can ensure that only the appropriate kind of computation is implemented in the FPL, even if the application has other sorts of computation that are not well suited for the particular FPL architecture in question. This is actually what many prior works have done in order to leverage RC in general-purpose computing (see the next chapter). Proposition 2.3.1 above then boils down to whether there are enough computations of the proper kind in the target application domain to justify the cost, complexity, and risk of deploying a new technology such as RC.

2.4 Trading off time and space

As discussed in Section 1.3.1, FPL technology is typically used for circuit implementations in the space domain. Operations present in the application can be mapped directly onto operators in space, many times with a one-to-one binding. This way, the intrinsic parallelism of an application can be fully exploited in hardware, in a data-flow computing fashion.

The parallelism exploited by computing in space does not come without a cost. Note in Figure 1.8 on page 11 that the custom hardware implementation of an algorithm requires several arithmetic operators, while the mono-issue slot MIPS processor performs the same computations by re-using a single ALU in time. The circuit replication necessary in computing-in-space results in more arithmetic circuitry being utilized. In a reconfigurable processor, that replicated circuitry carries with it the silicon overhead of FPL.

DeHon [13], however, points out that the control circuitry, instruction and data memory resources, utilized in standard programmable processors to allow for the re-use of the logic and arithmetic units in a computing-in-time fashion, reduce the computational density[4] of processors relative to that of FPL. Still, one should be careful at this point. Without the memory and control circuitry that render the processors computationally less dense, logic and arithmetic units would have to be replicated in space, *and* with FPL silicon overhead.

The higher computational density of FPL does *not* necessarily mean that it is a more cost-effective implementation alternative than programmable processors. Performance requirements permitting, applications could potentially be implemented in a (computationally less dense) processor that is smaller and less expensive than the (computationally denser) FPL device necessary to map the same application in space. The cost-effectiveness of one or another alternative depends on the application's latency and throughput requirements, how arithmetically intensive it is, what sort of computation it contains (systolic or irregular, control-intensive or data-intensive, integer or floating point, etc.), and other parameters.

> The bottom line is that the use of either computing-in-time or computing-in-space have their own costs and benefits, and to which degree one is more cost-effective than the other will depend on the application domain under consideration.

As discussed by Brebner [9], reconfigurable processors support both computing styles. This introduces a new degree of flexibility in trade-offs related to circuit area and processing time.

Proposition 2.4.1. *The challenge in reconfigurable processors is to balance the computational load between space and time computing in order to obtain the most cost-effective solution for the target application domain.*

2.5 A think-model

A cost-effective implementation of the RC paradigm will depend on four main questions, which define a think-model:

1. Is there a real need for hardware reconfigurability, or can either ASICs or standard programmable processors perform the computations more efficiently? (Section 2.2)

2. If there is a need, then what is the right FPL architecture? (Section 2.3)

3. If we have the right architecture, what is the right trade-off between computing in time and computing in space? (Section 2.4)

4. Are there particular requirements that must be taken into account for the target application domain, like need for user-friendly programming tools, low power consumption, device testability, etc.?

[4]Roughly defined as ALU operations per unit of silicon area. See [13] for details.

FPL granularity / FPL integration approach	Loosely-coupled	Closely-coupled RDS	Closely-coupled RFU
General-purpose, fine-grained, island-style FPL	Independent bit stream-oriented processors. E.g. networking, bit-serial DSP.	Complex, application-specific units integrated in the data-path of a processor. E.g. finite-field computations.	Small accelerator units for bit-level manipulations. E.g. cryptography.
Domain-specific, fine-grained, cellular-style FPL	Systolic processors connected to a bus. E.g. radar applications, front-end image processing.	Complex, application-specific units implemented in a systolic way. E.g. filter sections.	Simple units implemented in a systolic way. E.g. multiplication.
Domain-specific, coarse-grained FPL	Word-level computations in real-time, multiprocessor systems. E.g. video display processing.	Complex, application-specific units. E.g. transform coding.	Data-parallel processing, alternative to increased number of instruction issue slots. E.g. multimedia instruction set extensions.

Table 2.1: Mapping application requirements onto the taxonomy of RC platforms.

Table 2.1 is derived from this think-model. The rows represent the different levels of FPL granularity. The columns represent how the FPL is integrated with the hard-wired resources. Together, rows and columns map the taxonomy introduced in Section 1.3.4 with the extensions introduced in Section 2.3.

The table summarizes application requirements that map effectively onto each group and gives examples of applications. The RFU column in the table requires modest investment in FPL. It fits into a design environment where programmers have little hardware background, because automatic hardware partitioning and synthesis is easier to implement for these platforms (the RFU is typically small, and cannot implement complex circuitry anyway). The column in the middle is related to complex reconfigurable units for fixed-rate, high-throughput processing. The left-most column relates to applications demanding real-time, concurrent data processing with dynamic rates.

Armed with this think-model, one is now capable of briefly evaluating the wisdom of utilizing the RC paradigm when addressing different problems. In addition, the awareness and perspective provided by the model facilitate the trade-off decision-taking during the specification phase of any new RC approach. The model has been used in the definition of the ConCISe approach described in Chapter 4, and the insights it provided will be mentioned when appropriate. It has also been used in the review of prior work in the next chapter.

3 Review of prior work

"*- Where are you going?*
- Where they went.
- Suppose they went nowhere?"
Kirk and McCoy, in "Star Trek II".

This list of prior work on reconfigurable processors is not, and does not intend to, be exhaustive. Instead, its purpose is to give an overview of different approaches to RC, with their respective pros and cons, so that the reader gains a feeling for the field as a whole, and for what its issues are. For obvious reasons, in the next few sections special attention will be dedicated to fine-grained approaches (see Table 2.1) like ConCISe itself. I expect to justify design decisions made for ConCISe with the insights obtained from the review.

This review also aims at substantiating the propositions made in the first two chapters. When using the think-model of Section 2.5 to analyze each prior work, it will be assumed that there is always a motivation for hardware reconfigurability, because this is an implicit assumption in the original works themselves.

3.1 PRISM

PRISM stands for "Processor Reconfiguration through Instruction-Set Metamorphosis". It's a fine-grained, loosely-coupled approach (see Table 2.1 on page 29).

In the PRISM-I prototype [30], a standard programmable processor (Motorola 68010 running at 10MHz) is augmented with an FPGA board containing four Xilinx 3090 devices. Both processor and FPGA boards are connected to, and communicate through, a 16-bit system bus. A so-called "configuration compiler" partitions the application into software and hardware images, executed in the processor and in the FPGA board, respectively. The partitioning granularity, however, is at the C-function level, as illustrated in Figure 3.1. The idea is that the processor handles mainly the application sequencing and control, while the FPGAs crunch the data (extensive use of computing-in-space).

PRISM's configuration compiler is not fully automatic. Instead, it prompts the programmer with a list of functions that can be implemented in hardware, and it is the programmer who makes the partitioning decisions.

A limitation of PRISM (and of many other similar approaches) is the communication latency between processor and FPGA accelerator. Unless the computations performed in the FPGAs are complex and de-coupled enough from the processor, that overhead

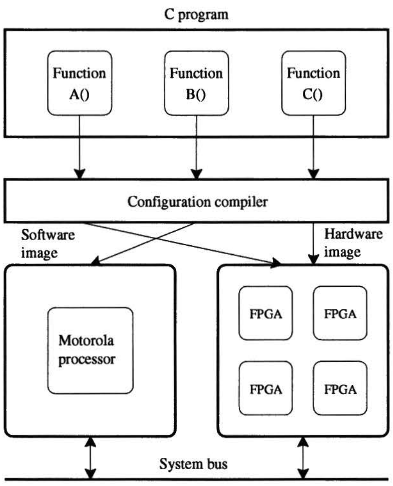


Figure 3.1: The PRISM architecture.

may dominate the processing time. Therefore, PRISM's effectiveness depends, to a large extent, on how well the programmer can split his/her application into de-coupled C functions that can be efficiently distributed over the FPL resources. A separate attempt to automatically minimize the communication overhead between host and FPGA accelerator [41] provides some insight into the complexity of the issues involved. Today's faster and cheaper processors can, for times, meet performance requirements with sequential implementations. They are easier and faster to program. These processors typically carry less silicon overhead than a distributed, reconfigurable computing-in-space approach like PRISM (question 3 of the think-model). In addition, because large chunks of the application must be mapped onto the FPGA board to minimize the communication overhead, it is likely that computations that do not fit well with the chosen FPL architecture will also be mapped onto it (question 2 of the think-model). For these reasons, today's standard processors may restrict PRISM-like approaches to limited application niches (Table 2.1).

PRISM's weakest point is the difficulty to program applications in a way that prevents the communication overhead between main processor and FPL co-processors from dominating the processing time. The FPL silicon overhead of extensive utilization of fine-grained computing-in-space also limits its application domain.

3.2 DISC

In spite of the name, under the classification criteria of Section 1.3.4 PRISM is not an instruction set accelerator (i.e. unified instruction stream), but a function-level one. DISC (Dynamic Instruction Set Computer) [18], however, is a fine-grained, closely-coupled,

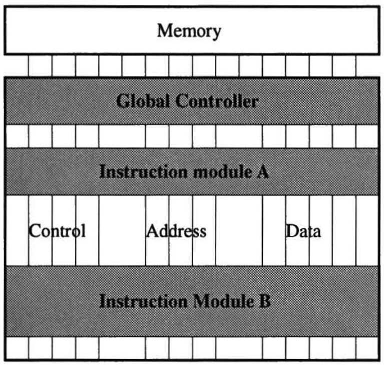


Figure 3.2: The DISC architecture, adapted from [18].

RDS approach (see Table 2.1) that does operate at the ISA level. It is the successor of the NanoProcessor project [19].

In the DISC processor, the instruction set is implemented by so-called "instruction modules", which are configured in FPL resources. The FPL is decomposed into fixed, vertical buses for control, addresses, and data (see Figure 3.2). Program sequencing and control, and management of global resources like external memory and I/O, are carried out by an also fixed global controller. The approach is such that the global controller and the buses can be implemented in hard-wired logic. Only the resources where the instruction modules are to be mapped need to be based on FPL. In practice, however, DISC's prototype was entirely implemented in an FPGA.

The DISC processor needs a host (a standard programmable processor) to manage the run-time reconfiguration of instruction modules. Every time a new module is needed, the host evaluates the current state of the FPL and chooses a physical location for the requested module. If possible, the new module is placed in a non-occupied position. Otherwise, a least-recently-used algorithm is applied to remove idling modules. The approach uses partial RTR (see 1.3.5), so while a new module is being configured, the others can still be used concurrently. The host also relocates modules at run-time, to minimize the distance in between them, and optimize the availability of contiguous FPL space where new modules can be placed. Therefore, DISC's reconfiguration management is particularly sophisticated. This, however, comes at a very high cost: a full, separate standard processor (the host), dedicated to run-time reconfiguration management.

Although a set of retargettable tools is available to program DISC [38], there is no automatic hardware/software partitioning and synthesis implemented. Indeed, the need for each module to be designed so to match with fixed buses physically, to be relocatable, and to implement potentially complex functions, renders automatic synthesis difficult. Therefore, instruction modules are hand-crafted and stored in a library, where they can be accessed by the host. Lastly, because the global controller implements none but the simplest (8-bit) operations, the FPL needs to implement most computations, even those that could be efficiently implemented sequentially, using a hard-wired ALU (question 3 of

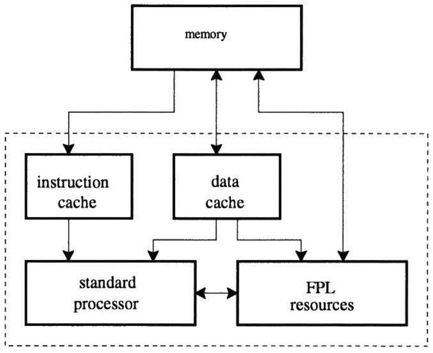


Figure 3.3: Garp architecture, adapted from [26].

the think-model). Consequently, the FPL architecture cannot be optimized for a specific kind of computation (question 2).

> DISC's weakest point is the complexity of its run-time reconfiguration management strategies, which require a dedicated standard processor. It also does not efficiently leverage the availability of instructions that can be implemented in hard-wired silicon (in the global controller).

3.3 Garp

Garp [26] resembles DISC in the way function modules mapped onto the FPL are oriented in horizontal rows, with global buses running orthogonally through the rows to bring values in and out. Garp's FPL had its routing architecture optimized for the vertical communication between function modules. Local interconnect is also favored. The logic blocks in the array are configurable at the granularity of a pair of bits, instead of an individual bit, as most commercial FPGAs. These characteristics relate to question 2 of the think-model, and represent a compromise. While trying to optimize the FPL's efficiency for performance-critical computations like multiplies and adds, Garp designers seek to preserve the FPL's flexibility for mapping more irregular logic functions.

Garp, as DISC, also uses RTR, including a configuration cache for recently used configurations. However, it is based on a full MIPS-like processor core, instead of a simpler global controller. The standard MIPS-II ISA was extended to interface to, and control the FPL. Therefore, Garp, as DISC, can be classified as a closely-coupled, RDS, fine-grained approach (see Table 2.1 and Figure 3.3). This classification concurs with that in [6].

There are efforts to build high-level, smart compilation tools for Garp [27], as well as low-level mapping tools [28] (see Figure 1.14 on page 20). In what I believe is an interesting and suggestive approach, the compiler re-utilizes traditional VLIW compilation techniques in the context of automatic hardware/software partitioning.

The partitioning strategy is such that the standard MIPS data-path is used for control, system interfacing, and other non-critical tasks (question 3). The FPL array maps relatively complex application segments (entire loop bodies) that can read and write data directly to memory (see Figure 3.3). This leverages the availability of a large FPL array. In addition, loop bodies mapped onto the FPL can be pipelined (time parallelism), allowing for some extent of stream computing.

Still, an FPL array like Garp's, with direct access to memory, suggests a separate co-processor controlled by a separate instruction stream, which would allow FPL and host to operate more de-coupled and concurrently. Yet, Garp's FPL array is controlled according to an unified instruction stream approach (see Section 1.3.4 on page 15), which simplifies the architecture since communication and synchronization between processor and FPL is now straight forward.

> Garp is a relatively complex RC approach. Since large segments of data and control flow can be mapped onto FPL, automatic partitioning and synthesis become a challenging task. Still, Garp introduces interesting innovations at the automatic hardware/software partitioning level, re-utilizing techniques originally developed for VLIW compilation.

In the next section, a tightly-coupled RFU approach similar to ConCISe will be reviewed.

3.4 PRISC

PRISC [21][22] stands for "Programmable Reduced Instruction Set Computer". Together with Spyder [16], it was the pioneer of RFU approaches. ConCISe is, to a large extent, inspired and based on PRISC's ideas.

In PRISC, a small, fine-grained FPGA-based RFU is inserted into the execution stage of a standard RISC pipeline, in parallel with the standard Functional Units (FUs). See Figure 3.4. The RFU is state-less, so that no FPGA state has to be saved in a context switch. The RFU must also execute in a single clock cycle to prevent synchronization difficulties in the pipeline. For this reason, the FPGA architecture was designed as to maximize the regularity of the timing-model, in an approach that relates to the question 2 of the think-model. Still, as in any FPGA, the final parameters of a circuit implementation (like area and delay) in PRISC's RFU depends on placement and routing. Because PRISC uses RTR extensively, reconfiguration control logic must be attached to the RFU via the ports `Paddr` and `Pdata`. This logic is typically a finite state machine responsible for reading configuration data from memory (possibly via the system bus) and controlling the sequential loading of this data into the RFU. The silicon required to implement it will increase the cost of the RFU, although this is not mentioned in PRISC's literature [21][22].

PRISC's compiler also aims at partitioning the application into software and hardware images. However, hardware partitions are now as small as short sequences of instructions. The partitioning tries to optimize both control and data flow. In the first case, PRISC's main optimization is to convert a set of `if-then-else`s into a switch statement that can

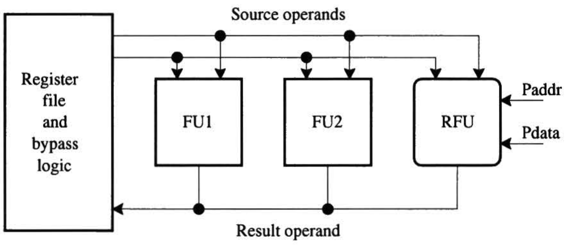


Figure 3.4: PRISC architecture, adapted from [21].

be translated into a logic function executed in the RFU, followed by a single jump. This way, conditional branches are eliminated from the control flow. In the second case, the compiler tries to convert sparse logic functions, which take several native instructions to execute, into an RFU instruction that executes in a single cycle (bit-level parallelism). Further acceleration can be obtained by specific programming techniques [23]. These, however, are manual code optimizations not implemented automatically by the compiler. In any case, the RFU is used only for the computations it is well suited for (question 2), and for which it can perform better than the standard functional units. This is possible solely because of the very fine partitioning granularity the architecture allows for. In addition, the high degree of integration between FPL and hard-wired resources allows for an efficient distribution of the computational load between computing in time and space (question 3).

PRISC's smart compiler prototype uses object code as input and classifies each native instruction as either being RFU-logic or not. RFU-logic instructions are those that implement sparse logic functions, which is checked according to an algorithm that evaluates the effective bit-width of each instruction's input operands. Then, the compiler utilizes a simple, bottom-up, greedy algorithm to detect the instruction sequences that make up the hardware image. Starting from an RFU-logic instruction, it walks backwards along the control flow as far as possible. The walk stops when the next instruction visited is not RFU-logic (e.g. floating-point operations, multiplies, wide adds, variable-length shifts, loads and stores, etc.), or when its addition would produce a function requiring more than two input operands, or more than one result. If a control flow segment detected this way (called a *maximal*) does not fit in the FPGA resources available in the RFU, the compiler prunes one instruction at a time from the top of the maximal, until it fits.

Only one operation at a time can be configured in the RFU. Every time a different RFU operation is needed, the pipeline is stalled while the RFU is reconfigured. The authors estimate a reconfiguration latency of 500 clock cycles, which limits PRISC's ability to accelerate the application's core loops with multiple RFU operations (in which case the reconfiguration latency would be part of the critical path). PRISC's compiler actually does not try to extract more than one RFU instruction per loop. Given that each RFU operation is necessarily simple, this can imply a loss of many acceleration opportunities.

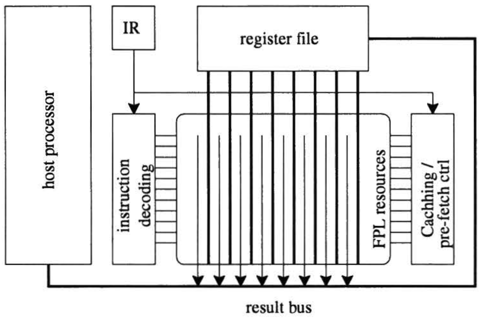


Figure 3.5: Chimaera architecture, adapted from [20].

As pointed out in [29], there are sub-graphs[1] (*kernel graphs*) derived from maximals that occur more frequently in the program code than their parent maximals[2]. By replacing a set of maximals with a unique core sub-graph, common to all maximals in the set, PRISC's reconfiguration overhead could be reduced. Sophisticated and innovative compiler algorithms would be necessary to compute and select optimal sets of kernel graphs, and to efficiently instantiate kernel graphs in the original code (partial graph covering). PRISC's compiler approach, however, is much simpler, and does not try to re-utilize RFU operations in different segments of code.

PRISC is a well-balanced architecture according to all questions of the think-model. However, it is limited by the fact that its RFU (unlike DISC's, for instance) can only implement a single operation at a time. PRISC's straightforward compiler approach also does not use techniques that could reduce the implied reconfiguration overhead by selecting and re-instantiating core RFU operations in different segments of code.

3.5 Chimaera

A tightly-coupled approach like PRISC, in which RFU operands are read from the register file just like the ALU operands, has obvious benefits for the integration between hardware and software partitions. However, it limits the number of input operands for the RFU to that of the ALU (typically two operands). If more input operands were available, the complexity (and consequent speed-up) of RFU configurations could be increased. To overcome this limitation, one could think of a register file with multiple read ports. This, however, would require major changes in the instruction encoding/decoding scheme to make room for the addresses of multiple source registers.

[1]Here I interpret segments of code (maximals or not) as graphs in which vertices represent instructions, and edges represent data dependencies between instructions.

[2]Although in [29] the maximals are pure data-flow segments, while in PRISC they can contain control-flow, the observation is still valid.

Chimaera's main innovation is in the way its RFU reads source operands. The FPL has direct access to individual bits of a sub-set of eight registers of the register file. See Figure 3.5. The addresses of these registers need not be encoded in the instruction word, because the FPL is already hard-wired to the desired input bits during configuration. This does eliminate all the register addressing flexibility of an individual RFU operation, which can prevent that operation from being used in other segments of the code. However, just as PRISC, Chimaera does not seek to re-utilize an individual RFU operation in segments of code other than the one the operation was originally extracted from. Instead, still as PRISC, it simply utilizes RTR, loading new RFU operations as they are needed.

The general partitioning strategy of Chimaera is like PRISC's. RFU operations must execute in one cycle after being scheduled. They target the acceleration of sparse logic functions (bit-level parallelism) or simple control flow segments that could be implemented in hardware using predicated (i.e. multiplexed) assignments. The FPL architecture in the RFU is state-less, and also somewhat tuned to the irregular bit manipulations it targets. It is not used to perform any other kind of computation (question 2 of the think-model). Other than that, because the RFU is hard-wired to its input operands even before an RFU instruction is scheduled, a form of "speculative" execution is possible. This, however, is very difficult to manage and exploit in a practical way.

A point that differs from PRISC is that Chimaera's RFU may contain several custom operations at a time. Similarly to DISC, different operations occupy different rows in the FPL array. For this reason, cross-optimization between operations during both logic synthesis and placement and routing is not possible.

> Chimaera's architecture merges features from PRISC and DISC. It has the advantage of being capable of providing multiple input operands to its RFU, while preserving a high degree of operands coupling between RFU and standard functional units. Still, custom operations are hard to re-instantiate multiple times in the code due to lack of register addressing flexibility. In addition, logic and floor-planning optimizations cannot be exploited across operation modules.

Other fine-grained, tightly-coupled approaches not reviewed in detail here include CoMPARE [24], an RFU approach similar to PRISC, but which allows RFU and ALU to execute concurrently; and OneChip [25], an approach in between PRISC and Garp with respect to the complexity of its FPL unit.

3.6 Coarse-grained reconfigurable processors

Most, if not all, commercial FPL devices in existence today are based on a fine-grained architecture that maximizes the device's flexibility and generality. However, when the target applications consist of regular, DSP-like computations, much of that flexibility (and silicon) goes unused. These computations typically have word-sized operands and local, systolic communication. This way, by reducing the amount of global interconnect resources, and by hard-wiring coarser-grained computing units in the logic blocks, the

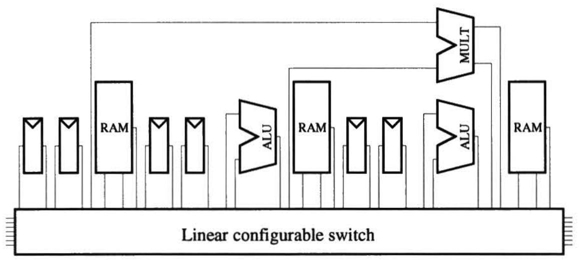


Figure 3.6: RaPiD architecture, adapted from [31].

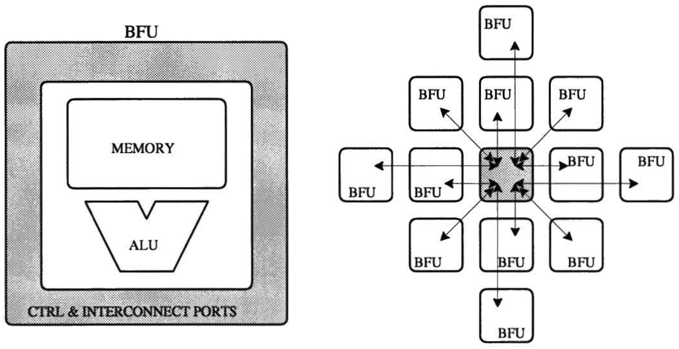


Figure 3.7: MATRIX architecture, adapted from [34]. There are two extra levels in the interconnect, a length-4 bypass and global lines, which are not shown here.

FPL can be fine-tuned to take advantage to the regularity and locality of the computations (see the discussion in Section 2.3, page 25). New FPL architectures have been developed in the academia, which try to achieve just that. The target is usually the ever growing market for multimedia applications.

RaPiD [31] ("Reconfigurable, Pipelined Data-path") is a coarse-grained FPL architecture optimized for DSP computations, which tries to exploit time parallelism, or loop-level pipelining (Section 1.3.1). Figure 3.6 illustrates a basic cell of a RaPiD array. The complete array is made up of 16 of these cells, connected in sequence (left to right in the figure) through the linear configurable switch. The configurable switch (made up of 16-bit bus segments) can connect RaPiD's 16-bit functional units (registers, memories, ALUs and multipliers, all hard-wired) in different ways, so to build up deeply-pipelined, linear, application-specific systolic arrays. Some interesting examples can be found in [32].

However, it is not trivial to specify efficient systolic arrays in an automatic fashion (i.e. with a smart compiler) [40]. RaPiD tools [39] require the programmer to explicitly specify the array's structure, timing, and parallelism. RaPiD's functional units are heterogeneous, which render automatic partitioning and synthesis yet more difficult. Other

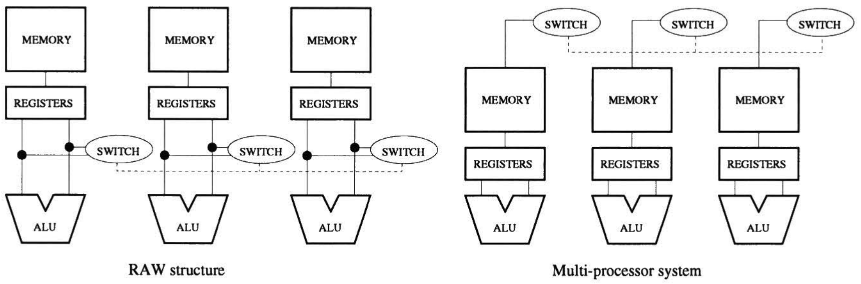


Figure 3.8: RAW architecture, as opposed to that of multi-processor systems. Adapted from [37].

coarse-grained FPL approaches seek to use homogeneous arrays with a single type of hard-wired unit that are easier to program. Examples are MATRIX [34] and Colt [35].

In MATRIX, each logic block (called a BFU, for "Basic Functional Unit") contains a hard-wired ALU, a memory block, and some interfacing/control logic. See Figure 3.7. Resources for instruction and data storage, and computation, are unified. The BFUs are placed in a network of hierarchical 8-bit buses, rich in nearest-neighbor connections. The structure is bi-dimensional (as opposed to the mono-dimensional RaPiD array), and resembles that of Figure 1.3, page 6, with BFUs as logic blocks and 8-bit buses as interconnect lines. MATRIX can be used in both a systolic and in a VLIW configuration, as shown in [34]. It strongly substantiates the point made in Section 1.3.1, page 10, that VLIW processors complex enough to exploit loop-level pipelining would actually need to have coarse-grained FPGAs in their data-paths.

Colt is a similar, bi-dimensional array approach, although its Interconnected Functional Units (IFUs) contain registers but lack the memory block of MATRIX BFUs. Colt also introduces the concept of "worm-hole run-time reconfiguration", in which packets composed of a configuration header and data steer themselves through the device. As the packet proceeds through a certain path, its header is consumed, configuring the computing resources in the path. The data that follows is then computed by the just-configured resources. Other examples of coarse-grained reconfigurable arrays include the KressArray [36] and MorphoSys [33].

Note that coarse-grained approaches like MATRIX, in which each logic block contains memory and computing resources, start to resemble multi-processor systems. In the RAW ("Reconfigurable Architecture Workstation") architecture [37], one step further is made in that direction. RAW is made up of identical tiles, each tile containing its own, separate instruction and data memories, registers, and ALU. Tiles resemble small processors, each running its own instruction stream. The architecture distributes the register file and memory ports, communication between ALUs taking place directly through a programmable switch. The difference between RAW and standard multi-processor systems is in the granularity of communication, as illustrated in Figure 3.8. RAW aims at exposing low-level details of the processor architecture to the compiler. It has more to do with traditional compilation techniques and resource scheduling, than with hardware synthesis itself.

As it can be seen from the examples above, the basic motivation for coarse-grained architectures is three-fold: Firstly, less silicon area is needed than in a bit-oriented approach. For the routing, a single configuration bit can control the (dis-)connection of an entire bus line. In addition, hard-wired, coarser-grained logic blocks occupy less silicon than an equivalent circuit implemented by configuring finer-grained logic blocks together; Secondly, because the basic computing elements are more complex, their performance is superior for the word-based computations they target; Thirdly, coarse-grained devices dissipate less power due to less configuration points. Naturally, these arguments only hold for the target application domain of these architectures, as discussed in Section 2.3.

Coarse-grained FPL can be inserted in a standard programmable processor just as fine-grained FPL. In this case, the classification of Section 1.3.4 (Figure 1.13, page 17 in particular) applies. RaPiD, for instance, can be an RDS in a reconfigurable processor. Approaches like MATRIX suggest a higher degree of de-coupling, and could be used as an accelerating co-processor. Architecture's like RAW can be interpreted either as coarse-grained FPL or as multi-processor systems with a finer granularity of communication between individual processors. For RAW-like approaches, however, because the reconfigurable array alone can implement the entire system (including the control part) by just deploying as many tiles as necessary, the need for integration with a standard processor is at least less evident.

The spectrum of FPL granularities seen above, starting from standard FPGAs, to a RaPiD array, to RAW processors, provides further insight into the point made in Section 1.3.3, page 14, that the differences between programmable processors and FPL are just relative.

3.7 Summary of prior work

Table 3.1 summarizes this chapter. It classifies the architectures discussed in the previous sections according to taxonomy illustrated in Table 2.1. It also mentions each architecture's strong and weak points, as discussed earlier.

Architecture	Integration approach	FPL type	Strong points	Weak points
PRISM	Loosely-coupled	Fine-grained, island-style	High levels of computing-in-space	Communication and FPL silicon overheads, programmability
DISC	Closely-coupled, RDS	Fine-grained, island-style	High integration between FPL and standard processor	Run-time reconfiguration management complexity and costs
Garp	Closely-coupled, RDS	Fine-grained, cellular-style	High integration between FPL and standard processor with high levels of computing-in-space	Complexity of automatic partitioning and synthesis
PRISC	Closely-coupled, RFU	Fine-grained, cellular-style	High integration between FPL and standard processor, friendly architecture to be targeted by a smart compiler	RFU can implement only a single operation at a time. Weak smart compiler, reconfiguration overhead
Chimaera	Closely-coupled, RFU	Fine-grained, cellular-style	High integration between FPL and processor while allowing for the FPL to read several operands at a time	Hard to re-instantiate custom operations, no cross-optimization across operation modules

Table 3.1: Summary of prior work.

4 ConCISe: a Compiler-driven, CPLD-based Instruction Set accelerator

"*(...) conceptual integrity is* the *most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.*"
Frederick Brooks, Jr., in "The Mythical Man-Month".

Conceived with the think-model of Chapter 2 as a guide, and based upon the insights derived from the prior work discussed in the previous chapter, ConCISe is designed around a single concept: all the multiple and disconnected segments that make up the hardware partition of an application are mapped onto, and encoded together into, a single configuration of the FPL. In spite of its simplicity, it is a powerful concept with far reaching consequences. In both the smart compiler and low-level design tools of ConCISe, novel problems and opportunities, which arise as a consequence of that concept, motivate many of ConCISe's innovations.

Throughout this chapter, I will argue along two lines. Firstly, I will contend that ConCISe is an innovative, sound, and efficient solution for the problems of its target application domain. This is the most important line of reasoning, for this is what justifies the whole work. Secondly, I will contend that ConCISe is a cost-effective design according to the think-model developed in Chapter 2. Yet, more often than not, these two lines of reasoning will be hard to distinguish from one another.

4.1 The rationale behind ConCISe

ConCISe targets the embedded cryptography domain. With the current convergence of media and information-sharing systems over the Internet, and the consequent necessity to protect data, embedded cryptography has grown in relevance. It has grown, however, in an environment that poses conflicting requirements. On the one hand, the demand for higher throughput in communication systems requires crypto systems with higher computing power. On the other hand, a growing population of diverse cryptographic algorithms and unstable standards-to-be that become moving targets, require a high degree of flexibility.

The rationale above motivates the use of RC, for the reasons illustrated in Figure 2.1, page 24. Consequently, it also provides the answer for the first question of the think-model in Section 2.5, page 28. In the target application domain of ConCISe, hardware reconfigurability is justified, to the extent that the device may be required to work with a number of different cryptographic algorithms throughout its life-time. As we will see in the next sections, the focus on a particular application domain also permits the choice of an FPL architecture that is properly fine-tuned to the typical computations of that domain, answering the second question.

Given the fast pace of change in today's communications world, the speed and reliability with which crypto systems can be reconfigured[1] to meet new standards and algorithms are also crucial factors to their commercial feasibility. In this context, because the programming of a processor is better understood, considerably faster, and more reliable than the design of new hardware circuits, a programmable processor has typically been the paradigm of choice [53][54]. For this reason, ConCISe is conceived as an extension (or an accelerator) to a programmable processor. Programming a ConCISe processor, therefore, should be as easy and direct as programming a standard processor.

The above-mentioned need for ease and speed of programming in the target domain is related to the fourth question of the think-model. It also means that it is important for an RC approach that aims at being cost-effective in the domain to feature an efficient, and fully automatic smart compiler as an integral part of it. One such a compiler is the heart of ConCISe. Other important requirements for high-volume embedded systems in general are: simplicity, reliability (paramount in cryptography), testability, and low cost. These requirements point out the way in which computing-in-time and computing-in-space are balanced in ConCISe, answering the third question of the think-model.

The next sections will tackle all these points in detail.

4.2 Architecture overview

As pointed out already in footnote 2, page 25, the closer the reconfigurable microarchitecture is to that of a standard processor, the easier it is to develop an efficient smart compiler using known compiler techniques. Looking at prior works, an approach like PRISC seems promising from this perspective. The close integration between FPL and host processor facilitates automatic hardware/software partitioning, because there are fewer concerns regarding communication overhead and data consistency between the partitions. ConCISe's integration approach is just like that of PRISC, illustrated in Figure 3.4, page 36.

Similarly, the idea of wiring the FPL directly into the register file in Chimaera (Section 3.5) could also be useful. Still, it is not adopted in ConCISe, for two main reasons: Firstly, it adds non-standard complications for the compiler (mainly during register allocation, because RFU instructions do not have input register addressing flexibility) and for the microarchitecture itself (at the operands-read and write-back stages of the pipeline). As mentioned above, simplicity and reliability are requirements that match better with

[1]Here, I use to word "reconfigured" in the broader sense, not necessarily in the context of FPL.

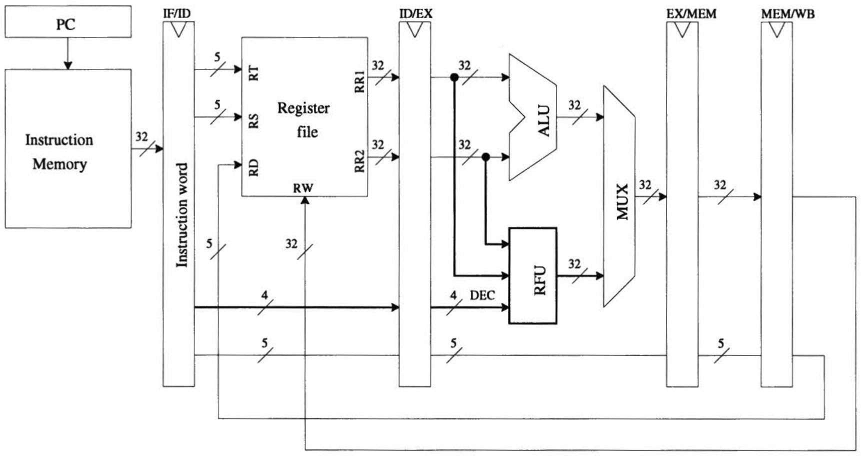


Figure 4.1: ConCISe architecture. Additions to the standard MIPS pipeline are marked bold. Only the lines relevant to a register-register operation are shown.

an approach like PRISC due to the RFU integration in the standard RISC pipeline as a normal functional unit. Secondly, and more subjectively, to preserve ConCISe's conceptual integrity around a single, basic design principle.

Having made these considerations, however, we are left with the weak points of a PRISC-like approach, as mentioned in Section 3.4. Namely: The RFU reconfiguration overhead, its inability to extract more than one custom operation per loop, and its simplistic compiler approach. ConCISe overcomes these limitations, as we will see in the next sections.

4.2.1 The data-path

ConCISe's data-path architecture is illustrated in Figure 4.1. As in PRISC, an RFU is added to the execution stage of a standard RISC pipeline, in this case a MIPS R3000, receiving the same two source operands from the register file as the ALU. All RFU-based instructions are register-register operations. ALU and RFU cannot execute concurrently (the RFU is not a VLIW or Superscalar extension). Like the ALU, and as in PRISC, the RFU must also execute in a single clock cycle. It can be configured differently for each particular application program, in order to implement different sets of custom operations that extend the native instruction set. Also as in PRISC, ConCISe's executables contain not only a list of instructions and program data, but also hardware configuration bits for the RFU. Prior to execution, the system loader loads instructions and program data into memory, and the configuration bits into the RFU.

Now, unlike PRISC, more than one custom operation can be configured in the RFU concurrently. Still, ConCISe does not use RTR to achieve it, as do DISC and Chimaera, but only CTR (see Section 1.3.5, page 18). All custom operations that make up the

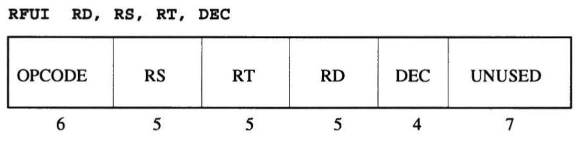


Figure 4.2: Program-Specific Instruction (PSI) encoding format.

hardware partition of any given application are known at compile-time, and they are all encoded together in a single RFU configuration (the way in which this is done will be described in details in Chapter 5). Therefore, there is one RFU configuration per application program.

We associate a single *instruction* mnemonic to all RFU operations configured. We call it a Program-Specific Instruction, or PSI. The particular one of the several *operations* configured that is to be executed at run-time is identified by an immediate value that follows the PSI's mnemonic in the assembly. This way, there is a one-to-one binding between a PSI and an RFU configuration, so there is only one PSI per application program. The PSI is encoded as a MIPS register-register operation, in which all three operands, plus a 4-bit immediate field called DEC, are present. Figure 4.2 illustrates the PSI encoding scheme. We chose an unused opcode from the MIPS R3000 instruction set to identify a PSI. The mnemonic "RFUI" represents it in the associated assembly language. The field RD is the address of the destination register, while RS and RT address the two source registers. DEC identifies the particular custom operation within the RFU configuration that is to be executed (such as the input signals in the ALU responsible for selecting an **add**, **sll**, or **xor** operation). Note in Figure 4.1 that DEC is also an input for the RFU. Because DEC is a 4-bit field, there can be up to $2^4 = 16$ custom operations per PSI. The benchmark results reported in Section 7 will demonstrate that 16 custom operations are typically enough to optimize the critical path of ConCISe's target applications.

From this point on, we will consistently use the word *instruction* to refer to a particular mnemonic in the assembly language, like a PSI instruction, and the word *operation* to refer to a particular custom operation encoded in the RFU configuration corresponding to the PSI instruction.

ConCISe's CTR approach is simpler, more reliable, and easier to test than RTR approaches. It is easy to see that verifying and validating a device whose very hardware architecture can change dynamically, in ways that potentially cannot be foreseen statically, is a non-trivial task. In addition, ConCISe is not limited to one custom operation per loop body, as PRISC is, because there is no reconfiguration overhead to be considered.

As a consequence of this approach, there is no need for run-time reconfiguration management units in ConCISe (as seen in Section 3.2, DISC required a full processor just to handle it), nor any need for the configuration caches used in Garp and Chimaera. Instead, the silicon that would have been taken by these two elements is used up in more FPL resources to implement all custom operations concurrently (see the next section). Naturally, there is no easy way to verify that the trade-off is even. But even if ConCISe's

RFU actually takes more silicon, the benefits should pay off in the target application domain.

As seen in the last chapter, FPL accelerators typically need a configuration controller (a state machine) to control the loading of configuration data. According to a specific protocol, and to a specific configuration data packet format (typically containing addresses and data itself), the configuration controller makes sure the proper configuration bits are loaded in the proper configurable points. In Figure 3.4, page 36, the configuration controller is attached to the RFU via the ports `Paddr` and `Pdata`. Note that a configuration controller differs from an RTR management unit, like DISC's so-called "host processor". The configuration controller is considerably simpler and operates at a more basic level. Still, it does add to the silicon overhead of FPL, with respect to hard-wired circuitry.

Because in ConCISe a configuration only happens at load-time, never at run-time, the processor itself can be used as a configuration controller for its own RFU, therefore reducing its silicon overhead. When a new application is to be run, the system loader runs a special routine that commands the processor to take the hardware configuration data in the executable, and to load it in its own RFU, prior to application execution. Again, one can look upon ConCISe as using up the saved silicon area of the configuration controller in more FPL resources to accommodate all custom operations.

Naturally, when a new application is to be loaded and run, the RFU reconfiguration process will take its time to complete. However, this is a static overhead associated to the initialization time of a new application. It never happens when the processor is in *line-mode*, i.e. crunching data streams circulating through communication lines in real-time. Initialization overheads occur more seldom and are considerably less critical than line-mode overheads.

To be fair, however, we must make a point at this stage. The cost-effectiveness of encoding all custom operations in a single RFU configuration in ConCISe would be less evident in a multi-tasking environment in which there are frequent context switches triggered by an operating system. The entire RFU would have to be reconfigured at every context switch, which would imply in huge overheads. Chimaera's partial RTR approach with configuration caches is designed precisely to tackle this point, fundamental in general-purpose computing. The overhead of a context switch is reduced to the extent that custom operations in the configuration cache are used by different applications. In the case of PRISC, there is no cache and reconfigurations occur throughout execution anyway, so context switches do not make things worse.

Because ConCISe is not targeted at general-purpose computing, but focuses on an embedded domain where there are no context switches, it can get around the complications of RTR and exploit the RC paradigm more cost-effectively.

Proposition 4.2.1. *By focusing an RC approach on an embedded domain where dynamic context switches do not occur, one can avoid the complications of RTR management at both software and hardware levels, and eliminate reconfiguration overheads. The idea is that all custom operations are encoded in a single FPL configuration, and only CTR is used. The resulting approach is simpler, more reliable, and easier to test.*

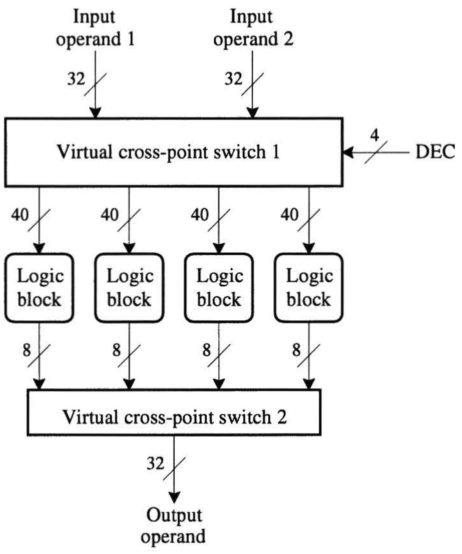


Figure 4.3: ConCISe's RFU architecture.

4.2.2 The RFU

In the embedded cryptography domain, bit-level manipulations and boolean operations are an important part of the computations. On top of that, as seen in Section 1.3.2, page 12, these operations are very inefficiently mapped onto a standard programmable processor. Efforts have been made to create crypto algorithms that use as few bit manipulations as possible, to prevent performance degradation [53].

ConCISe aims at easing this problem, by augmenting a programmable processor with an FPL-based accelerator. The idea is to exploit bit-level parallelism by mapping the bit-level manipulations in space. Because the FPL is placed within a RISC pipeline, and must always execute in a single clock cycle, it is crucial that the timing model of the chosen FPL architecture is stable and predictable. A CPLD was the chosen structure (Section 1.2.2, page 6), which can efficiently map bit-level manipulations under stable and predictable timings[2]. The reasons are:

1. In a CPLD logic block (typically a PLA, as in Figure 1.6), any individual bit line can be used in any PT (by setting the corresponding configurable connection), independently of how any other bit line is used. In an FPGA architecture, this would typically depend on resource constraints at the routing level;

2. Each logic block has many input bits coming from the interconnect, which can be

[2]An interesting comparison between the suitability of PT-based logic (as found in CPLDs) and FPGA-like logic for purely combinatorial circuits can be found in [44]. The conclusion is that, indeed, PT-based logic is more efficient for functions like the bit-level manipulations ConCISe targets.

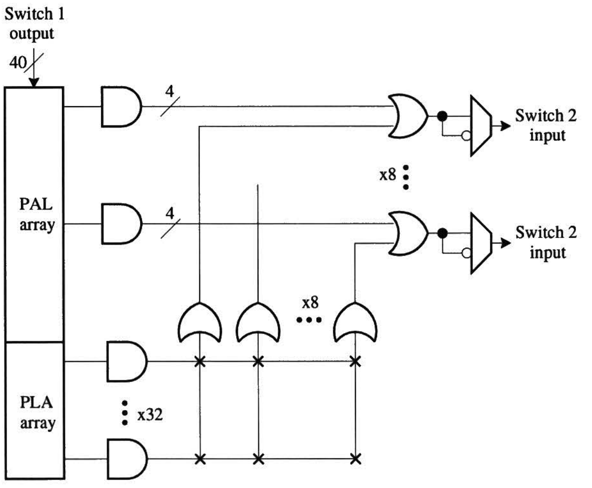


Figure 4.4: ConCISe's logic block architecture.

composed in relatively complex, irregular boolean functions, with fixed timing (the delay through the PLA). In an FPGA, complex, irregular boolean functions would need to be composed out of several logic blocks, which timing would be determined by placement and routing constraints.

We will see later that there are actually other good reasons why to use a CPLD structure in ConCISe.

The basic RFU architecture[3] is shown in Figure 4.3. It is a stripped, slightly modified version of commercial XPLA2 devices [55]. These devices are known for their low-power dissipation characteristics, achieved by a design technique that replaces the sense amplifiers, traditionally used in CPLDs to implement PTs, by a cascade of CMOS gates. Four logic blocks each receive 40 input signals from an XPLA2 virtual cross-point switch [55], reconfigurable at the level of individual bit lines (from this point on, every time we refer to a "switch", it will be assumed to be an XPLA2 virtual cross-point switch, unless explicitly stated otherwise). The inputs for the switch come from two sets of 32 input lines (the two source operands of the RFU) and the 4-bit DEC that comes from the instruction word (compare to Figure 4.1). Each logic block has 8 output lines, making up the 32 output bits of the result operand. A second switch is added in between the output of the logic blocks and the result word. This allows the fitter to balance the distribution of the logic among the four logic blocks. An XPLA2 virtual cross-point switch is only partially connected, saving area and reducing delay through the routing lines when compared to a fully connected switch. Still, it is designed so to guarantee a more than 99.9% chance of any given signal in a circuit being successfully routed to the desired destination. For

[3]The first ConCISe RFU architecture was disclosed in [48]. A later version was published in [49]. The version introduced in this dissertation is the latest and final one.

this reason, in the ConCISe tool-set, we assume that both switches are 100% routable (i.e. fully connected). The assumption greatly facilitates the construction of the tools, while having a negligible effect on the accuracy of the results.

The internal architecture of the logic block is shown in Figure 4.4. The combination of PAL (programmable AND plane followed by a fixed OR plane) and a PLA (programmable AND plane followed by an also programmable OR plane) in the logic block allows for building complex logic functions with a single pass through the arrays. For the sake of simplicity, the boxes "PAL array" and "PLA array" in the figure represent the matrix of configurable connections between inputs and AND gates in the PTs (compare to Figure 1.6 on page 8). The precise balance between the amount of PAL and PLA terms in a logic block has been achieved by the XPLA2 design team based on extensive simulations. Each output line has 4 dedicated PTs from the PAL array connected to it and, optionally, up to 32 PTs from the PLA array. Therefore, a total of 36 PTs can be used to produce a single result bit. Note that there are no macrocells with flip-flops or latches, because only combinatorial circuits are to be mapped onto the RFU.

The PAL/PLA combination allows for PLA PT sharing among different result bits (through different Sum Terms, or STs, which are just the OR gates). This increases the effective density of the device and allows for yet larger and more complex functions to be implemented. Again, the architecture is ideal for bit manipulations and, therefore, well tuned to the target application domain of ConCISe. The cross-point and the PAL/PLA arrays render arbitrary combinations of input bits available to any PT, facilitating random logic functions. The configured circuits are always purely combinatorial. The delay through the CPLD depends solely on whether PTs from the PLA are used, and is a constant in either case. With 1998 technology and a $0.35\mu m$ process, the maximum delay is approximately $5.0ns$. Therefore, the RFU can comfortably execute in a single cycle for clock frequencies of 100MHz or slightly more (enough for its target domain). We estimate its size to be between 4 and $5mm^2$ in the same $0.35\mu m$ process, based on the actual layout of commercial XPLA2 devices [61].

Although DISC and Chimaera can also implement several custom operations at a time in their FPL accelerators, they do it by using the concepts of modularization and fixed internal buses. Hardware modules corresponding to custom operations must be symmetrically mapped across these buses. This restricts the ability of the low-level mapping tools to minimize the logic. Furthermore, it poses limitations for the optimal utilization of the available FPL, since idling FPGA logic blocks may be the price for matching the modules with the appropriate buses. ConCISe does not have these limitations. The hardware partition is not organized in separate modules, and the tools can minimize it at will, both at a logic level and at a placement level. No matching with internal buses is necessary. In addition, the logic of different custom operations in one configuration (i.e. one PSI) can be cross-minimized, unlike in DISC or Chimaera. Specifically, when the logic of two or more custom operations have a common PT, under certain circumstances this PT can be shared (i.e. multiplexed) among them, reducing the total circuit size.

Corollary 4.2.2. *It follows from proposition 4.2.1 that ConCISe's CTR approach offers the added benefit of allowing for cross-minimization of the hardware corresponding to* *different custom operations. This is not possible with partial RTR approaches based on hardware modularization, like DISC and Chimaera.*

In chapter 8, cross-minimization opportunities will be discussed. That chapter contains benchmark results that substantiate and quantify my claim that cross-minimization is one of ConCISe's advantages with respect to similar prior works.

First, however, we need to address the smart compilation issues that make up the core of ConCISe.

5 Compiling applications for ConCISe

As mentioned in the previous chapter, the heart of ConCISe is its smart compiler, capable of automatically partitioning an application into software and hardware images, and synthesizing the hardware. In this chapter, we will go through the main phases of ConCISe's smart compilation process. Specific formalisms, however, will be left to the next chapter.

5.1 The partitioning at a glance

A complete tool-set has been written for ConCISe. It features a hardware/software partitioning module, a hardware translation module, a hardware synthesis module, an assembler/linker, an instruction set simulator/profiler, and a system simulator. The core of the tool-set is its partitioning and synthesis algorithms. The general objective of the partitioning can be conceptually illustrated as in Figure 5.1. The instances of the PSI in the code are the "glue" that connects the modified assembly to the hardware partition. The other modules are used to evaluate the approach. The simulators, in particular, are used to make up for the current lack of a silicon implementation of ConCISe. Note that it is not enough to connect a standard MIPS to a CPLD on a board to emulate ConCISe, since the data-path integration approach of Figure 4.1 (page 45) requires access to data-path signals not accessible from external pins.

The input chosen for ConCISe's partitioning algorithms is assembly code generated by a standard MIPS compiler. The decision to do so needs some clarification at this point. It is true that some potentially useful information for the partitioning algorithms is lost after code generation and register allocation, already done at assembly level. Examples of this will be discussed later. However, there are two main reasons why to start from assembly code instead of some intermediate code representation internal to the compiler:

1. By starting from assembly code, we isolate the partitioning and synthesis part of the compilation flow from the front-end compiler. This way, an off-the-shelf optimizing compiler can be used to generate the assembly. This has a two-fold advantage: Firstly, the design and construction of the ConCISe-specific part of the tool-set is much facilitated; Secondly, we guarantee a commercial level of quality for the software partition. This will be important when comparing a standard MIPS processor with its ConCISe-extended counter-part. Whatever level of speed-up is observed, we will be able to guarantee that it is not biased by low-quality code generation;

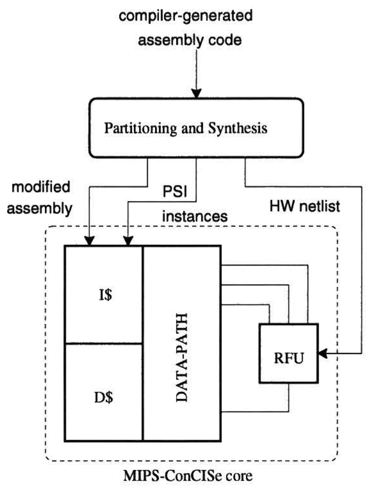


Figure 5.1: Conceptual view of the partitioning process.

2. From a more commercial point of view, the chances of the ConCISe approach and tool-set being taken up by the industry are increased if the tools are isolated from the front-end compiler. Again, there are two main reasons for this: Firstly, a preferred front-end compiler is typically part of the culture of design teams, and is difficult to change; Secondly, a full compiler is a complex piece of software that demands constant maintenance and upgrades to remain competitive. Typically, companies use compilers from third parties, these parties being responsible for all maintenance and product upgrades for a variety of customers. Transferring a full compiler as part of ConCISe's tool-set to a business, would imply the need for that business to maintain a proprietary compiler on its own.

The next section contains an overview of the tool-set.

5.2 Flow overview

Figure 5.2 illustrates the tool-set and the way its modules are connected. It targets a MIPS R3000 as basis platform, and performs the following steps:

1. Source code is processed by a compiler, generating optimized MIPS assembly code;
2. A simulator runs the assembly and produces profile data (basic block execution counts);
3. The assembly code is processed by a hardware/software partitioning module, which looks for data-flow segments, within the basic blocks, potentially suitable for hardware synthesis (these segments are henceforth called *candidates*). This module is

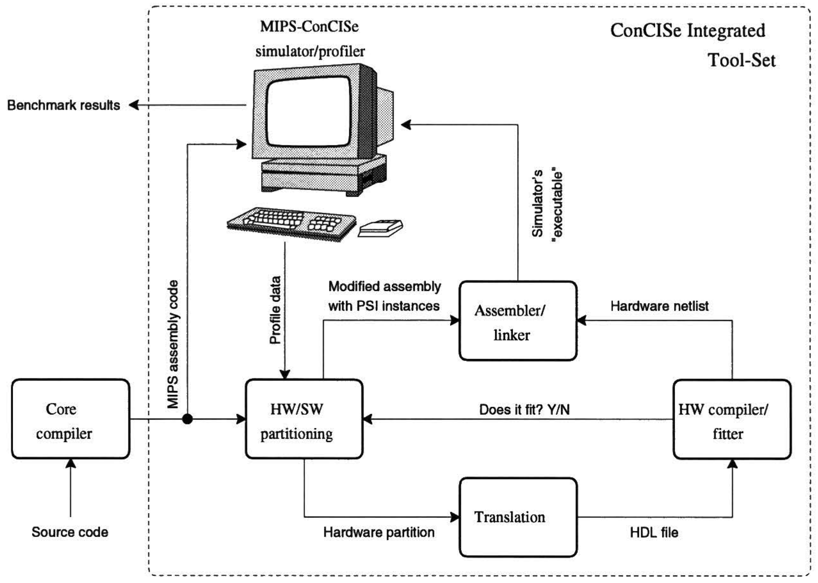


Figure 5.2: ConCISe Integrated Tool-Set, featuring automatic hardware/software partitioning and hardware synthesis.

further described in the sequel, and discussed in detail in the next chapter. Up to 16 candidates are selected in the critical path of the code (identified by the profile data). Each selected candidate will later be replaced by a custom operation represented by a PSI instance. They constitute the application's hardware partition, and are then synthesized together in a single RFU configuration;

4. The hardware partition, still made up of segments of MIPS assembly code, is then sent to a translator, where the assembly instructions are converted into a hardware description in HDL. Decoding logic (a multiplexer, whose select word is DEC) is added, such that the different custom operations may be executed independently. Details about the translation module are given later in this chapter;

5. The resulting circuit description is processed by a hardware synthesis tool. A fitting report is generated, as well as a circuit netlist. The synthesis tool is based on the commercial XPLA2 tool-set, which has been re-targeted towards a ConCISe RFU;

6. If the hardware partition does not fit in the available CPLD resources, the maximum number of custom operations allowed in it is reduced by one, and an entirely new selection starts. The cycle repeats until the hardware partition fits or no custom operation is left;

7. The data-flow segments in the assembly code that are part of the finally selected

hardware partition are replaced by their equivalent PSI instances. The DEC field in each instance identifies the particular data-flow segment it is replacing;

8. The resulting assembly is sent to a modified assembler that recognizes the newly added PSI instances. The netlist generated by the hardware synthesis tool is combined with the assembled machine code, producing the final executable;

9. The executable is then run by a ConCISe simulator. When a PSI instance is to be executed, the logic operations specified in the hardware netlist are simulated at the logic gate level. Finally, benchmark results are produced.

Note that, again, the regular and predictable timing model of the CPLD considerably facilitates the partitioning process (item 3 above). Grouping multiple candidates in a single configuration could sensibly change their individual implementation delays in a typical FPGA structure. This would render the automatic hardware/software partition fairly more difficult, since the algorithms would need to take this (unpredictable) delay variations into account when selecting candidates (remember, the RFU has a single CPU clock cycle to execute). With the CPLD, adding more custom operations to the hardware partition simply requires more PTs. As long as the circuit fits in the RFU, the delay will be constant, and need not be taken into account by the partitioning algorithms.

The tool-set can be run fully automatically (e.g. from within a script), or in an interactive mode that allows the user to monitor and/or control the optimizations being done. Figure 5.3 illustrates a screen-shot of the tool-set.

5.3 Detection of candidates

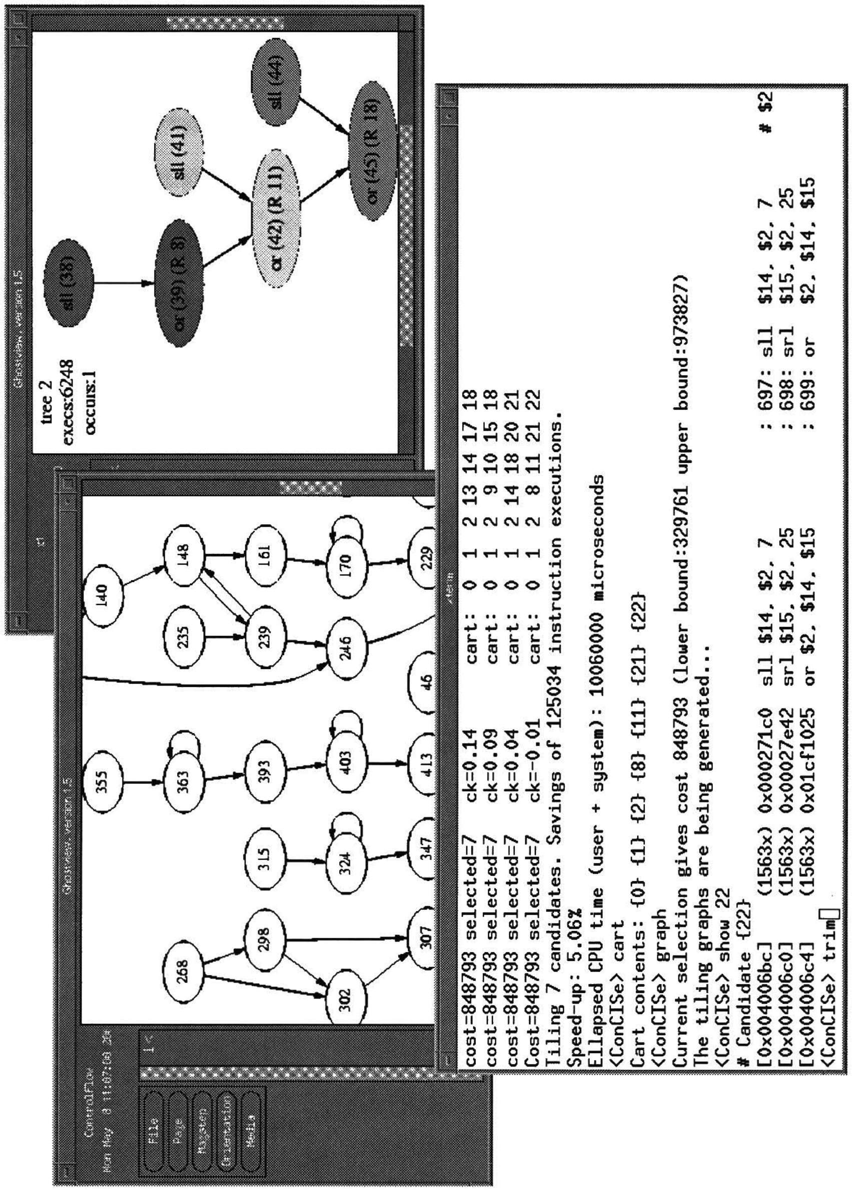
The partitioning is divided in two phases: detection of candidates, and selection of candidates. In this section, we will concentrate on the former.

5.3.1 Building the application's control-flow graph

It is well-known that an assembly program can be split into data and control flows [62]. The control-flow can be represented by a graph in which vertices are basic blocks (straight-line sequences of code that can be entered only at the beginning and exited only at the end), and edges are the possible flows of control between basic blocks. The ConCISe tool-set pre-processes its input assembly file to detect the basic blocks and construct the corresponding control-flow graph. Figure 5.4 illustrates the control-flow graph produced by the tool-set for the RSA Data Security Inc. MD5 message-digest algorithm, commonly used as a benchmark in the cryptography world.

5.3.2 Detecting synthesizable MaxMISOs

The data-flow computations within each basic block can be represented by Directed Acyclic Graphs (*DAGs*) [62], henceforth also referred to as *data-flow graphs*, or simply *data-flow*, in which vertices represent instructions and edges represent data dependencies



```
cost=848793 selected=7   ck=0.14      cart:  0  1  2 13 14 17 18
cost=848793 selected=7   ck=0.09      cart:  0  1  2  9 10 15 18
cost=848793 selected=7   ck=0.04      cart:  0  1  2 14 18 20 21
Cost=848793 selected=7   ck=-0.01     cart:  0  1  2  8 11 21 22
Tiling 7 candidates. Savings of 125034 instruction executions.
Speed-up: 5.06%
Ellapsed CPU time (user + system): 10060000 microseconds
<ConCISe> cart
Cart contents: {0} {1} {2} {8} {11} {21} {22}
<ConCISe> graph
Current selection gives cost 848793 (lower bound:329761 upper bound:973827)
The tiling graphs are being generated...
<ConCISe> show 22
# Candidate {22}
[0x004006bc]    (1563x) 0x000271c0  sll $14, $2, 7          : 697: sll  $14, $2, 7       # $2
[0x004006c0]    (1563x) 0x00027e42  srl $15, $2, 25         : 698: srl  $15, $2, 25
[0x004006c4]    (1563x) 0x01cf1025  or $2, $14, $15         : 699: or   $2, $14, $15
<ConCISe> trim
```

Figure 5.3: Screen-shot of the ConCISe integrated tool-set running in interactive mode. A command window is shown alongside two monitoring windows. The monitoring windows are ghostscript-based, graph representations of the control and data-flow in the application program. In the latter, ConCISe optimizations can be monitored as they are carried out.

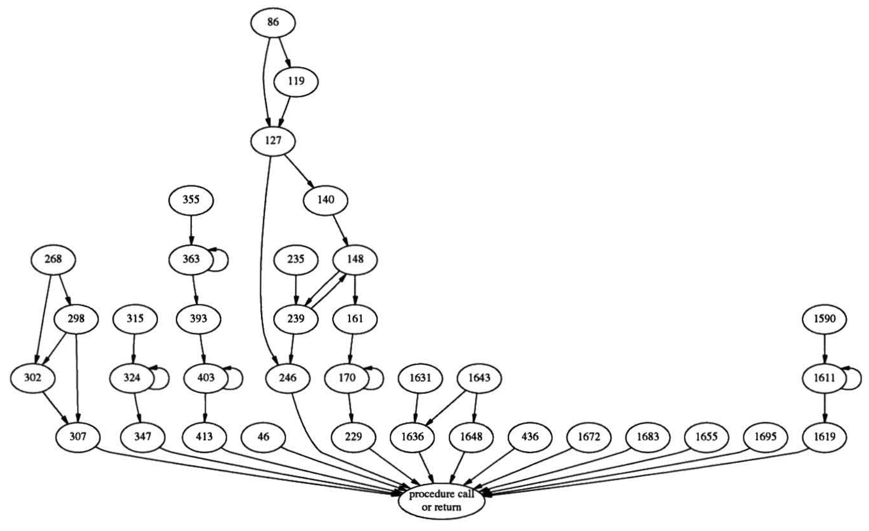


Figure 5.4: The control-flow of the MD5 algorithm, as given by the ConCISe tool-set. Each vertex represents a basic block whose number is the line number of the basic block's leader instruction in the source assembly.

between instructions. This way, a data-flow graph will correspond to each basic block in an application program. Candidates are sub-graphs of a data-flow. They have at most two inputs and a single output (only sub-graphs like this can be replaced by a 3-operand instruction in the MIPS data-path). Because ConCISe targets the hardware acceleration of bit manipulation segments of an application, candidates can contain only MIPS assembly instructions related to the manipulation of bits. These are: `and`, `andi`, `lui`, `nor`, `or`, `ori`, `sll`, `sra`, `srl`, `xor`, and `xori`. We call them *synthesizable instructions*, as they are the only ones the tool-set considers to be valid for hardware synthesis in the RFU. Note that no carry-based instructions (like additions and multiplications), memory instructions (loads and stores), and control-flow instructions (like jumps and branches) can be part of a candidate. They would either be too complex for the RFU logic (possibly not fitting in the available resources), or would require a different microarchitecture approach for the RFU (in the case of control-flow and memory operations).

Using the same terminology introduced in [29], let a *MISO* (for "Multiple Input, Single Output graph") be a sub-graph of a data-flow, such that this sub-graph may have multiple inputs but only a single output. Let also a *MaxMISO* be a MISO that is not contained in any other MISO[1]. In the context of this work, we will only be interested in MISOs and MaxMISOs made up exclusively from synthesizable instructions, so-called *synthesizable MISOs* and *MaxMISOs*. From this point on, every MISO and MaxMISO discussed in this chapter will be assumed to be synthesizable.

[1]Formal definitions for MISOs and MaxMISOs will be given in the next chapter.

It is easy to see that all candidates in an application program will be 2-input MISOs contained in MaxMISOs, or will be MaxMISOs themselves. For this reason, the detection of candidates is based on the search for MaxMISOs. After the detection mechanism constructs the application control-flow graph, a bottom-up search algorithm is applied to each basic block in order to look for its MaxMISOs. For instance, consider the following segment of MIPS assembly code, part of a basic block:

```
lw $15, 68($sp)
srl $24, $15, 16
and $14, $24, 255
sll $25, $14, 24
lw $10, 64($sp)
and $9, $10, 255
or $11, $25, $9
srl $24, $10, 16
and $14, $24, 255
sll $25, $14, 8
or $9, $11, $25
and $24, $15, 255
sll $14, $24, 16
or $11, $9, $14
sw $11, 44($sp)
```

The equivalent data-flow is illustrated in Figure 5.5. Note that not only the instruction's mnemonic, but also its immediate value, if any, are necessary to specify the computation associated with a vertex in the graph. The detection starts from the first synthesizable instruction it finds in the basic block, bottom-up, starting from the lastly scheduled instruction in the block. In this case, it is the `or` instruction nearest to the bottom of the assembly segment. Then, the detection proceeds upwards along the data dependencies until one of the following stop criteria is met:

1. the next instruction read is non-synthesizable (e.g. a `lw` or an `add`);
2. the next instruction read is part of a previously detected MaxMISO;
3. there is an attempt to read beyond the beginning of the basic block; or
4. the next instruction read has a non-reconvergent fan-out, i.e. it would lead to multiple outputs.

The entire process is repeated for all other instructions in the basic block, bottom-up, which are not yet part of a MaxMISO.

Figure 5.5 illustrates the MaxMISO detected in the assembly segment above. It has two inputs (read by four different instructions), and one output (written by the `or` instruction at the bottom, and subsequently stored in memory). Note that the MaxMISO

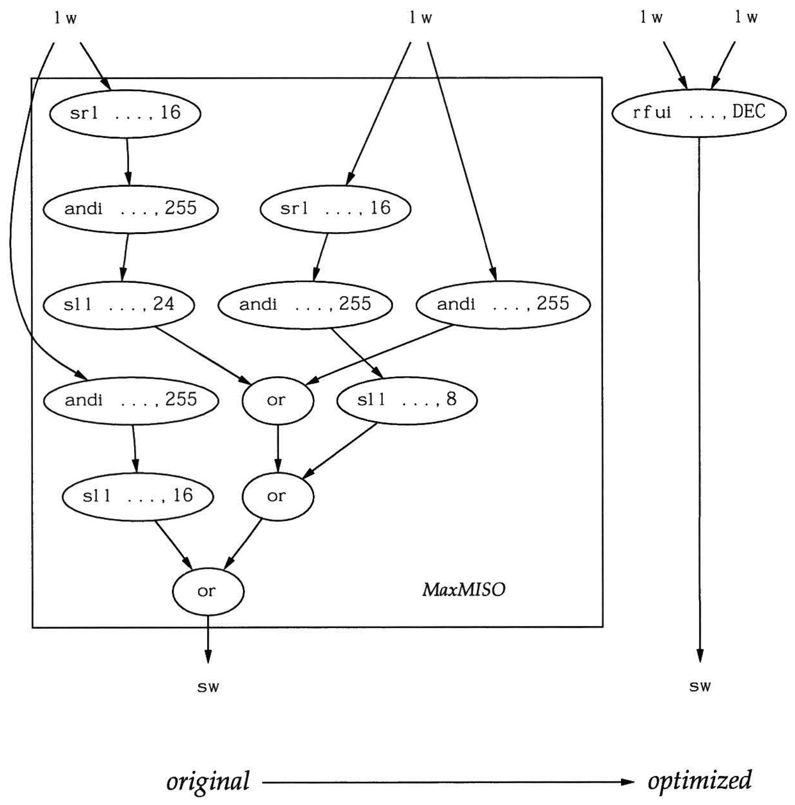
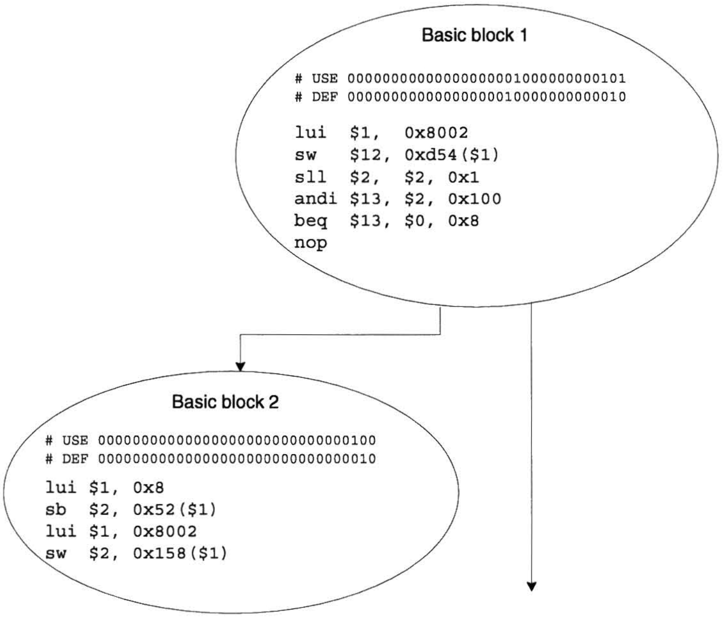


Figure 5.5: Example of ConCISe's data-flow optimizations. A candidate data-flow segment consisting of several native instructions is replaced by an instance of the RFU instruction that executes in a single clock cycle.

itself is a candidate, which, if selected for the hardware partition, will have all native instructions in it replaced by a single instance of a PSI instruction with an associated DEC value. This is illustrated in the right-hand side of Figure 5.5. Equivalent data-flow computations will then be synthesized in the RFU hardware. While the original segment would take several cycles to execute (at least one cycle per instruction, assuming no pipeline stalls), the PSI instance, implementing the same functionality, will execute in a single cycle. This is only possible because the spatial RFU implementation can exploit the available bit-level parallelism in the data-flow segment (Section 1.3.2) and benefit from boolean-level logic optimizations.

Graph-based techniques can be used to guarantee re-convergence (stop criterion 4 above) within a basic block (see Section 6.2.2). However, there can also be data dependencies across basic block boundaries. Figure 5.6 illustrates two basic blocks of the



```
# USE 00000000000000000001000000000101
# DEF 00000000000000000010000000000010

lui  $1,  0x8002
sw   $12, 0xd54($1)
sll  $2,  $2, 0x1
andi $13, $2, 0x100
beq  $13, $0, 0x8
nop
```

```
# USE 00000000000000000000000000000100
# DEF 00000000000000000000000000000010

lui $1, 0x8
sb  $2, 0x52($1)
lui $1, 0x8002
sw  $2, 0x158($1)
```

Figure 5.6: Cross-basic-block dependency in the Magenta benchmark.

Magenta encryption algorithm [58]. Register **$2** is written by the `sll` instruction in basic block 1 and immediately read by the `andi` instruction. No other instruction in basic block 1 reads register **$2** subsequently. However, the value stored in the register is indeed read in basic block 2 by the `sb` (store byte) instruction. If the pair of instructions `sll` and `andi` in basic block 1 turned out to be a selected candidate, register **$2** would no longer be written, since it would become a temporary value internal to the new hardware-based operation. Consequently, the `sb` instruction would not have the right value to store, causing an error.

To tackle this problem, the ConCISe tool-set has an extra criterion to prevent the collapse of temporary values that are used across basic block boundaries. Each basic block is annotated with two 32-bit masks, DEF and USE. Each bit in the masks corresponds to a register, the least significant bit corresponding to register **$0**. If a register is read in a basic block before it is written, its corresponding bit in USE is set. If a register is written in a basic block before it is read, its corresponding bit in DEF is set. To deal with function calls and returns, the tool-set creates ghost basic blocks in which DEF and USE are defined according to the MIPS convention for caller- and callee-saved registers. Figure 5.6 shows both masks for both basic blocks.

The way in which the algorithm uses the masks is as follows. First, it identifies the register that is written by the assembly instruction in consideration. For that register, it goes down the control-flow graph, checking the masks DEF and USE. The instruction can only be part of the MaxMISO currently being detected if, for all paths in the control-flow starting at the basic block in question, a DEF precedes a USE (if any) for that register. Note that, in Figure 5.6, a USE for register \$2 is found already in the next basic block of the control-flow graph. Therefore, the `sll` can only be the first instruction (bottom-up) of another MaxMISO, not the one starting at the `andi` instruction.

5.3.3 Deriving MISOs from the MaxMISOs

The finally detected MaxMISOs are DAGs. For reasons discussed in the next chapter, MaxMISO DAGs are unraveled into directed trees[2] [66], and only these trees are further processed by the tool-set. The MaxMISO of Figure 5.5 is already a tree, and need not be unraveled. Naturally, directed trees are a sub-set of DAGs.

At this stage, all MaxMISOs are internally represented as trees. The algorithm then extracts every sub-tree that can be derived from each MaxMISO. By definition, these sub-trees are MISOs[3]. Figure 5.7 illustrates two of the MISOs that can be derived from the MaxMISO in Figure 5.5. A list containing all detected MaxMISOs and all derived MISOs is built.

Each element in the list then goes through a filtering process. Elements with more than two inputs are discarded, since no more than two source operands are allowed in the R3000 pipeline. Of the two derived MISOs illustrated in Figure 5.7, one would be filtered out for having three inputs (check it out). Elements made up of a single instruction are also discarded, since no speed-up is achieved if one native instruction is replaced by one PSI instance. The final list contains only trees representing data-flow segments that can be replaced by a PSI instance. These trees are the detected candidates.

Figure 5.8 summarizes the detection mechanism.

5.4 Selection of candidates

Detected candidates go through a selection process, described below.

The input code can be decomposed into code that is contained in MaxMISOs and code that is not. Candidates can only appear within MaxMISOs. Therefore, to optimize the entire code using a set of candidates, it is sufficient to look at MaxMISOs only. An *occurrence* of a candidate is one specific appearance of a candidate within a certain MaxMISO, that is, it is a specific sub-tree of a MaxMISO equal to a candidate. For example, Figure 5.9 illustrates three MaxMISOs and three candidates. Candidate 2 has two occurrences, one in MaxMISO A and one in MaxMISO B. Note that candidate MISOs may have occurrences in MaxMISOs other than the ones they were originally derived from. Candidates may also have occurrences in MaxMISOs that are themselves not valid candidates.

A *tiling* of a MaxMISO is a set of candidate occurrences within the MaxMISO, such that none of the occurrences overlap. See MaxMISO A in Figure 5.9 for an example.

[2]In this dissertation, unless explicitely mentioned otherwise, every tree is assumed to be a directed tree.

[3]All directed paths in a directed tree converge to a single vertex, called the *root* of the tree. Therefore, directed trees always have a single output vertex, its root, and are MISOs.

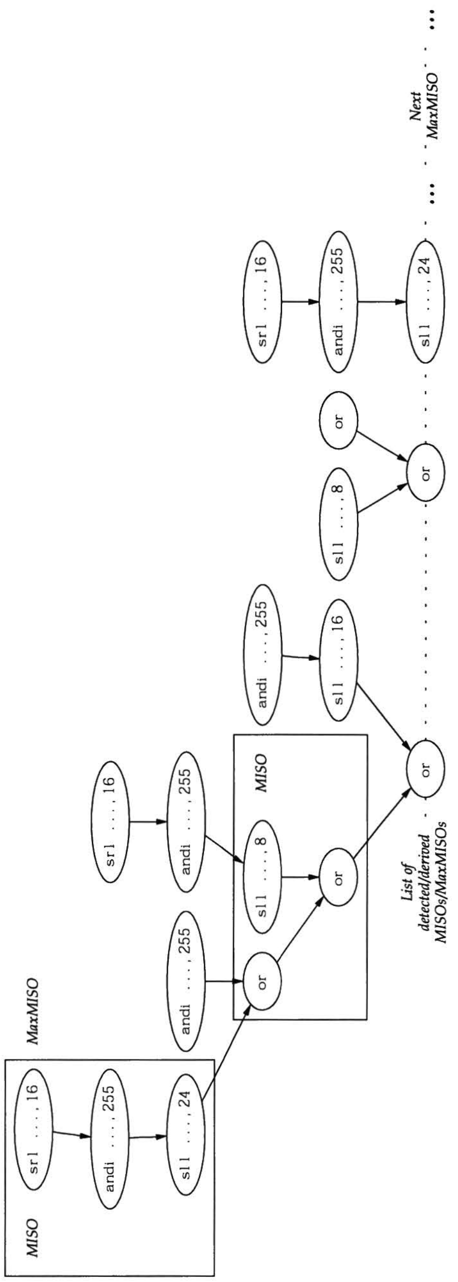


Figure 5.7: Deriving MISO sub-trees from the MaxMISO of Figure 5.5, and adding them to the list of potential candidates. Only two of the various MISOs that can be derived from that MaxMISO are shown.

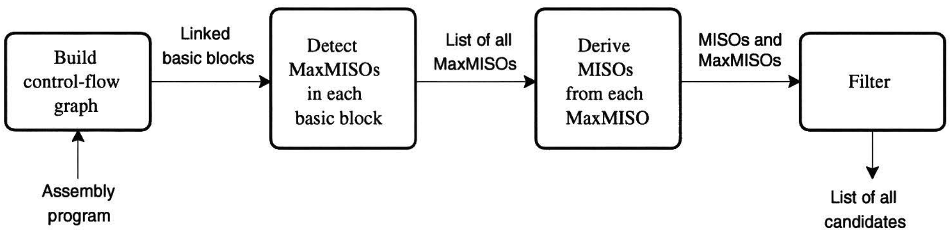


Figure 5.8: ConCISe's detection mechanism.

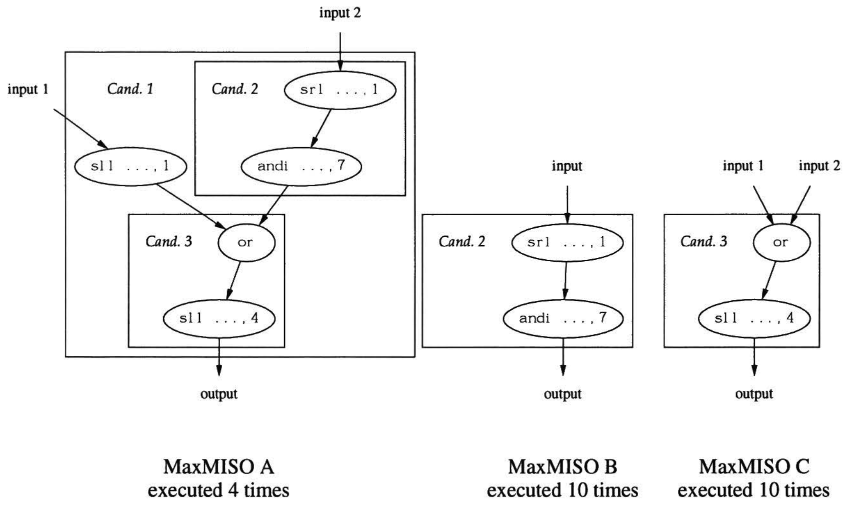


Figure 5.9: An example program consisting of three MaxMISOs.

The set containing the shown occurrences of candidates 1 and 2 is not a tiling, because they overlap. The set containing the shown occurrences of candidates 2 and 3 is a tiling, because they do not overlap. A tiling can be translated into optimized assembly code by replacing candidate occurrences in the tiling by their corresponding PSI instances. All other instructions are preserved. The *tiling cost* of a MaxMISO with a given tiling equals the number of candidate occurrences in the tiling (i.e. the number of PSI instances it will have) plus the number of instructions that are not covered by any candidate occurrence in the tiling (i.e. the remaining native instructions)[4]. Given a set of selected candidates, the task of the so-called *tiler* is to find a tiling of minimal tiling cost for each MaxMISO.

[4]As we will see in the next chapter, the tiling cost must be calculated only after the unraveled trees are fold back to the domain of DAGs. In the examples considered here, however, all MaxMISOs are originally trees and needed not be unraveled, so this can safely be ignored for the time being.

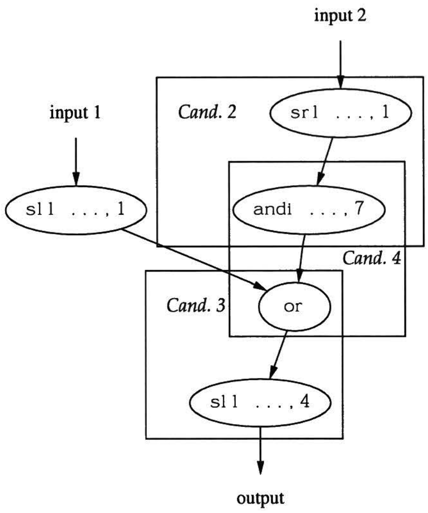


Figure 5.10: A MaxMISO showing occurrences of three candidates.

5.4.1 The tiler

Using pattern matching, the tiler looks for a minimal tiling cost *cover* of the MaxMISOs with candidates. *Dynamic programming* can generally be applied to a tiling process when the following *optimality principle* holds [62]: if the tilings of sub-trees of the given MaxMISO have been solved optimally, the optimal tiling of the entire MaxMISO can be achieved by a particular method of combining the optimal solutions for the sub-trees. However, because the MaxMISO and candidate trees are unraveled versions of information originally in the DAG domain, the optimality principle does *not* apply to our cost function. This will be discussed in the next chapter.

Therefore, our tiler cannot use dynamic programming. Instead, given a set of candidates, the tiler performs a nearly-exhaustive search for different tiling possibilities, using a branch-and-bound algorithm to reduce the search space. To provide for some more insight into this process, consider the MaxMISO in Figure 5.10. The pattern-matcher finds occurrences of three different candidates in it. This way, there are four different, possible tilings: Three tilings using a single candidate each, and one tiling using candidates 2 and 3 together (since they do not overlap). Clearly, the minimal tiling cost (3) is achieved when tiling with candidates 2 and 3, and ignoring candidate 4.

It is easy to see that the complexity of the tiler is exponential with the size of the MaxMISO, and linear with the number of MaxMISOs.

5.4.2 The selection engines

The limited set of candidates ultimately selected to be synthesized makes up the hardware partition. The aim of the selection process is to come up with a hardware partition from which maximum speed-up can be achieved. We examined two heuristic selection mechanisms to do this: a greedy selector and a selector using simulated annealing. Both mechanisms try to find a solution with minimal total cost, where *total cost* is defined as the summed instruction execution count of all MaxMISOs. The execution count of a MaxMISO equals the number of times the MaxMISO is executed, multiplied by its tiling cost (i.e. the number of instructions in it, PSI or native). The selection engines try to select candidates for the hardware partition such that the total cost is as small as possible after a tiling run. Selected candidates become RFU custom operations.

The greedy selection engine.

The greedy selector runs through all unselected candidates and, for each candidate, after calling the tiler, computes the new total cost if that candidate alone had been chosen. The candidate that results in the largest cost reduction is selected. The process is repeated until the maximum number of candidates allowed in the hardware partition is selected.

The simulated annealing selection engine.

In each iteration of the simulated annealing selector, one of three actions is randomly chosen (using a predefined probability distribution): *add*, *remove* or *swap*. If the maximum number of candidates in the hardware partition has not yet been reached, the *add* action selects a candidate not present the hardware partition and computes the difference in total cost (naturally, the tiler is always used before computing the cost). If the *remove* action is chosen and the hardware partition is non-empty, one of the candidates in the hardware partition is randomly chosen and the cost difference of removing this candidate is computed. If *swap* is chosen, a candidate randomly chosen from the hardware partition is replaced by a candidate not in the hardware partition, and the corresponding difference in cost is computed. If the action results in a cost reduction, it is accepted; If it results in a cost increase, the action is accepted with the following probability:

$$P(accept) = \exp(\frac{-\Delta cost}{T})$$

where $\Delta cost$ is the cost difference, and T is the "temperature", which slowly decreases throughout the iterations. If the temperature is high, large cost increases are accepted. The probability of acceptance of an action that increases the cost, reduces with the temperature. In the end, only cost reductions are accepted.

Depending on the application program, the annealing algorithm can potentially give better results than the greedy algorithm. This is because annealing can try more possible combinations of candidates, whereas greedy inspects only one combination. Figure 5.9 shows the three MaxMISOs of an example program. MaxMISO A is executed four times, B and C are executed ten times. The figure shows three candidates. Candidates 2 and 3

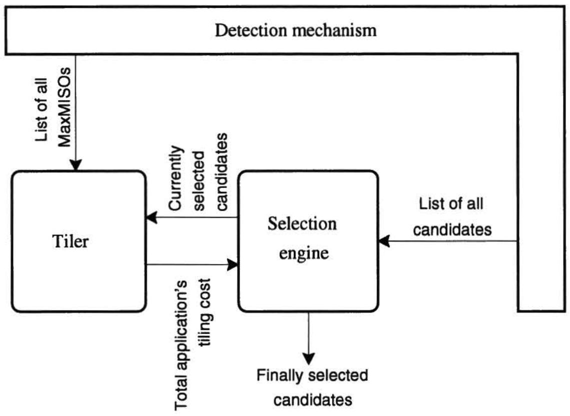

Detection mechanism
List of all MaxMISOs
Tiler
Currently selected candidates
Total application's tiling cost
Selection engine
List of all candidates
Finally selected candidates

Figure 5.11: ConCISe's selection mechanism.

both have two occurrences, one as a sub-tree in MaxMISO A, and one covering either the entire MaxMISO B or C. Note that many more sub-trees of MaxMISO A are candidates, but they are not shown.

Assume that we are allowed to select two candidates. The greedy algorithm first selects the candidate with the highest total cost reduction. It can choose between candidates 1, 2, and 3, and all candidates which are sub-trees of A but are not shown in Figure 5.9. Candidates 2 and 3 are both executed $10 + 4 = 14$ times, and could give a cost reduction of 14 (two instructions are replaced by one). The selected candidate is candidate 1, with a cost reduction of 16, which is the execution count of A (four) times the number of saved instructions (five minus one instruction to replace candidate 1). The greedy algorithm can then choose between the remaining candidates 2 and 3. Both have occurrences in MaxMISO A, but the tiler will decide that it is cheaper to tile MaxMISO A with candidate 1, instead of using candidate 2 or 3 and leaving the remaining instructions untouched. Therefore, both candidates 2 and 3 result in an equal additional cost reduction of 10, so the greedy algorithm simply picks one of them and finishes with a total cost reduction of 26. The annealing algorithm, on the other hand, has more freedom to select combinations of candidates and could also try a hardware partition made up of candidates 2 and 3 (although no guarantee is given). This would give a cost reduction of 28, which is better than the result of the greedy algorithm.

Figure 5.11 summarizes the selection mechanism of ConCISe. The process illustrated in the figure is, to the best of the author's knowledge at the time this thesis was written, an innovation in the field of RC.

Proposition 5.4.1. *Known techniques from the field of compiler technology can be successfully used in RC. One main example is the application of graph covering techniques (tiling), typically used in code generation, to the problem of automatic hardware/software partitioning.*

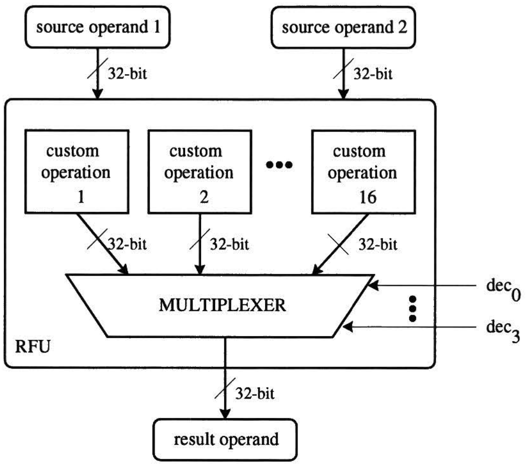


Figure 5.12: Encoding multiple custom operations in a single RFU configuration.

5.5 Translation and synthesis

The logic functions in the selected candidates are translated into equivalent statements in Philips Hardware Description Language (PHDL) [45], a sub-set of ABEL [46], formerly used for the XPLA2 CPLDs. The translation process is straight-forward: to each assembly instruction will correspond an equivalent logic operation in the hardware description. Details on how this is implemented can be found in [50].

Each selected candidate becomes a custom operation in the hardware partition. A multiplexer is added, controlled by the DEC signals, which are responsible for run-time selection of the operation within the partition that is to be executed. Depending on the value of DEC, the result of one of the custom operations is forwarded to the output. At the HDL level, the architecture of the hardware partition is presented in Figure 5.12 (compare to Figures 4.1 and 4.3). The architecture in the figure, however, is not necessarily preserved after hardware synthesis. All custom operations in the partition (and the multiplexer) are logically minimized by the hardware synthesis tool. Again, cross-minimization may occur when different custom instructions share a common PT (see Chapter 8).

The hardware synthesis module is based on the commercial XPLA2 tool-set. The PHDL front-end and technology mapping and optimization routines could be re-used without changes. The fitting routines, however, have been modified as to target a ConCISe RFU. As mentioned before, a simplifying assumption is made during fitting that both switches are 100% routable[5].

[5]For the size estimation of the RFU, however, we considered a true XPLA2 virtual cross-point switch, which is smaller than a fully connected one.

5.6 ConCISe and prior works

It is my hope that, at this point, the reader can clearly perceive the main advances in compilation and hardware/software partitioning technology ConCISe introduces, when compared to RTR-intensive approaches like PRISC. This has been illustrated, one way or the other, throughout this and the previous chapter. Still, to make the point clear, one could look upon PRISC's compiler as a ConCISe compiler in which the following limitations are added:

1. There is no selection procedure. Every candidate detected simply becomes a custom operation;
2. There is no search for kernel candidates that occur in multiple segments of the input code;
3. There can be no more than one candidate detected in each loop body;
4. There is no tiling. A candidate is never used to cover segments of the code other than the one where it was originally detected.

Currently, however, ConCISe does not implement the control-flow optimizations PRISC does (see Section 3.4, page 35). This is not an intrinsic limitation of our approach, since the tool-set can well be extended to perform those optimizations. We believe, however, that in the target application domain of ConCISe, the data-flow optimizations it performs suffice to account for most of the main speed-up opportunities.

The main differences between ConCISe, DISC, and Chimaera have also been discussed throughout, and will not be re-emphasized here. ConCISe and PRISM differ in their very concepts and objectives, and the differences need not be discussed. Finally, when compared to Garp, ConCISe searches for simpler hardware partitions with a finer granularity. The respective tool-sets reflect this basic difference. While Garp's compiler tries to map segments of the application's control-flow, potentially loops or entire state machines, onto the FPL array, ConCISe's tool-set concentrates on the data-flow.

Generally speaking, ConCISe's two most distinguishing characteristics when compared to prior works are the encoding of several custom operations in a single FPL configuration, and the use of partial graph covering techniques during hardware/software partitioning. These two basic concepts unfold in other innovative techniques at various levels in the tool-set, as discussed in chapters 6 and 8. In particular, besides formalizing many of the ideas discussed here, the next chapter introduces the concept of DAG-to-tree unraveling for partitioning. The opportunity for the development and application of this concept arose from the use of graph covering techniques in ConCISe.

6 Formalizing ConCISe's partitioning techniques

In the previous two chapters, ConCISe was introduced, discussed, and evaluated without the use of formalisms. The idea was to provide the reader with a perspective of the approach as a whole and the main propositions behind it.

In this chapter, I introduce the formalisms necessary to describe the techniques and ideas behind ConCISe in an unambiguous way. I also discuss a few formal results that prove the correctness of some of the innovative partitioning techniques used.

Although both chapters 5 and 6 go about the same subject, they hardly overlap. The information provided here augments what has already been discussed in the previous chapter.

6.1 Problem representation and basic definitions

Graph theory will be the basic tool of our discussion. The idea is to represent an application program by means of graphs. The partitioning problem then becomes that of selecting sub-graphs of the program graph that are to be part of the application's hardware partition.

As we have seen in the last chapter, the data-flow within basic blocks can be represented by a graph. In this case, vertices are instances of native instructions and edges are data-dependencies between the instances. Naturally, the edges are directed, which reflects the orientation of the dependencies. An entire program can then be represented by a control-flow graph where each vertex represents a data-flow graph.

From the definition of a basic block, the data-flow graphs that represent them will always be acyclic. These directed and acyclic graphs are referred to as DAGs. A DAG, represented by the capital letter D, is defined by a pair of sets $D = (E, V)$, where V is the set of vertices, and E is the set of edges connecting vertices. We may also refer to the set of vertices as $V(D)$, and to the set of edges as $E(D)$. Elements of E are pairs of vertices, thus satisfying $E \subseteq V \times V$. The edge (k, l) is an *outgoing edge* of vertex k, and in *incoming edge* of vertex l. If an edge is an incoming or outgoing edge of a vertex k, it is said to be *incident* with k. This is illustrated in Figure 6.1.

In any connected graph, there are no vertices without incident edges, so $V(D)$ can always be reconstructed from $E(D)$. For this reason, a DAG can be fully defined by making $D = E$. We will utilize this property in several of the definitions that follow.

The predecessor set of a vertex $l \in V(D)$ is the set:

$$pred_D(l) = \{k \mid (k,l) \in E(D)\}$$

Similarly, the successor set of a vertex $k \in V(D)$ is the set:

$$succ_D(k) = \{l \mid (k,l) \in E(D)\}$$

In figure 6.1, k is a *predecessor* of l, and l is a *successor* of k, i.e. $k \in pred_D(l)$ and $l \in succ_D(k)$. For a vertex v of a graph D, if $|succ_D(v)| > 1$, then v is said to be a *multi-dependency vertex*, or *multi-dep vertex.* In figure 6.1, k is the only multi-dep vertex.

A *path* connecting two vertices k and l in D is a finite sequence of vertices starting at k and finishing at l, in which no vertex is repeated, such that each vertex in the sequence is joined to the next vertex in the sequence by an edge. In Figure 6.1, the sequence $[k, l]$ is a path connecting k to l, as is $[k, m, n, l]$. Note that paths need not necessarily be directed, i.e. the sequence of vertices need not obey the orientation of the edges connecting them. A *directed path*, however, does obey the orientation of the edges. In Figure 6.1, there are two directed paths connecting k to n, but only one connecting k to l.

A vertex i can be concatenated to a directed path $[k, l, n]$ iff there is an edge (i, k) connecting i to k. The concatenation operation is represented by the symbol $+\!\!+$. This way, $i+\!\!+[k, l, n] = [i, k, l, n]$, a directed path connecting i to n. See Figure 6.1 again.

If $E(D') \subseteq E(D)$ we say that D' is a *sub-graph* of D. Similarly, if $E(D') \subset E(D)$, we say that D' is a *proper sub-graph* of D.

If D' is a proper sub-graph of D, a vertex $v \in V(D')$ is an *input vertex* of D' in D iff it has an incoming edge that is an edge of attachment of D' in D, that is, iff $pred_D(v) \setminus D' \neq \emptyset$. Similarly, the vertex v is an *output vertex* of D' iff $succ_D(v) \setminus D' \neq \emptyset$. Figure 6.2 illustrates a DAG P with a proper sub-graph M, in which the edges of attachment of M in P are dashed. The edge (p, i) is an input edge of M in P. Similarly, the edge (n, x) is an output edge of M in P.

In our problem representation, for the sake of simplicity, the notation of a vertex abstracts from the particular computation that vertex represents. As we have seen in the previous chapter, the computation of a vertex can be completely specified by the native instruction mnemonic it represents, plus its immediate value, if any (see again Figure 5.5, on page 60). Therefore, from this point on, we assume that each vertex v will have an implicit *computation property,* denoted *comp*(v), associated to it. This computation property is defined by the mnemonic and immediate of the native instruction instance represented by v. For instance, if v represents the assembly `srl $8, $9, 16`, then *comp*(v) = {`srl, 16`}.

More details about the basic definitions of graph theory introduced in this section can be found in [66] or [67].

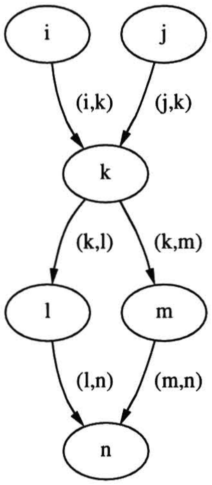


Figure 6.1: Basic notation of a DAG.

6.2 Formalizing MISOs and MaxMISOs

The following definitions are a formalization of concepts introduced in the last chapter, based on the problem representation discussed in the previous section. They will soon be very useful.

6.2.1 Definitions

Definition 6.2.1. A connected proper sub-graph M of a DAG D is a *MISO* of D iff M contains precisely one output vertex v of M in D. In this case, v is the *MISO output*, denoted $out(D)$.

□

Figure 6.2 illustrates a MISO M in a basic block DAG P. In the figure, $out(M) = n$. The input vertices of M in P are i, j, and l. MISOs will generally be denoted by the capital letter M, and the DAG that represents the entire data-flow in a basic block will generally be denoted by the capital letter P.

Lemma 6.2.2. *A MISO M is a directed tree iff there are no multi-dep vertices in $M \setminus out(M)$.*

Proof. If there are no multi-dep vertices in M, except perhaps for $out(M)$, then there can be no more than one path joining every two distinct vertices of M. Hence, M must be a directed tree (see [66]).
□

In general, we denote a MISO tree by the capital letter T. Note that directed trees are a sub-set of DAGs.

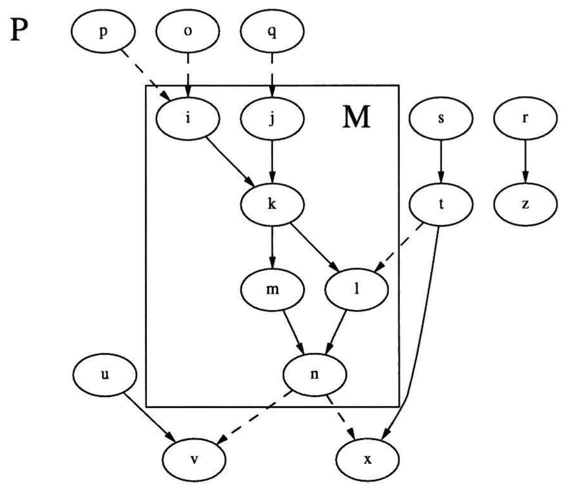


Figure 6.2: A MISO as a connected sub-graph of a basic block. The edges of attachment are shown as dashed lines.

Definition 6.2.3. A MaxMISO MM of a basic block DAG P is a MISO of P that is not contained in any other MISO of P. That is, $\forall M \mid M \subseteq P : MM \not\subset M$.

□

It follows immediately that, for all MISOs M of P, there will be at least one MaxMISO MM of P such that $M \subseteq MM$. MaxMISOs will generally be denoted by the capital letters MM.

6.2.2 Detection

After building the application's control-flow graph, the ConCISe tool-set starts the search for candidates in each of the basic blocks. The search continues until each synthesizable instruction in the block is assigned to a MaxMISO. Therefore, for every basic block DAG P, the search finds a set of MaxMISOs containing all vertices of P corresponding to a synthesizable instruction. See Section 5.3, on page 56.

More formally, given P, the problem is to find a set of MaxMISO sub-graphs of P containing all vertices that correspond to a synthesizable instruction, and only those vertices. These are the *synthesizable MaxMISOs* and, from this point on, all MISOs and MaxMISOs discussed will be assumed to be synthesizable.

A MaxMISO MM cannot contain a vertex $v \neq out(MM)$ with a non-reconvergent fan-out, since this would lead to a graph with multiple outputs. In [29], a recursive algorithm is proposed to detect a MaxMISO MM in P. It counts the number of times a vertex v is visited when traversing P bottom-up along its edges, starting from $out(MM)$. If v is visited as many times as $|succ_P(v)|$, then $v \in MM$, since this implies re-convergence.

In ConCISe, we adopted a non-recursive approach. In the search for a MaxMISO MM, vertices of P are traversed not according to their data-dependencies (edges), but according to the scheduling order of the assembly instructions they represent. The vertex v corresponding to the lastly scheduled synthesizable instruction, not yet included in any MaxMISO, is visited first. Naturally, this implies:

$$out(MM) = v$$

Let O be a set of vertices called *open dependencies*, initially empty. For each vertex k visited during the traversal, the algorithm checks whether k corresponds to a synthesizable instruction. If not, k is discarded and the traversal continues to the next vertex. If k does correspond to a synthesizable instruction, we first make:

$$O := O \cup \{k\}$$

The algorithm then checks whether:

$$succ_P(k) \subseteq O?$$

If so, this means that all vertices in the successor set of k have already been visited by the algorithm, which implies re-convergence. Thus, if there are no cross-basic-block data dependencies (see Section 5.3), we know that:

$$\{(k, l) \mid l \in succ_P(k)\} \subseteq MM$$

Otherwise, $k \notin V(MM)$ and the algorithm proceeds. Each traversal will correspond to a different MaxMISO. At the end of all traversals, all MaxMISOs in P will have been detected.

The algorithm above may mislead the reader into concluding that the set of MaxMISOs detected is a property not only of the DAG P, but also of the particular scheduling order of the instructions in P. This is not true, as shown in the next two results.

Theorem 6.2.4. *Two MaxMISOs of a basic block DAG P cannot partially overlap in P.*

Proof. It has been proven in [29] that two MaxMISOs cannot partially overlap. The main lines of that proof are adapted and shown here for convenience.

The theorem is proven by contradiction. If two MaxMISOs, MM and MM', partially overlap in P, then there must be at least one vertex $k \in V(P)$ that belongs to MM and MM'. In this case, the alternative possibilities are:

1. $k = out(MM) \vee k = out(MM')$. Then, from definition 6.2.1, there is a MISO M such that $M = MM \cup MM'$. As a consequence, according to definition 6.2.3, the sub-graphs MM and MM' must not be MaxMISOs, contrary to the assumption;

2. $k \neq out(MM) \wedge k \neq out(MM')$. Then, there is a directed path from k to $out(MM)$ and to $out(MM')$. By hypothesis, $out(MM) \notin V(MM')$ and $out(MM') \notin V(MM)$, or there would be a vertex k' meeting the conditions of possibility 1 above. Therefore, k is an output vertex of either MM or MM', contrary to the assumption.

$\square$

Corollary 6.2.5. *The cover of a DAG P with its MaxMISOs is unique.*

Proof. Assume the MaxMISO cover is not unique. Then, because all possible covers must cover all synthesizable vertices in P, there must be at least one vertex $k \in V(P)$ that belongs to two different MaxMISOs: MM, belonging to a first cover of MaxMISOs; and MM', belonging to a second cover. That is, MM and MM' must partially overlap in P, which violates Theorem 6.2.4.

□

6.3 Defining *expand*() and *collapse*()

When a register written by a given instruction is subsequently read by more than one other instruction in the same basic block, a multi-dep vertex in the basic block DAG is necessary to represent this situation. Therefore, directed trees do not suffice to represent basic block segments like the MaxMISOs. Still, directed trees are easier to manipulate and operate upon than DAGs. Because there is precisely one path between any two vertices in a directed tree, recursive traversals do not lead to overlapping visits to vertices of the tree, as they do with DAGs.

In this section, we formally define a transformation from DAG MISOs to directed trees that preserves all the original data-flow information. The objective is to perform all optimizations in the domain of trees. In the ConCISe framework, not only is it easier to do, but it also may lead to higher speed-ups in the optimized code than those obtainable in the domain of DAGs. We shall demonstrate this point later in this chapter.

A directed MISO tree T can be constructed from a DAG MISO D in the following way. Each vertex $v \in D$ leads to one or more vertices $\langle v, p\rangle \in T$, each labelled with a different possible path p from $out(D)$ to v. There is a correspondence between the edges of T and the edges of D. Note that the labelling will eliminate multi-dep vertices. Figure 6.3 illustrates the process. T is said to be the *expanded tree* of D. The symbols "$\langle\rangle$" are used, instead of parenthesis, to differentiate the notation from that of edges. The next definition formalizes all this.

Definition 6.3.1. *expand*() is a mapping from DAG MISOs to directed trees defined by:

$$expand(D) = exp_D(\langle out(D), \epsilon\rangle)$$

$$exp_D(\langle v,p\rangle) = \bigcup (\{(\langle w,q\rangle\,, \langle v,p\rangle)\} \cup exp_D(\langle w,q\rangle) \mid w \in pred_D(v) \wedge q = v {+\!\!+} p)$$

The computation property of the vertices is preserved:

$$\forall v,p \mid \langle v,p\rangle \in V(expand(D)) : comp(\langle v,p\rangle) = comp(v)$$

□

Because D is connected, the traversal in the definition reaches every vertex in D. It is easy to see that the expanded tree $T = expand(D)$ will also be connected. Therefore,

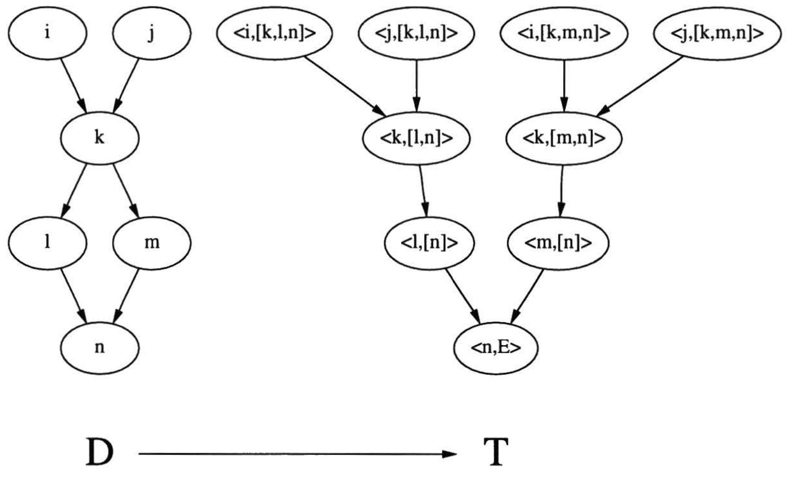


Figure 6.3: A DAG MISO D and its expanded tree T.

generating the set of edges suffices to fully define T, a fact that is exploited in the definition. Because the computation property of the vertices is preserved, it follows trivially that D, and its expanded tree T, both represent the same data-flow computations. From this point on, every tree denoted by the capital letter T will be an expanded tree.

Naturally, we can also define the inverse operation:

Definition 6.3.2. *collapse*() is a mapping from expanded trees to DAG MISOs defined as:

$$collapse(T) = (\{(w, v)\} \mid (\langle w, q\rangle, \langle v, p\rangle) \in E(T))$$

□

collapse() removes the path annotation from each vertex, causing, for instance, two vertices $\langle v, p\rangle \in V(T)$ and $\langle v, p'\rangle \in V(T)$ to be collapsed into a single vertex $v \in V(D)$.

Note that the annotation of expanded tree vertices with paths is only used in the formalism, *not* in the implementation of the algorithms, since that could potentially lead to very large record sizes.

It follows trivially from the definitions above that:

Lemma 6.3.3. *For all DAG MISOs D : $collapse(expand(D)) = D$.*

□

Consider an algorithm that generated the DAG MISO D in Figure 6.3 from its corresponding assembly segment. As described in Section 5.3, page 56, the algorithm should start from the lastly scheduled assembly instruction, going backwards along the data-dependencies and producing a vertex for each new instruction visited. If a dependency lead to an instruction already visited before, the algorithm should create an edge pointing to the vertex previously created, instead of creating a new vertex. In this case, the vertex in question would become multi-dep, as k in Figure 6.3.

With respect to implementation, ConCISe maps DAG MISOs to expanded trees in a straight forward way, during MaxMISO detection (see Figure 5.8). While traversing the assembly in a basic block bottom-up, if a data dependency leads to an instruction for which a vertex has already been created, the tool-set simply creates another vertex for that same instruction and proceeds recursively along the data-dependencies as if it was visiting the instruction for the first time. The effect is exactly that of definition 6.3.1.

The next definition is useful in identifying structurally identical trees that collapse to the same DAG MISO:

Definition 6.3.4. An expanded tree T is said to be *isomorphic* to another expanded tree T' iff there is one bijective function $\theta()$, such that:

$$T' = \{(\langle k, \theta(p)\rangle, \langle l, \theta(q)\rangle) \mid (\langle k, p\rangle, \langle l, q\rangle) \in E(T)\}$$

□

The tree T in Figure 6.4, page 83, has two isomorphic sub-trees. One with root at the vertex $\langle k, [l, n]\rangle$ and the other with root at the vertex $\langle k, [m, n]\rangle$. Note that this definition of isomorphism is more strict than the classical definition. Having said this, every time we use the notion of isomorphism in the remainder of this text will refer to definition 6.3.4.

Lemma 6.3.5. *If T and T' are isomorphic expanded trees, then collapse(T) = collapse(T'). Moreover, if T = expand(D), then collapse$(T') = D$.*

Proof. It follows immediately from definition 6.3.2 and lemma 6.3.3.
□

When formalizing the tiling process discussed in the previous chapter, we will need to assert that two tree sub-graphs are a match, during partial graph covering. To match, two tree sub-graphs need to be equivalent in terms of computation properties. This notion is formalized in the next definition:

Definition 6.3.6. An expanded tree T is said to *match* another expanded tree T' iff there is one bijective function $\omega()$ and one bijective function $\theta()$, such that:

$$T' = \{(\langle \omega(k), \theta(p)\rangle, \langle \omega(l), \theta(q)\rangle) \mid (\langle k, p\rangle, \langle l, q\rangle) \in E(T)\}$$

and:

$$\forall k, p \mid \langle k, p\rangle \in V(T) : comp(k) = comp(\omega(k))$$

□

Assume that the MaxMISO tree in Figure 5.10, page 65, is an expanded tree. The figure shows three different sub-trees in the MaxMISO, each matching another tree corresponding to a different candidate. Unlike isomorphism, the match depends on the computational property of the vertices (illustrated in the figure), not on their names (not shown in the figure).

Note that a match does not imply isomorphism, since matching trees may collapse to different DAG MISOs. However, isomorphism does imply a match. It is also easy to see that both isomorphism and matching are transitive.

6.4 Tiles and tilings

At this point, all MaxMISOs have been detected and are represented by directed trees, instead of DAGs. We now need to put together the set of candidates (see Figure 5.8). For every MaxMISO MM, every different sub-graph of MM with two or more vertices, including MM itself, goes through the filtering process described in Section 5.3. The sub-graphs passing the filtering criteria become candidates. Because every MM is a directed tree, all candidates in the set will also be directed trees. As a result, the optimizations will be performed on the tree representations exclusively. See again Figure 5.11, on page 67.

If two or more candidates, derived from the MaxMISOs by the process described above, match (in the sense of Definition 6.3.6), they represent multiple copies of the same data-flow computation. In this case, only one of the copies suffices and is kept in the set of candidates.

To formally describe the tiling process, we first need a formal way to represent occurrences of candidates in MaxMISOs:

Definition 6.4.1. Let C be a MISO representation of a candidate. A *tile* of a MaxMISO MM, corresponding to C, is a MISO sub-graph $S_{MM} \subseteq MM$ such that S_{MM} matches C.

□

Each tile represents one occurrence of a candidate in MM. Naturally, S_{MM} is connected. We now can formally define a tiling:

Definition 6.4.2. A *tiling* of a MaxMISO MM is a set SS_{MM} of tiles, such that the tiles in the set do not overlap. That is, for any $S_{MM} \in SS_{MM}$, and for any other $S'_{MM} \in SS_{MM}$, it is true that $V(S_{MM}) \cap V(S'_{MM}) = \emptyset$. The tiling, therefore, represents a *partial graph covering* of the MaxMISO.

□

It is convenient to specify a MaxMISO and one of its tilings as a pair:

Definition 6.4.3. A *duet* is a pair (MM, SS_{MM}), i.e. a MaxMISO together with one choice of all its possible tilings.

□

A vertex $k \in V(MM)$ is said to be a *tiled vertex* in the duet (MM, SS_{MM}) iff it is contained in a tile, i.e. iff there is at least one $S_{MM} \in SS_{MM}$ such that $k \in V(S_{MM})$. A *tiled edge* is an edge that is incident with two vertices tiled by the same S_{MM}.

As discussed in the previous chapter, once an expanded MaxMISO tree is tiled, we can replace each tile by a vertex corresponding to its custom operation (see Figure 5.5, on page 60). To properly incorporate the vertices corresponding to custom operations in our formalism, we need the following convention:

Convention 6.4.4. *Let $\vartheta()$ be an injective function whose domain is the set of expanded tree representations of candidates, and whose co-domain is the set of first elements in the pairs that represent vertices of expanded trees. Let C be an expanded tree representation of a candidate. Let also (T, SS_T) be a duet in which T is an expanded MaxMISO tree. From definition 6.4.1, each $S_T \in SS_T$ matches one candidate, but any given candidate may match more than one $S_T \in SS_T$. We then establish the convention that each tile S_T, matching a candidate C, and for which $out(S_T)$ is some arbitrary vertex $\langle k, p\rangle$, is replaced by a vertex $\langle \vartheta(C), p\rangle$.*

□

The motivation for the convention above is to make sure that definition 6.3.2 works properly for vertices corresponding to multiple occurrences of the same custom operation in a given MaxMISO. This will become clear in the next section.

6.5 A useful injection

When representing both the set of candidates (what we tile with) and the MaxMISOs (what we tile on) by expanded trees, we must make sure that all tiling possibilities in the domain of DAGs are mappable onto the domain of expanded trees and, from there, back onto the domain of DAGs. By demonstrating that there is an injective function mapping tilings between these two domains and defined for *all* DAG tilings, we can prove it to be so. For the sake of clarity, let us clearly define what these domains are:

- The domain of the injection (*concrete domain*) is that of tilings in which both the MaxMISOs and the candidates are represented by DAGs, which may themselves be directed trees;

Having clarified this, from this point on we may refer to the concrete domain simply as the "DAG domain".

- The co-domain of the injection (*abstract domain*) is that of tilings in which both MaxMISOs and candidates are represented by expanded trees, according to definition 6.3.1.

Having clarified this, from this point on we may refer to the abstract domain simply as the "domain of trees", or "directed trees". We must now prove that:

1. there is such an injection; and
2. the injective function is complete, i.e. it is defined for all tilings in the concrete domain.

The first step in this road is to develop the notion of a tile projection, subject of the next subsection.

6.5.1 Tile projections

Definition 6.5.1. Let S_D be a tile of a MaxMISO DAG D. Let also $T = \mathit{expand}(D)$ be the expanded MaxMISO tree of D. Then:

$$T \upharpoonright S_D = \{(\langle k,p\rangle, \langle l,q\rangle) \mid (\langle k,p\rangle, \langle l,q\rangle) \in E(T) \wedge (k,l) \in E(S_D)\}$$

is a *tile projection* of S_D in T.

□

The symbol "$\upharpoonright$" is called "projection". The next theorem is a key result:

Theorem 6.5.2. *Let D be a MaxMISO DAG, $T = \mathit{expand}(D)$ be the expanded MaxMISO tree, and S_D be a tile of D. Let also $\mathit{paths}_D(k)$ be the set of all paths in D from each of the successors of a vertex $k \in V(D)$ to $\mathit{out}(D)$. Then:*

$$T \upharpoonright S_D = \bigcup(\mathit{exp}_{S_D}(\mathit{out}(S_D),p) \mid p \in \mathit{paths}_D(\mathit{out}(S_D)))$$

Proof. When a branch of the traversal of $\mathit{expand}(D)$, as in definition 6.3.1, reaches $\mathit{out}(S_D)$ via a path $p \in \mathit{paths}_D(\mathit{out}(S_D))$, the remainder of the recursion branch defines:

$$\mathit{exp}_D(\mathit{out}(S_D),p) \subseteq T \tag{6.1}$$

The tile projection of definition 6.5.1 is monotonic. That is, if $T' \subseteq T$ then $T' \upharpoonright S_D \subseteq T \upharpoonright S_D$. This way, we can re-write equation 6.1 as:

$$\mathit{exp}_D(\mathit{out}(S_D),p) \upharpoonright S_D \subseteq T \upharpoonright S_D \tag{6.2}$$

We now concentrate on the left-hand side of equation 6.2. Given that $S_D \subseteq D$, we can derive from definitions 6.3.1 and 6.5.1 that:

$$\mathit{exp}_D(\mathit{out}(S_D),p) \upharpoonright S_D = \mathit{exp}_{S_D}(\mathit{out}(S_D),p) \tag{6.3}$$

Applying equation 6.3 to 6.2 we get:

$$\mathit{exp}_{S_D}(\mathit{out}(S_D),p) \subseteq T \upharpoonright S_D \tag{6.4}$$

Since equation 6.4 holds for all $p \in \mathit{paths}_D(\mathit{out}(S_D))$, we can write:

$$\bigcup(\mathit{exp}_{S_D}(\mathit{out}(S_D),p) \mid p \in \mathit{paths}_D(\mathit{out}(S_D))) \subseteq T \upharpoonright S_D \tag{6.5}$$

The sub-graph S_D is a MISO, so from definitions 6.2.1 and 6.4.1, and from the definition of an output vertex, we can write:

$$\forall k \in V(S_D) \mid k \neq \mathit{out}(S_D) : \mathit{succ}_D(k) \setminus S_D = \emptyset \tag{6.6}$$

If l is a vertex of D such that $l \notin V(S_D)$, this implies that:

$$\forall k \in V(S_D) \mid k \neq out(S_D) : k \notin pred_D(l) \tag{6.7}$$

Therefore, $out(S_D)$ is the only entry point to any sub-graph of S_D in the recursive traversal of definition 6.3.1. We can then re-write equation 6.5:

$$\bigcup(exps_{S_D}(out(S_D), p) \mid p \in paths_D(out(S_D))) = T \restriction S_D \tag{6.8}$$

This concludes the proof.
□

Corollary 6.5.3. *$T \restriction S_D$ is a sub-graph of T whose component(s)*[1] *are (is) isomorphic and match(es) $expand(S_D)$.*

Proof. We know from theorem 6.5.2 that:

$$T \restriction S_D = \bigcup(exps_{S_D}(out(S_D), p) \mid p \in paths_D(out(S_D)))$$

Each path $p \in paths_D(out(S_D))$ leads to a connected sub-graph:

$$exps_{S_D}(out(S_D), p) \subseteq T \restriction S_D$$

Because p is different in each case, the sub-graphs are not connected to each other, so they are components of $T \restriction S_D$. It follows from definitions 6.3.1 and 6.3.4 that each component is isomorphic (and therefore a match) to $expand(S_D)$.
□

Corollary 6.5.4. *Given a MaxMISO DAG D, the tree $T = expand(D)$, and a tile S_D:*

$$collapse(T \restriction S_D) = S_D$$

Proof. It follows from corollary 6.5.3 by applying lemma 6.3.5.
□

The two corollaries above have a fundamental consequence. Consider a MaxMISO DAG D for which a hypothetical tiler defined a single tile S_D, matching a candidate DAG C. Because ConCISe's tiler operates only in the domain of expanded trees, it does not have access to D or C, but only to $T = expand(D)$ and $expand(C)$. It follows from corollary 6.5.3 that T can be tiled with $expand(C)$ in such a way that, if $G \subseteq T$ is the sub-graph consisting of all tiled edges in T, then $T \restriction S_D \subseteq G$. Moreover, from corollary 6.5.4 and from lemma 6.3.3, by making $D = collapse(T)$ and $S_D = collapse(T \restriction S_D)$, we can recover the original MaxMISO DAG and its tile.

For instance, consider Figure 6.4. The tile S_D in the MaxMISO DAG D, matching a hypothetical candidate C, has a projection $T \restriction S_D$ in T. Note that $T \restriction S_D$ is a disconnected sub-graph made up of two isomorphic components, each of which is a match

[1]A *component* of a graph G is a maximal connected sub-graph of G [66]. If G is a connected graph, its only component is itself.

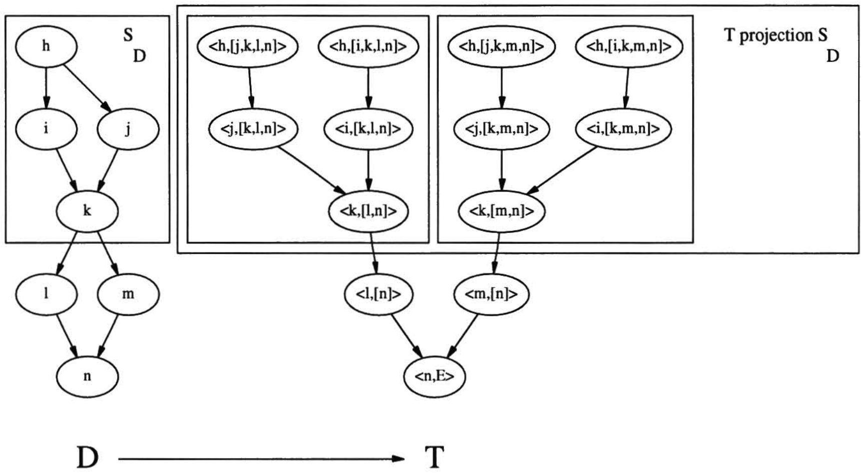


Figure 6.4: A tile in S_D and its projection in T.

to *expand*(S_D) and, therefore, to *expand*(C). Thus, we can tile T with *expand*(C) such that all vertices and edges in $T \restriction S_D$ are tiled. Then, we can replace each of the two resulting tiles by a vertex that corresponds to the custom operation represented by C. From convention 6.4.4, these vertices would be $\langle z, [l, n]\rangle$ and $\langle z, [m, n]\rangle$. The resulting tree T' would have five vertices. However, $D' =$ *collapse*(T') would have four vertices, the same result obtained by tiling D with C.

This result is one of the most significant of this chapter. It means that, by using a pair of standard transformations, *expand*() and *collapse*(), we can move the problem of tiling a MaxMISO with one tile back and forth between the domains of DAGs and directed trees, preserving completeness. The idea now is to generalize this result to tilings with an arbitrary number of tiles. This is the subject of the next section.

Before concluding this section, though, note that theorem 6.5.2 only holds because tiles are necessarily MISOs. Consider a tile S_D of a MaxMISO DAG D, matching a candidate DAG C. If the target architecture allowed for multi-output custom operations, there could be more than one entry point to S_D in the recursive traversal of *expand*(D). In this case, at least one component of $T \restriction S_D$ would necessarily not match *expand*(C) and corollary 6.5.3 would not hold. This is illustrated in Figure 6.5. Note that two *different* candidates would be necessary to tile $T \restriction S_D$. Let T' be the optimized version of T in which each tile is replaced by a vertex. From convention 6.4.4, these vertices would be $\langle x, [n]\rangle$ and $\langle y, [m, n]\rangle$. Therefore, $D' =$ *collapse*(T') would have four vertices (two custom operations and two native instructions), while the tiling of D with C leads to only three vertices.

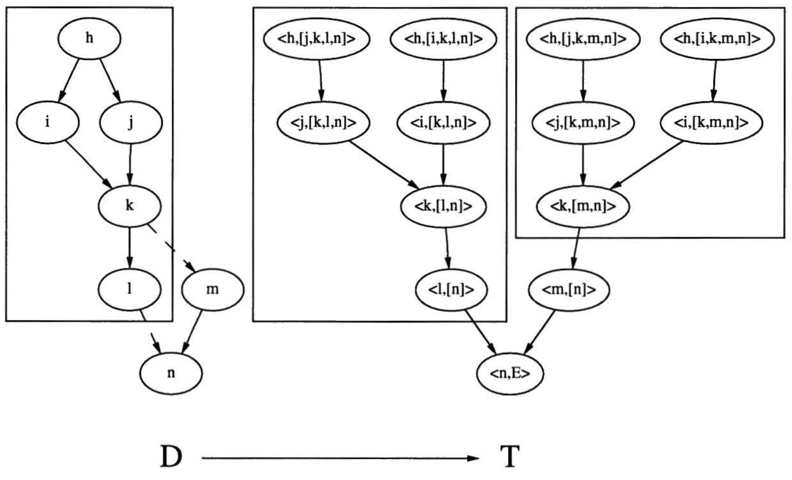


Figure 6.5: Covering D with a non-MISO sub-graph, and the corresponding situation in T.

6.5.2 Completeness

Let us now extend the notion of a tile projection to that of a tiling projection:

Definition 6.5.5. Consider a duet (D, SS_D), in which D is a MaxMISO DAG. Let also $T = expand(D)$ be its expanded tree. Then:

$$T \restriction SS_D = \{T \restriction S_D \mid S_D \in SS_D\}$$

is the *tiling projection* of SS_D in T.

□

It is easy to see that, because the tiles in SS_D do not overlap, the tiles in $T \restriction SS_D$ will also not overlap.

The previous results give us a mapping between the domains of DAGs and directed trees:

Definition 6.5.6. Let D be a MaxMISO DAG and T be the corresponding MaxMISO tree. We define a mapping from duets to duets:

$$EXP((D, SS_D)) = (T, SS_T)$$
$$COL((T, SS_T)) = (D', SS'_D)$$

between DAGs and trees, with $EXP((D, SS_D)) = (T, SS_T)$ given by:

1. $T = expand(D)$; and
2. $SS_T = T \restriction SS_D$.

and with $COL((T, SS_T)) = (D', SS'_D)$ given by:

1. $D' = collapse(T)$;
2. $SS'_D = \{collapse(S_T) \mid S_T \in SS_T\}$.

□

We can then prove the following theorems:

Theorem 6.5.7. *Definition 6.5.6 defines an injection between DAG and tree duets, such that,* $\forall (D, SS_D) : COL(EXP(D, SS_D)) = (D, SS_D)$.

Proof. From lemma 6.3.3:

$$D' = collapse(T) = collapse(expand(D)) = D$$

Now, because the tiles do not overlap, we can apply theorem 6.5.2, followed by corollary 6.5.4, to every tile projection, resulting in:

$$\forall S_T \in SS_T : collapse(S_T) = collapse(T \upharpoonright S_D) = S_D$$

It follows that $EXP()$ is an *abstraction function*, and $COL()$ a *concretization function* between the domains of DAGs and trees. This concludes the proof.

□

The result above proves that the mapping of definition 6.5.6 is complete. In other words, that every partial covering possibility in the domain of DAGs has a projection in the domain of trees that can be brought back to the domain of DAGs by standard transformations, and without loss of information.

The next result is the last we need to prove the correctness of our tree-based partitioning approach in ConCISe:

Theorem 6.5.8. *In the mapping of definition 6.5.6, for each* $S_D \in SS_D$, *if* C *is a candidate DAG that matches* S_D, *then all edges of* $T \upharpoonright S_D$ *can be tiled with* $expand(C)$.

Proof. Again, because the tiles do not overlap, we can apply theorem 6.5.2, followed by corollary 6.5.3, to every tile projection. This suffices to prove the theorem.

□

Let us denote the set of candidates by SC. Let us also define $expand(SC) = \{expand(C) \mid C \in SC\}$, the set of expanded candidates. Figure 6.6 then summarizes the results above, and the way in which they are used in ConCISe. The left-hand side of the figure illustrates the tiling process in the concrete domain of DAGs. The candidates in SC are used to produce the tiling SS_D. Theorem 6.5.7 proves the correspondence between the DAG duet and the tree duet, shown on the left-hand side of the figure. On the right-hand side, instead of projecting SS_D in T, the tiling $T \upharpoonright SS_D$ is generated by tiling T directly, with the expanded candidates in $expand(SC)$. The equivalence between the left and the right-hand sides is proven by theorem 6.5.8. Since ConCISe's tiler does precisely what is illustrated on the right-hand side, its approach is proven sound.

Theorems 6.5.7 and 6.5.8, as illustrated in Figure 6.6, are the main results of this chapter.

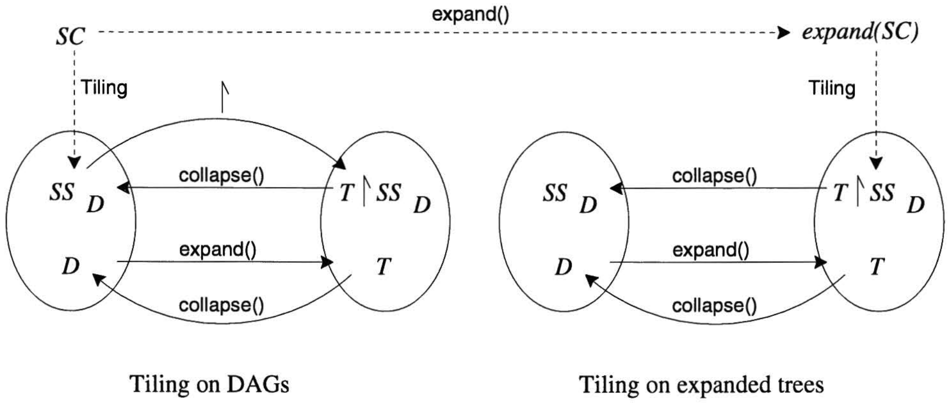


Figure 6.6: Tiling on DAGs and expanded trees.

6.5.3 The advantage of tiling in the domain of trees

Summarizing so far, ConCISe's detection algorithms already unravel MaxMISO DAGs into expanded trees during detection. Because candidates are subsequently derived from these MaxMISOs as tree sub-graphs, the detected set of candidates will automatically be *expand*(SC), instead of SC. In the previous section, we have proven that the problem of tiling the expanded MaxMISO trees with elements of *expand*(SC) incorporates every solution of the problem of tiling the original MaxMISO DAGs with elements of SC. The question one is faced with now is: Can there be an optimal tiling solution in the domain of trees that does not exist in the domain of DAGs? That is, is the mapping of tilings from the concrete to the abstract domain injective but not surjective? If the answer is positive, then ConCISe's approach of expanding DAGs into trees may potentially lead to reduced tiling costs.

The burden of proof for this question is much less severe than that of proving completeness. Because it is an existence question, one only needs an example to prove it. Figure 6.7 is one such an example. A MaxMISO DAG D is shown alongside its expanded MaxMISO tree T. The computation property of each vertex is shown. Assume the sets SC and *expand*(SC) have three candidates. In the figure, three tiles are shown, one in D and two in T. Each tile matches a different element of *expand*(SC). Note that all three candidates are originally trees, so each element of SC matches one element of *expand*(SC).

Tiling D with the three candidates gives only one tile, the one shown on the left-hand side of Figure 6.7. The other two candidates only match sub-graphs of D that have two outputs and, therefore, are not tiles (check it out). The resulting graph has a tiling cost of 11 instructions.

Now, tiling T with the same three candidates gives two tiles, as shown on the right-hand side of Figure 6.7. Because the multi-dep vertex in D is replaced by two non-multi-dep vertices in T, the two largest candidates now match sub-graphs of T that are

MISOs, producing the shown tiles. The tiling cost is now 9 instructions. However, the two untiled vertices with computation property `sll ...,4` will be collapsed into a single vertex after the tiling, when bringing the tiled graph back to the domain of DAGs. As a result, the final graph will have 8 instructions, instead of the 11 resulting from the DAG tiling.

The elimination of multi-dep vertices in the domain of trees opens further tiling possibilities than those available in the domain of DAGs. The extra freedom comes from the fact that a single instruction can be represented by multiple vertices in the expanded tree, which can each be tiled independently. In Figure 6.7, note that the instruction `lui ...,32` is represented by two different vertices in T. One of them is in the tile S_T, while the other is untiled. Therefore, the computation `lui ...,32` is redundantly executed twice: once in the custom RFU hardware, and once as a native instruction. A similar story holds for other three instructions represented each by two vertices, one in S_T and one in S'_T. They will be redundantly computed in the hardware corresponding to two different custom operations (albeit possibly being cross-minimized, as we will see in Chapter 8). However, note that a tiling with minimal cost is all that matters for performance, even if redundant computations are the price for it.

At this stage, we finally have material enough to strongly substantiate one more of the propositions of this thesis:

Proposition 6.5.9. *A DAG covering problem for single-output instruction selection, as the one faced during partitioning in the ConCISe tool-set, can be translated into a directed tree covering problem without loss of generality or completeness. The translation not only facilitates the implementation of the graph traversal and matching algorithms, but can also lead to covers whose cost is lower than the minimal cost achievable in the concrete domain of DAGs.*

6.6 Computing the tiling cost

To help the reader keep things in perspective, Figure 6.8 illustrates the partitioning process at a glance, as represented by the formalisms introduced in this chapter. A single-input MaxMISO DAG D of a basic block P (Figure 6.8(A)) is detected already in its expanded tree form (Figure 6.8(B)), and tiled. Each tile is then replaced by a vertex, according to convention 6.4.4 (Figure 6.8(C)), and the optimized tree is collapsed according to definition 6.3.2 (Figure 6.8(D)). The resulting DAG is inserted back into the application's basic block according to the original data dependencies, as shown by the dashed edge in Figure 6.8(D).

Now that we have proven the advantages of tiling in the domain of trees, we need to formalize the tiling algorithm itself. The previous section already suggested one of the complications involved: the tiling cost of the tree T in Figure 6.7 (nine instructions) is not equal to the actual number of instructions after collapsing (eight).

Consider Figure 6.9, for instance. The tree tiling illustrated in Figure 6.9(B), from the point of view of the tree alone, is better than that of Figure 6.8(B), since less tiles

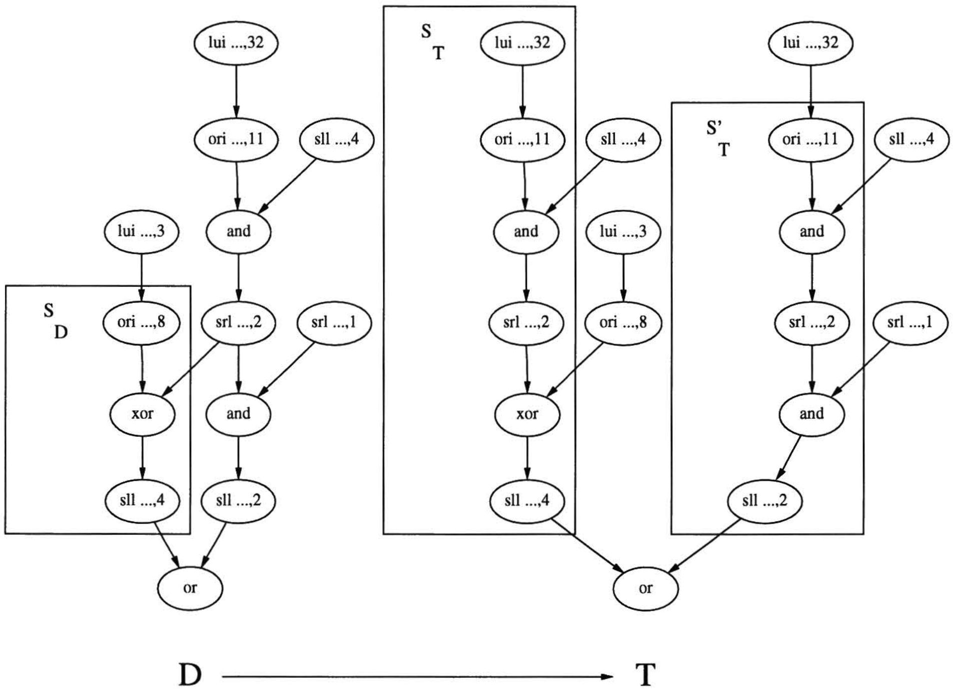


Figure 6.7: Covering D and its corresponding tree T with the same set of 3 candidates.

are used to cover the same vertices. However, the true tiling cost can only be known after collapsing the tiled graph. This way, comparing Figure 6.8(D) and Figure 6.9(D), we see that the optimal tiling is, in fact, that of the former.

Generally speaking, the optimal tiling of the tree is not necessarily the optimal tiling of the resulting DAG. The effect of *collapse*() on the tiling cost breaks the *optimal substructure* [68] of the tiling problem, i.e. the optimal solution to the problem is no longer guaranteed to contain optimal solutions to sub-problems. For instance, Figure 6.9(B) shows an optimal solution to the sub-problem of tiling a tree sub-graph with root at $\langle m, [n] \rangle$. However, this is clearly not part of the overall optimal solution illustrated in Figure 6.8(B), which can only be achieved by taking into account the effect of *collapse*() in the tree as a whole. This way, as mentioned in the previous chapter, one can no longer apply dynamic programming to the tiling process.

Let us define the *collapsed cost* as the number of vertices of a collapsed MaxMISO, after tiling. The MaxMISO in Figure 6.8(D) has collapsed cost four, while the one in Figure 6.9(D) has collapsed cost five. The most straight-forward method to find a tree tiling whose collapsed cost is minimal is an exhaustive search. All possible tree tilings are computed and, after collapsing each of them, the collapsed cost is evaluated. The tiling with minimal collapsed cost is then chosen. It is easy to see that this approach has exponential complexity with the size of the MaxMISOs, and may result in large computing times.

In ConCISe, we used a branch-and-bound algorithm to reduce the computing time of the tiling process. If the partial collapsed cost of a tiling currently being computed already surpasses the total collapsed cost of some other tiling previously evaluated, the tiler stops and tries a different tiling. Naturally, in order to evaluate the partial collapsed cost of a tiling while the tiling itself is still being computed, the algorithm needs to partially evaluate the effect of *collapse*() *during* the computation of a new tiling.

Before describing how this is done in ConCISe, we need a couple of definitions. If the *distance* between two vertices in a directed tree T is the number of vertices in the directed path connecting them, a *breadth-first traversal* [68] of T means that, starting from a vertex $\langle k, p \rangle \in V(T)$, all vertices at a distance d from $\langle k, p \rangle$ are visited before any vertex at a distance $d + 1$. A *backward, breadth-first traversal* of T means that, from a vertex $\langle k, p \rangle$, only vertices $\langle l, q \rangle$ such that there is a directed path from $\langle l, q \rangle$ to $\langle k, p \rangle$ are visited.

In the ConCISe tool-set, for each vertex $\langle k, p \rangle$ visited during the tiling process, starting from $out(T)$, all possible tiles S_T, such that $out(S_T) = \langle k, p \rangle$, are computed by means of tree matching techniques. Once for each S_T, and once with $\langle k, p \rangle$ untiled, a backward, breadth-first traversal from $\langle k, p \rangle$ to other vertices $\langle l, q \rangle$, such that $\langle l, q \rangle \notin V(S_T)$, is executed. Again, all tiles S'_T such that $out(S'_T) = \langle l, q \rangle$ are computed for each $\langle l, q \rangle$ visited, and the corresponding traversals continue recursively.

Each branch of the recursion stops only when an input vertex of T is visited or included in a tile, at which point a possible tiling SS_T will have been computed. Each recursion branch will correspond to a different tiling, and when all branches stop, the tilings of T will have been computed.

Within a given recursion branch, the partial tiling cost is evaluated as the number of

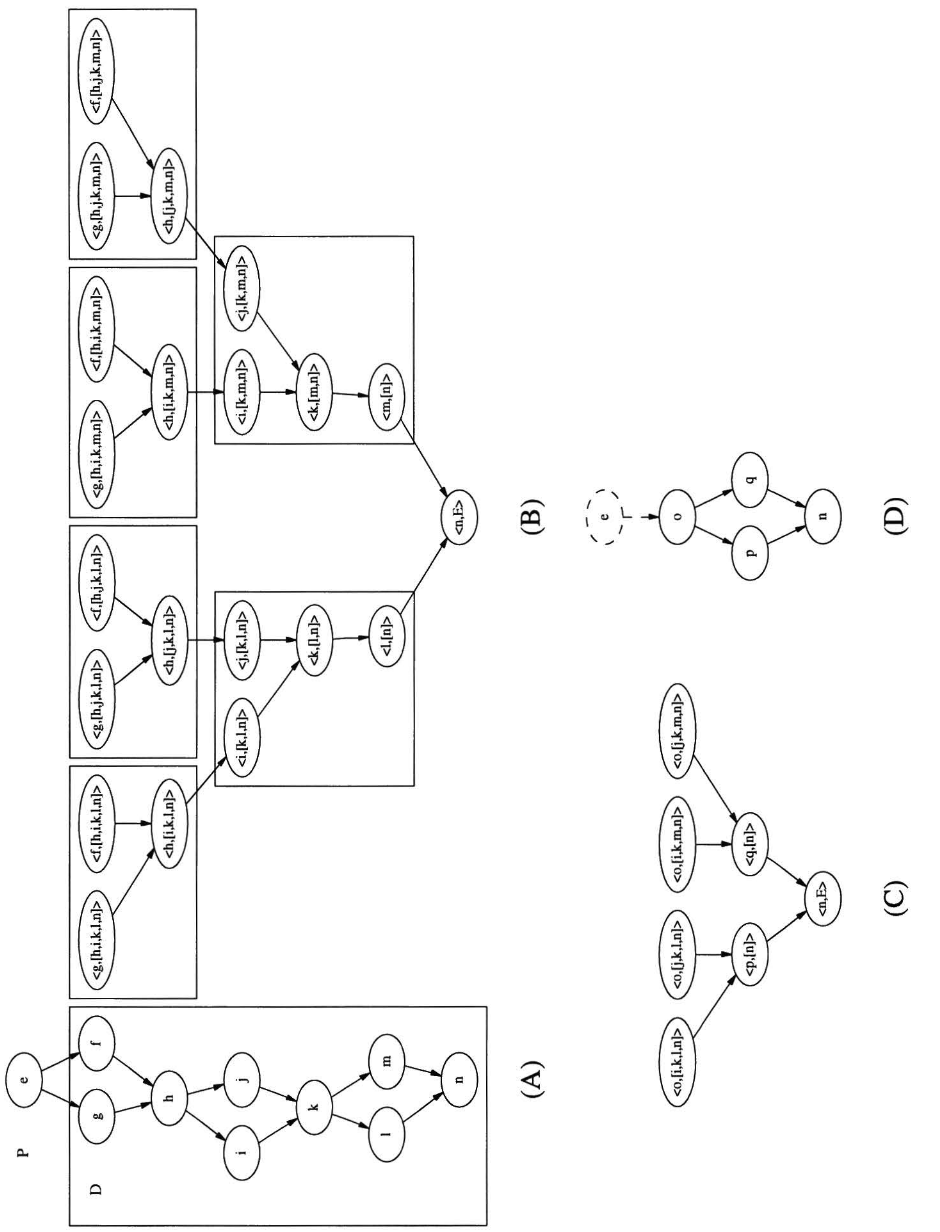


Figure 6.8: The partitioning process at a glance. (A) MaxMISO DAG D in a basic block P. (B) the MaxMISO tree *expand*(D) and its tiling. (C) Replacing tiles by vertices, according to convention 6.4.4. (D) The resulting, optimized data-flow after collapsing.

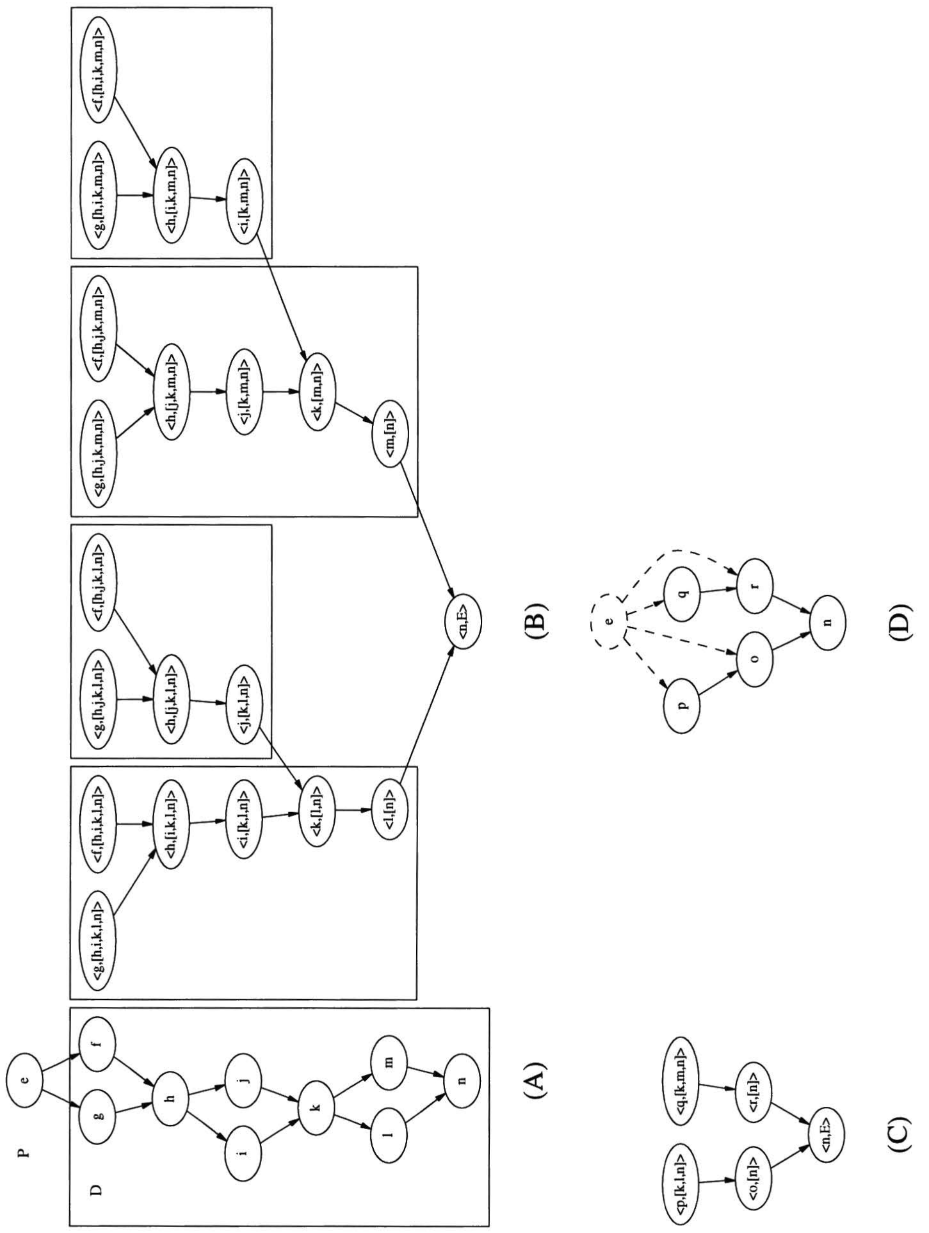


Figure 6.9: Optimal tree cover, as opposed to the optimal tiling of Figure 6.8. (A) MaxMISO DAG D in a basic block P. (B) the MaxMISO tree *expand*(D) and its tiling. (C) Replacing tiles by vertices, according to convention 6.4.4. (D) The resulting, optimized data-flow after collapsing.

tiles already computed in the branch, plus the number of untiled vertices. To take into account the effect of *collapse*(), the algorithm does the following. When a vertex $\langle k, p\rangle$ is visited (and, therefore, is still untiled), the algorithm checks whether other vertices $\langle k, q\rangle$ are also untiled in this recursion branch. In that case, it is easy to see that the tiling with minimal collapsed cost will be that in which the tilings of all sub-trees of T with root at an untiled vertex $\langle k, q\rangle$ are identical to the tiling of the sub-tree of T with root at $\langle k, p\rangle$. This way, the tiling continues to be computed only for the latter sub-graph, and the partial tiling cost is evaluated as if the other sub-graphs did not exist (they will be collapsed anyway). This situation is illustrated in Figure 6.4 for the vertices $\langle k, [l, n]\rangle$ and $\langle k, [m, n]\rangle$. As a result, the evaluation of the partial tiling cost can correctly incorporate the effect of *collapse*() while the tiling is still being computed. That cost is, therefore, the partial collapsed cost needed for the branch-and-bound algorithm.

The procedure described in the previous paragraph already prunes part of the tiling space known to be sub-optimal. In addition, when the partial collapsed cost evaluated during the computation of a tiling exceeds the total collapsed cost of another tiling previously computed, the current recursion branch is abandoned before completion, and the tiler bounds to another branch. Together, this procedure and the one in the previous paragraph make up the branch-and-bound criteria used.

6.7 Final considerations

In this chapter, we have defined an injection between the concrete domain of DAGs, in which ConCISe's partitioning problem is defined, and an abstract domain of directed trees, in which that problem is solved by the tool-set. We have also given formal descriptions of the techniques used by the tool-set to perform data-flow optimizations by means of graph covering techniques. Although the complexity of the covering algorithm utilized is exponential, their typical computing times are still within very reasonable limits, as we will see in the benchmark results of the next chapter. The branch-and-bound techniques described help make it so.

7 Benchmarking ConCISe

"*We must look the facts in the face.*"
The Master, in "The Master and Margarita", by Mikhail Bulgakov.

In this chapter, we evaluate the performance and cost-effectiveness of ConCISe. Because we still have no silicon implementation, the tests make use of two ConCISe simulators, described in the next section.

7.1 Simulation

We have developed two simulators to evaluate ConCISe: an instruction set simulator and a nearly-cycle-accurate system simulator. They are positioned in the overall flow as illustrated in Figure 5.2, on page 55.

The first one, henceforth called the *functional simulator*, is based on the SPIM model [43]. It was extended so to recognize and execute PSI instructions. When simulating a ConCISe-optimized program, it reads the equations file generated by the hardware synthesis module. This file contains the synthesized and optimized logic equations corresponding to the RFU hardware (i.e. the PTs and STs). Each time a PSI instance is encountered, the simulator switches to a set of routines that evaluate the equations for the given set of input data at a logic-gate-level. The functional simulator is used as a profiler for MIPS native code (see Figure 5.2) and to generate performance results for ConCISe with respect to instruction execution counts (i.e. ignoring dynamic behavior like cache misses and pipeline stalls). It has also been used to verify the functional correctness of ConCISe's optimizations.

The second simulator, henceforth called the *system simulator*, can simulate the memory hierarchy and pipeline behavior, as well as other system components (buses, I/O interfaces, external memory, etc.). It has been used to gather data on the dynamic effects of ConCISe optimizations, namely their impact on cache performance. Just as the functional simulator, it also simulates PSIs based on the equations file produced by the synthesis module.

7.2 Benchmark results

We used five well-known encryption algorithms to benchmark our approach: Eric Young's DES implementation [57]; Bruce Schneier's A5 implementation [56]; a Philips in-house benchmark derived from the RSA Data Security Inc. MD5 message-digest algorithm; and

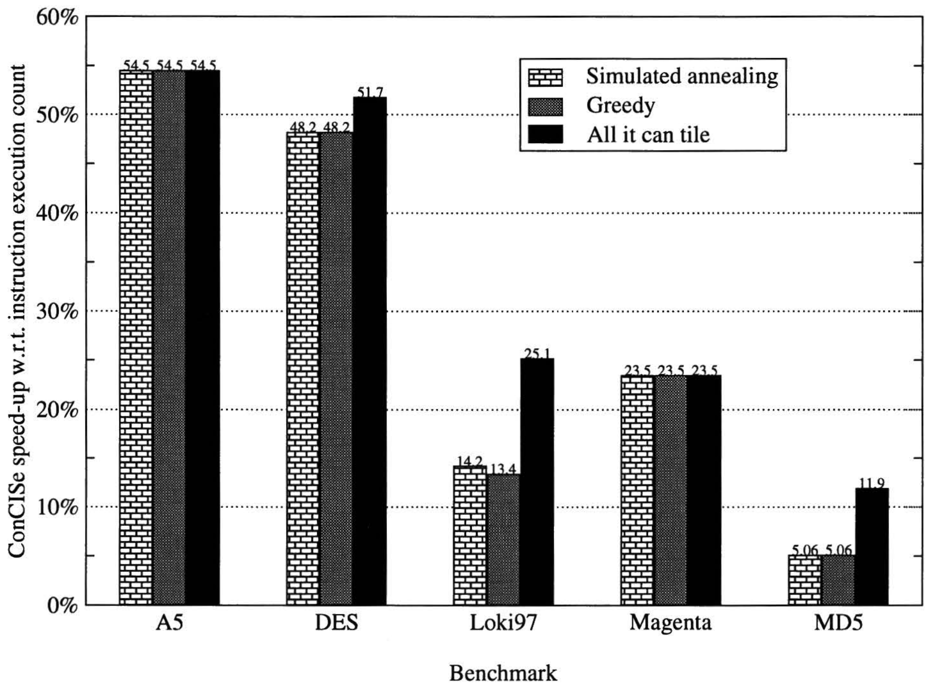


Figure 7.1: ConCISe speed-up for different encryption algorithms.

Brian Gladman's implementations of Loki97 and Magenta [58]. All five codes were taken off-the-shelf, without changes. In every case, the assembly was produced by an optimizing compiler. Note that a skilled programmer could re-write the codes to take advantage of ConCISe's RFU. In addition, encryption algorithm developers could use bit-level manipulations more freely in their algorithms, with fewer concerns about performance on a programmable microprocessor.

7.2.1 ConCISe's performance

The results are shown in Figure 7.1. The speed-ups refer to the performance of a MIPS R3000 augmented with a ConCISe RFU, when compared to the same MIPS R3000 without the RFU. In every case, we assume that the clock frequency is such that the RFU can execute in a single clock cycle (comfortably up to 100MHz). A speed-up of 100% means that the Instruction eXecution Count (IXC) is halved. Programmer intervention

Benchmark	No. MaxMISOs	Size of the largest MaxMISO	No. candidates
A5	48	10 instructions	117
DES	140	16 instructions	243
Loki97	227	14 instructions	92
Magenta	155	14 instructions	42
MD5	223	6 instructions	55

Table 7.1: Number of MaxMISOs, candidates, and size of the largest MaxMISO.

is never used to generate code for the RFU version, the tool-set being run from a script. Results obtained with both selection engines are shown, and are compared to the results we would have obtained if the tool-set could synthesize in the FPL all data-flow segments it found proper, that is, if the hardware partition always fitted (see Figure 5.2). We can see that the annealing selection engine obtains only slightly better results than the greedy mechanism for Loki97. Although one may argue the difference is not worth the longer computing times implied (see Figure 7.6, page 101), from this point on, unless explicitly mentioned otherwise, all results reported will have been obtained with the simulated annealing selection engine.

Figure 1.11, on page 13, illustrates one of the custom operations actually synthesized in our tests, which partly computes one of A5's Linear Feedback Shift Registers (LFSR). Figure 1.10, on page 13, illustrates an actually detected but not selected DES candidate (not in the critical path). Figure 5.5, on page 60, illustrates a detected MaxMISO of Magenta that was also a selected and synthesized candidate. It is made up of 12 native instructions and occurs in six different places in the input assembly. Magenta also has the largest synthesized candidate of all benchmarks, a native assembly segment with 14 instructions and four occurrences in the input code.

Table 7.1 illustrates the number of detected MaxMISOs, the number of (possibly partially overlapping) candidates, and the size of the largest MaxMISO for each of the five benchmarks. In spite of the size of the largest ones, many of the MaxMISOs actually have only a single assembly instruction in it. To give an idea of the complexity of the hardware partition, Table 7.2 illustrates the number of PTs required by each of the benchmarks (see Figure 4.4, page 49). Note that the PAL PTs are consistently utilized in their entirety.

Figure 7.2 shows ConCISe's RFU speed-up when a progressively higher number of custom operations is encoded in the RFU. The point at which the curve levels off represents the limit of RFU hardware synthesis of our tool-set. The filled marking on each curve shows the number of custom operations we can actually encode before the RFU circuit no longer fits in the available CPLD resources. For A5 and Magenta, the amount of FPL in the RFU is enough to allow for the highest speed-ups. For DES, it is close to the highest. For MD5 and Loki97, however, roughly twice the amount of FPL would be necessary to reach the point where the curves level off.

Figure 7.3 illustrates the IXC distribution for different kinds of instructions. Note that many synthesizable instructions may only occur in single-instruction MaxMISOs, in

Benchmark	PAL PTs	PLA PTs
A5	128	82
DES	128	112
Loki97	128	87
Magenta	128	20
MD5	128	112

Table 7.2: Complexity of the hardware partition in terms of number of Product-Terms required. The RFU has a total of 128 configurable PAL PTs, and 128 configurable PLA PTs, equally distributed over four logic blocks.

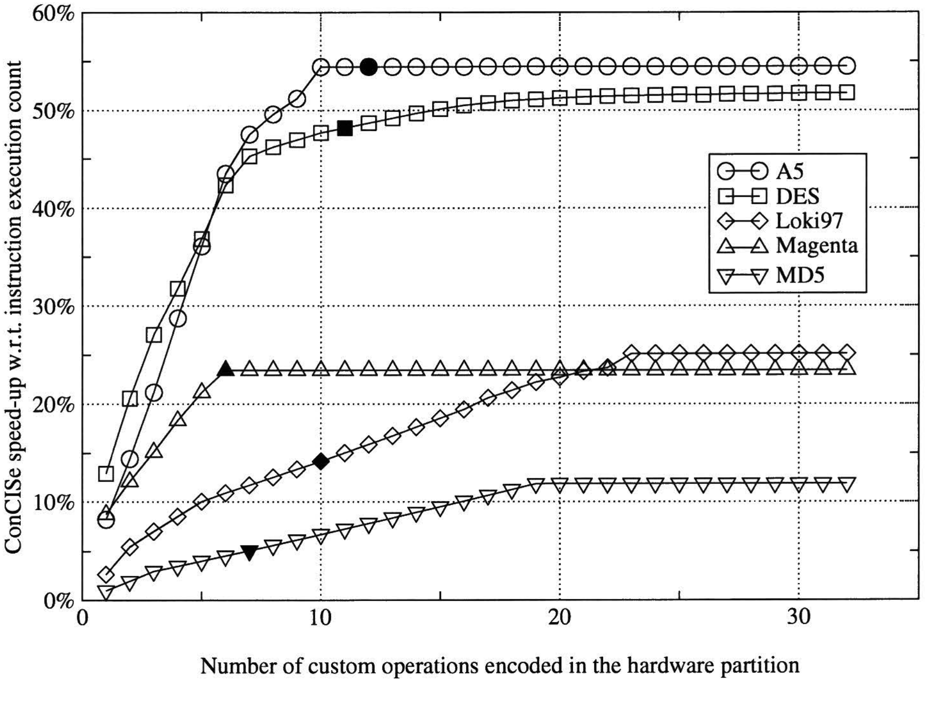


Figure 7.2: ConCISe speed-up for different numbers of custom operations. The filled marking on each curve shows how many custom operations did actually fit in the RFU hardware. In the case of Magenta, more candidates would have fitted in the RFU, but were not selected by the tool-set because no extra speed-up would have been achieved.

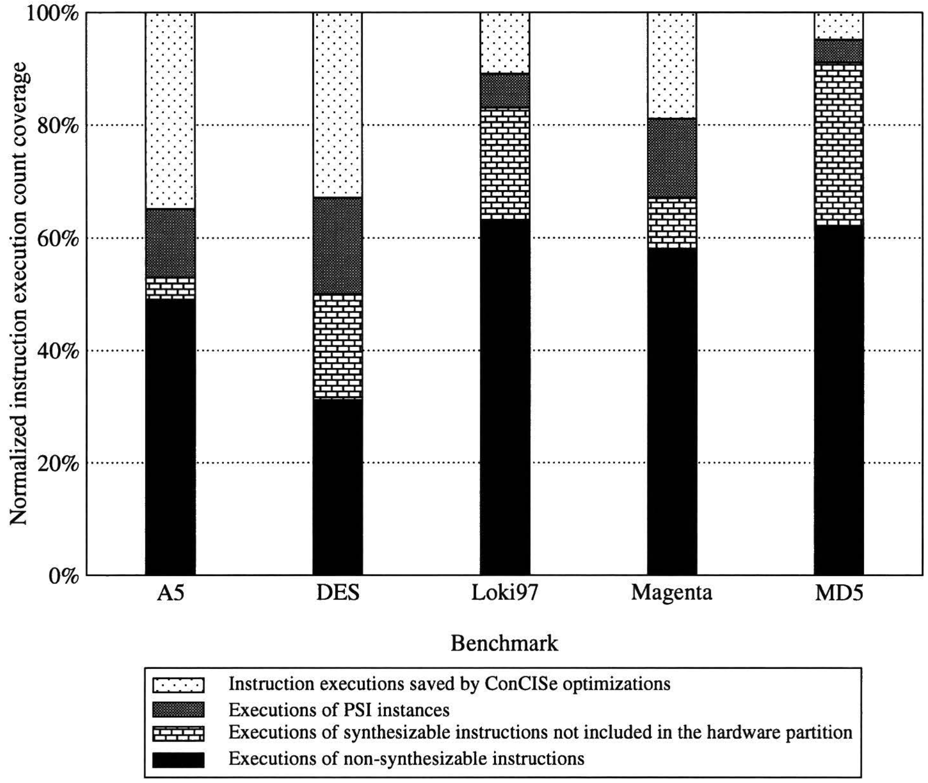


Figure 7.3: Normalized instruction execution count coverage, showing the savings allowed by ConCISe data-flow optimizations.

which case a PSI substitution would allow for no speed-up. Now, assume the computation of one custom operation in the RFU dissipates approximately as much power as an ALU operation. Then, Figure 7.3 would give a rough idea of the power savings possible by applying ConCISe's optimizations, to the extent that power dissipation is proportional to the IXC. However, although the CPLD utilized is a pure CMOS device with very low-power characteristics [55], we did not gather conclusive data to confirm (or disprove) that assumption.

In order to investigate the run-time effects of ConCISe, we used the system simulator. A system made up of a ConCISe-MIPS R3000 with 16K of data cache, a Philips PI bus, embedded flash-ROM, and DRAM was simulated. Figure 7.4 illustrates the RFU speed-up for different sizes of instruction cache. The peak seen at 512B of cache is due to the fact that the A5 assembly code with PSI instances already fits in a 512B instruction cache, while the original A5 assembly does not. This can be seen in Figure 7.5. For an instruction cache of 1KB or higher, both codes fit in the cache, and the speed-up converges to the IXC speed-up (slightly different from the value in Figure 7.1 due to differences in the assembler options used). The results illustrate the (side-effect) reduction of code size that results from the use of PSIs.

Overall, the results are very encouraging, given the relatively small amount of extra hardware the addition of the RFU implies, and given the fact that we used off-the-shelf code in our benchmarks. It is reasonable to assume that the adoption of RFU-aware coding styles could greatly improve performance [63], a scenario most likely in any industrial application of ConCISe. In addition, it is reasonable to assume that further speed-up opportunities have been missed because useful information about the application is lost after code generation. For instance, ConCISe optimizations could potentially reduce register pressure by eliminating some temporary values. This way, less spill code would need to be introduced, with a consequent further increase in performance. However, this is no longer possible at assembly level. Similarly, because the tool-set can only detect data dependencies via register accesses, spill code may break longer sequences of synthesizable instructions (i.e. larger candidates), which then become invisible at assembly level. All this is the price we pay for keeping ConCISe detached from the front-end compiler.

7.2.2 The tool-set's performance

The annealing selection mechanism calls the tiler once at every iteration, while the greedy mechanism does it just as many times as the number of candidates it is to select. Therefore, it doesn't come as a surprise that annealing takes roughly one order of magnitude more time than the greedy mechanism to execute, as illustrated in Figure 7.6. The times were measured in a Sun UltraSPARC 10, running at 336 MHz. Still, even the annealing executes always below a minute (and usually below 10 seconds), which is a very reasonable time specially when taking into account that the tiling algorithm has exponential complexity with the size of the MaxMISOs.

Actually, most of the time taken by the tool-set is spent on hardware synthesis, after the partitioning is complete. Table 7.3 illustrates the results, measured on the same

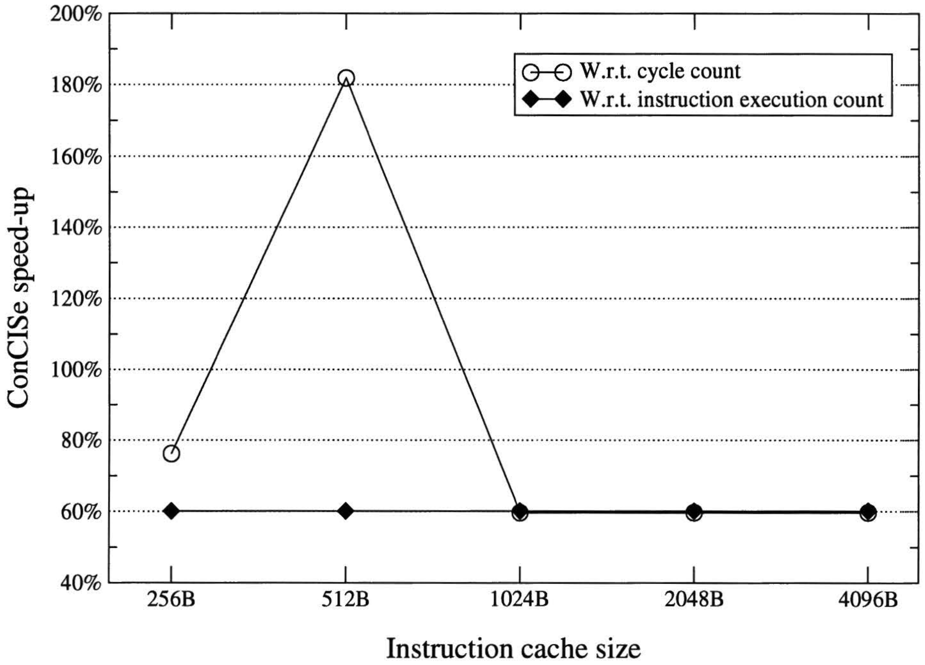


Figure 7.4: Instruction cache size versus speed-up for the A5 algorithm.

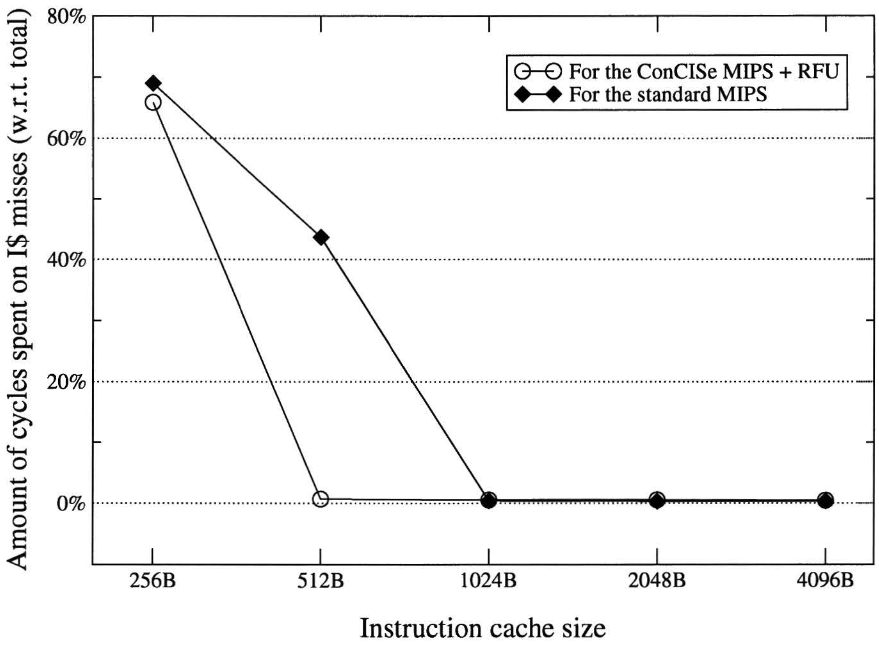


Figure 7.5: Relative penalty of instruction cache misses for the A5 algorithm.

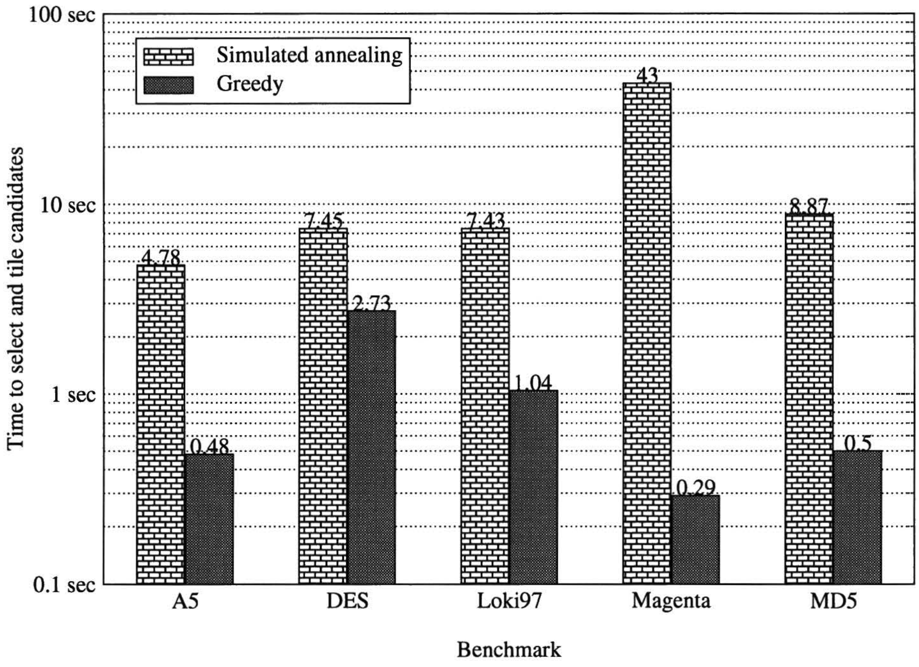


Figure 7.6: Time taken to select and tile candidates.

Benchmark	Time taken to synthesize the hardware partition
A5	≈ 27 seconds
DES	≈ 29 seconds
Loki97	≈ 18 seconds
Magenta	≈ 17 seconds
MD5	≈ 29 seconds

Table 7.3: Time taken to synthesize the hardware partition.

Sun UltraSPARC 10. The synthesis time depends not only on the number of custom operations in the hardware partition, but also on the nature of these operations. For this reason, the correlation between Table 7.3 and Figure 7.2 is only slight.

The time spent on detection is negligible when compared to the time taken by selection and synthesis together.

To completely process an input assembly in a fully automatic way, the tool-set needs to loop over the partitioning and synthesis steps until the synthesized hardware partition fits in the RFU. Assuming an average iteration time of 1 minute, an average of 10 operations fitting in the hardware partition, and given that the tool-set tries the maximum of 16 custom operations in the first iteration, we can estimate the tool-set's average computing time in approximately (16 - 10) x 1 = 6 minutes. In addition to this, the tool-set must also profile the input program before the first iteration (see Figure 5.2, page 55). This, however, is a function of the input program and data, not pertinent to the tool-set's performance itself.

We do not know at what point, in terms of the number and size of MaxMISOs, the exponential complexity of the tiling algorithm will lead to unacceptably high computing times for the tool-set. The average computing time of 6 minutes evaluated from our results, however, does indicate that this point has not been reached in the benchmarks we studied. Still, if more complex applications were to be processed by the tool-set, the user can always choose the greedy selection mechanism. As shown in Figure 7.6, the greedy selector requires an order of magnitude less computing time than the annealing mechanism. The penalty in performance is minimal, as illustrated in Figure 7.1.

7.3 Discussion

Before we proceed to the next chapter, a few relevant considerations can be made based on the results presented in the previous section.

7.3.1 Issues related to the front-end compiler

With the current tool-set, the front-end compiler utilized to produce optimized assembly code is unaware of the existence of ConCISe's RFU. This potentially wastes some optimization opportunities, as exemplified with the following segment of MIPS assembly, extracted from Loki97:

```
# 143 for (j = 4, k = 7; j < 8; j++, k += 8)
li $2, 7
.loc 2 144
# 144 pval |= (long)((i >> j) & 0x1) << k;
sra $3, $16, 4
and $3, $3, 1
sll $3, $3, 7
.loc 2 143
# 143 for (j = 4, k = 7; j < 8; j++, k += 8)
li $2, 15
sra $8, $16, 5
and $9, $8, 1
sll $10, $9, $2
or $3, $3, $10
addu $2, $2, 8
sra $11, $16, 6
and $12, $11, 1
sll $13, $12, $2
or $3, $3, $13
addu $2, $2, 8
sra $14, $16, 7
and $15, $14, 1
sll $24, $15, $2
or $3, $3, $24
.loc 2 145
# 144 pval |= (long)((i >> j) & 0x1) << k;
# 145 P[i].l = pval;
sw $3, 0($5)
```

All synthesizable instructions in this sequence could potentially be replaced by a single RFU instruction with one input (register `$16`) and one output (register `$3`). However, our tools did not consider the entire sequence a valid candidate, given the presence of additions and variable length shifts (emphasized in *italic*)[1]. These instructions are present because, when the compiler unrolls the `for` loop, the increments of 8 for the counter `k` are translated into a sequence of `addu $2, $2, 8` instructions with variable length shifts dependent on register `$2`.

If the front-end compiler was aware of the existence of the RFU, as well as its capabilities and limitations, it could have replaced the additions and variable length shifts

[1]The segment shown is constituted of so-called pseudo-instructions, a higher-level assembly representation that is later translated into real assembly instructions by the assembler. This explains why the mnemonic `sll` is used in variable length shifts. It is in fact a pseudo operation that will later be translated into a real `sllv` instruction. The ConCISe tool-set looks for synthesizable instructions after translation from pseudo to real code. Therefore, at the level the tool-set operates, the opcode `sll` will always correspond to a fixed length shift, and will always be a synthesizable instruction.

by fixed length shifts whose immediate fields were pre-computed at compile-time. This would render the resulting assembly sequence synthesizable in a custom operation.

Generally speaking, the front-end compiler could facilitate the detection of candidates by using code transformations that increased the size of basic blocks, or trying to generate longer sequences of synthesizable instructions by properly tuning the work of the code generator, instruction scheduler, and the register allocator.

7.3.2 Is Compile-Time Reconfiguration enough?

Proposition 4.2.1, on page 47, is to some extent based on the assumption that there are enough FPL resources to map all custom operations that can result in useful speed-ups, concurrently. Otherwise, it would be impossible to argue RTR away, since it would still have greater optimization potential (see Section 1.3.5, page 18). This assumption holds clearly for three of the benchmarks studied, as shown in Figure 7.2. For Loki97 and MD5, however, roughly twice the achieved speed-up could have been obtained if more custom operations could have been synthesized. Still, this does not invalidate at all the approach adopted in ConCISe.

Given our benchmark results, the designer is faced with a trade-off. He can either:

1. choose to stick to the CTR approach of proposition 4.2.1, due to its simplicity, reliability, and testability, and give up the extra speed-up he could potentially achieve for certain application programs. This is the option we currently adopt with ConCISe, supported by the encouraging results obtained; or

2. choose to resort to a limited RTR approach to exploit all speed-up opportunities.

The first option is more conservative and, as such, can potentially be taken up by the industry faster. Even in the second case, however, the idea of encoding several custom operations in a single FPL configuration is still cost-effective. In fact, Figure 7.2 suggests that just one more configuration would suffice to achieve nearly all possible speed-up for MD5 and Loki97. This way, the CPLD core used in the RFU could be provided with an extra configuration plane. In Figure 1.2, page 4, this would mean that each configuration point would have two, instead of one, SRAM cells, alternatively selectable from an external signal under the control of the application program. Prior to execution, both configuration planes would be loaded and, at run-time, the FPL configuration could be swapped between both planes in a period as short as a single clock cycle (see the DPGA [14], for instance). At the tool-set level, this would imply in practically no changes, apart from dividing the hardware partition in two. The simplicity of the approach would be, to a great extent, preserved. The run-time management of the two configuration planes could be accomplished simply by statically inserting special instruction opcodes in the assembly that triggered a configuration swap at the proper moments. This cannot be compared, in complexity or cost, to partial RTR approaches involving many hardware modules, like that of DISC. In addition, the reconfiguration overhead would be minor, because both configurations are stored locally in the FPL. The price, however, is in the extra silicon real-state necessary to double the amount of configuration memory. This investment must be traded-off against other opportunities to increase the overall system performance.

At this stage, the reader might argue that, instead of using extra silicon for an extra configuration plane, a better alternative could simply be to double the size of the RFU, such that all necessary candidates could again be encoded in a single configuration. This would, of course, be more in line with the basic design principle of ConCISe. Note, however, that doubling the configuration memory implies less extra silicon than doubling the entire CPLD structure, including switches, PTs, and the configuration memory itself. While some prior works may have overlooked cost-effectiveness considerations in order to focus on the scientifically challenging RTR approach, one should not make the opposite mistake. Moreover, doubling the number of PTs and switches will have implications on the timing of the CPLD. It will increase the associated delays, potentially rendering the RFU a limiting factor for the processor's clock frequency.

> As long as RTR is used with care, preserving simplicity and reliability, and without implying in complex reconfiguration management schemes expensive both in hardware and software, it can be a cost-effective option to the extent that it allows FPL to be re-used in time. In this context, an approach like ConCISe provides for a new degree of freedom that can help strike the most cost-effective balance between time and space for the hardware partition.

7.3.3 Estimation of hardware complexity

Another limitation of ConCISe is that the partitioning strategy does not take into account the amount of FPL resources a given candidate might require. The cost function only addresses the potential speed-up. Thus, it is conceivable that better results could have been obtained, had the cost function incorporated estimations for the hardware complexity of candidates. Imagine, for instance, a situation in which a large candidate is selected and occupies as much space in the FPL as a number of other smaller candidates that, individually, all deliver less speed-up than the selected candidate, but that, together, deliver more.

The reason why we did not incorporate hardware complexity estimation in the tool-set is two-fold: Firstly, the hardware cost of a candidate is measured in the number of PTs it requires. How complex each PT needs to be is not relevant, because it simply translates into more or less configurable connections set in the PT (see Figure 1.6, on page 8). The PT is already used anyway. This has the practical effect of reducing the actual relative differences in hardware complexity among different candidates; Secondly, the synthesizable instructions are similar among themselves in terms of hardware complexity (2-input boolean operations or fixed-length shifts). Each of them typically requires one PT for each output bit processed. An exception to this is the `xor` operation, which does not map well onto the AND-OR structure of the CPLD, and requires multiple PTs to be implemented. Some informal tests with the tool-set running in interactive mode showed that, for the DES benchmark, a slightly higher number of candidates could have been synthesized by abandoning a couple of candidates with `xor` operations. As suggested in

Figure 7.2, however, even then the corresponding increase in the speed-up would hardly justify the effort of making the extension to the tool-set. This way, one can look upon the absence of hardware complexity estimation routines in ConCISe as an extra bonus derived from the regularity of the CPLD architecture utilized.

8 Technology mapping optimizations for ConCISe

Corollary 4.2.2, on page 50, claims that the concept of statically encoding several custom operations in a single FPL configuration allows for logic cross-minimization. This chapter discusses techniques [51] that aim precisely at cross-minimizing the logic of the hardware partitions synthesized during the benchmarks of the previous chapter. The objective is to substantiate and quantify the effect of corollary 4.2.2.

This chapter is situated in the low-level tools design capability layer, and refers to technology mapping in particular (see Figure 1.14, page 20).

8.1 Setting the stage

Hardware Description Language (HDL) synthesis tools translate an HDL description of a circuit into logic equations. These equations are then optimized and later mapped onto hardware primitives. Examples of previous work on logic optimization can be found in [64] and [65]. The optimizations performed by the HDL compiler cannot modify the specified behavior of the circuit, as described in the HDL circuit description, because otherwise incorrect behavior could be introduced. Equivalently, the HDL description must completely specify the desired circuit behavior, such that the compiled netlist corresponds to it unambiguously.

There are, however, situations in which the circuit designer has no interest in unequivocally specifying certain aspects of the circuit behavior at HDL level. A possible example is the specification of select words in a multiplexer. Sometimes, the designer is only interested in ensuring that there is a unique correspondence between select words and multiplexer inputs, and not in which particular word encoding will correspond to each particular input. The hardware partitions of ConCISe are an example of one such a situation. In Figure 5.12, on page 68, it is not relevant which DEC word will correspond to which custom operation, as long as to each DEC word correspond a single custom operation. The HDL synthesis tool could, therefore, choose an encoding that favored logic minimization.

Actually, most of the information necessary to efficiently choose a certain select word encoding is only available after the HDL description has been translated into logic equations. Therefore, it is the HDL synthesis tool that must choose the select words, not the circuit designer himself. A method for enabling the HDL synthesis tool to exploit this new logic minimization opportunity is the subject of this chapter. From this point

Select word ($dec_3 dec_2 dec_1 dec_0$)	Corresponding custom operation
0000	0
0001	1
...	...
1011	11
1100	X
...	...
1111	X

Table 8.1: Arbitrary choice of select word encoding.

on, all circuits considered will be hardware partitions automatically synthesized by the ConCISe tool-set.

8.2 Logic minimization opportunities

When different custom operations in a hardware partition share terms in a given output bit, their logic can be cross-minimized, so that the common terms are not replicated multiple times in different AND gates. To illustrate this, let us take the hardware partition used to accelerate the A5 encryption algorithm, as described in Chapter 7. Twelve custom operations are encoded. The choice of the 4-bit select words was arbitrary and is illustrated in Table 8.1. It is simply the binary equivalent of the operation number (starting from zero). The symbol "X" represents a "don't care", and the value zero is put in the output of the multiplexer in Figure 5.12 by default[1].

Let us denote individual bits in the input operands of the RFU as `IMP1b0` up to `IMP1b31`, for the first input, and `IMP2b0` up to `IMP2b31` for the second input. Let us also denote individual bits in the output of the RFU as `OUTb0` up to `OUTb31`. The logic equation produced for the output bit `OUTb1` in the hardware partition of A5 is as follows:

```
OUTb1 =    dec0 & dec1 & !dec2 & dec3 & IMP2b1
         # !dec0 & dec1 & !dec2 & !dec3 & IMP1b8
         # dec0 & dec1 & !dec2 & !dec3 & IMP1b6
         # dec0 & !dec1 & dec2 & !dec3 & IMP1b0
         # dec0 & dec1 & !dec2 & dec3 & IMP1b0
         # !dec1 & !dec2 & !dec3 & IMP2b1
         # !dec0 & !dec2 & dec3 & IMP1b0;
```

Note that ANDs are represented by the symbol "&", ORs by "#", and negations by "!". Each line, therefore, corresponds to a PT in the hardware, as it can be seen in Figure 1.6, page 8. The equation is made up of 7 PTs. Bits of the select word DEC are added to each PT in order to multiplex terms from different operations to the output. This

[1]Naturally, in positive logic, a zero in the output costs no hardware, being the obvious choice for a "don't care".

Select word ($dec_3dec_2dec_1dec_0$)	Corresponding custom operation
0000	0
0001	1
0010	11
0011	11
0100	4
0101	7
0110	5
0111	6
1000	X
1001	X
1010	8
1011	3
1100	X
1101	9
1110	10
1111	2

Table 8.2: Select word encoding that favors cross-minimization.

way, one could read the first line as "`OUTb1` is connected to `IMP2b1` if the select word is $dec_3dec_2dec_1dec_0$ = 1011", that is, if operation 11 is selected. Or the seventh line as "`OUTb1` is connected to `IMP1b0` if the select word is $dec_3dec_2dec_1dec_0$ = 1000 or 1010", that is, if operations 8 *or* 10 are selected.

Note, however, that apart from the select word, 3 PTs are identical, consisting solely of the input signal `IMP1b0`. Still, note that `IMP1b0` is included via operations 5, 8, 10, and 11. Therefore, logic corresponding to four different custom operations occupies not four, but three different PTs. One of the three PTs (the last top-down) could already include `IMP1b0` for two different custom operations, because the select word encoding of those operations differed by a single bit (`dec1`). *That is equivalent to saying that cross-minimization of the logic corresponding to different custom operations was possible due to the particular select word encoding chosen.* An analogous situation occurs for other 2 PTs that include the input signal `IMP2b1`.

In general, by properly choosing the encoding of the select words, one can allow for more cross-minimization. Table 8.2 illustrates an alternative encoding. Note that one of the "don't cares" is replaced by a second encoding of operation 11. This does not damage the functional correctness of the resulting circuit but often helps in cross-minimizing the logic. The resulting equation for `OUTb1`, as given by the HDL synthesis tool, is shown below:

```
OUTb1 =   dec0 & dec1 & dec2 & dec3 & IMP1b8
        # dec0 & dec1 & !dec2 & dec3 & IMP1b6
        # dec1 & !dec2 & !dec3 & IMP1b0
# !dec2 & !dec3 & IMP2b1
# !dec0 & dec1 & IMP1b0
```

Note that only 2 PTs are now needed to carry the signal `IMP1b0` for four different custom operations, instead of the 3 PTs of the previous case. In addition, only 1 PT is now needed to carry the signal `IMP2b1` for 3 different custom operations, instead of the 2 PTs of the previous case.

Proposition 8.2.1. *The cross-minimization opportunities of corollary 4.2.2 can be exploited by properly encoding the select words of the multiplexer in ConCISe's hardware partitions.*

8.3 The minimization problem

The example illustrated in the previous section concentrates on the logic equation of a single output bit. The aim of the general minimization problem, however, is to search for an encoding that minimizes the logic of the circuit implementation as a whole (all output bits).

This problem has been tackled in the logic synthesis community as the *input encoding* problem [64]. State-of-the-art methods for input encoding typically minimize the logic by encoding inputs in words of variable bit-width. In that case, the optimal solution can be calculated exactly. For ConCISe, however, the `DEC` word must always be encoded in four bits. Because fixed-length encoding is more complex, current methods use heuristic search mechanisms for the minimization. These methods try to encode the inputs in such a way so as to minimize a cost function that estimates the amount of logic required to implement the circuit [64].

The method we developed to minimize the logic of ConCISe's hardware partitions is based on the state-of-the-art of input encoding. A heuristic search based on simulated annealing looks for an optimal encoding. However, because the dimensions of our particular problem are modest (only 4 bits to encode a handful of custom operations), we can afford to evaluate with greater precision the cost of a given encoding during the heuristic search.

Since input encoding in general is a known logic synthesis technique, descriptions of the particular algorithms and methods we used will be left out of this thesis. Details can be found in [51].

8.4 Benchmark results

We have built a prototype input encoding software. The prototype reads as input the same equations file used by ConCISe simulators to process PSI instances. The aim of this section is to use the prototype to quantify the benefits of our minimization method in the ConCISe framework. The method can increase the cost-effectiveness of ConCISe in two alternative ways. On the one hand, it can contribute to higher performances, since more efficient logic minimization could allow more custom operations to be encoded in the same amount of FPL resources considered in Chapter 4. In this case, the target result would indicate how many more custom operations would fit, and how much more speed-up they would allow for. On the other hand, the minimization could reduce the need for FPL resources, allowing the same number of custom operations to be synthesized in a smaller RFU. In this case, the target result would indicate how much less silicon the RFU would require.

Note that either indication, alone, already suffices to quantify the extent to which the method can reduce the logic, whatever the way in which this reduction will later be used. In this context, we will only look into how much smaller the minimization method could allow the RFU to be, while preserving levels of performance compatible to those observed in Chapter 7.

The prototype software found cross-minimization opportunities in the hardware partitions of four of the five benchmarks used: A5, Loki97, MD5, and DES.

A5

In total, 83 PTs have been saved. This is a reduction of approximately 31% relative to the *worst-case scenario*, which we now define as a hypothetical scenario in which the arbitrary encoding of select words is such that no cross-minimization is feasible. The actual arbitrary choice of Table 8.1, however, already allowed for a "by chance" cross-minimization of 58 PTs. Our method, therefore, actually reduced the total number of PTs utilized by approximately 11.9%.

Loki97

A total saving of 85 PTs is achieved with the final encoding, or approximately 32.3% relative to the worst case scenario. The arbitrary choice of select words (the binary equivalent of the operation number), however, already cross-minimized 48 PTs "by chance". Therefore, our method actually reduced the total number of PTs utilized by approximately 17.2%.

MD5

The final encoding optimized away 64 PTs. However, the arbitrary choice of select words originally made (again, the binary equivalent of the operation number) already accounted for a "by chance" cross-minimization of 32 PTs. Therefore, our method actually reduced the total number of PTs by approximately 10%.

DES

The total savings amounted to 10 PTs. Two PTs had already been cross-minimized "by chance", so the savings represent an effective reduction of approximately 3.33% in the number of PTs.

Benchmark	CPU time (user plus system)
A5	$\approx$ 70 seconds
Loki97	$\approx$ 53 seconds
MD5	$\approx$ 9 seconds
DES	$\approx \frac{1}{2}$ second

Table 8.3: Execution times of the prototype minimization software.

8.5 Summary of results and indications of silicon savings

Figures 8.1 and 8.2 summarize all results. All final solutions were actually run through the HDL synthesis tool, so that the cost calculated by our software could be checked against the actual number of PTs utilized, as given by the compiled circuit.

Table 8.3 shows the CPU time (as given by `clock()`) taken by the prototype software to converge to the final solutions in a Sun UltraSPARC 10 running at 336 MHz. The software was compiled with `gcc`, and compiler optimizations on. Compare Table 8.3 to trying out all select word encoding possibilities (16! $\cong$ 21 trillions), running each through the HDL synthesis tool (roughly 30 seconds per run, as shown in Table 7.3, page 102) to evaluate its cost. Still, the prototype was written for clarity, not at all for speed. There is ample room for manual optimization in its code.

If we reduce the number of configurable PTs in ConCISe's RFU by 10%, we can see in Figure 8.2 that, except for the DES benchmark, this reduction would allow for the same custom operations to be synthesized in the hardware partitions of all three other benchmarks, as long as the proposed minimization method be used[2]. For DES, a few custom operations would need to be abandoned. However, considering Figure 7.2, on page 96, this reduction should have only a slight effect on the achievable speed-up. Finally, although our minimization method had no effect on the Magenta benchmark, its hardware partition does not use all available FPL resources, since there was no need for more than 6 custom operations to be synthesized. All those six operations could still fit in an RFU with 10% less PTs (see Table 7.2, on page 96).

In conclusion, our method allows for a 10% reduction in the number of configurable PTs in the RFU, preserving the performance results already observed for four of the benchmarks, and with only a slight reduction in performance for the DES benchmark. We estimate that reducing the number of PTs by 10% would lead to roughly 4% reduction in the RFU's area [61]. Although this is certainly a modest result, the savings come at no recursive cost.

[2]Naturally, this would depend on how and which PTs are removed from the RFU. Although the rationale illustrated here has safe general conclusions, precise results could only be achieved by actually modifying the RFU architecture and the fitter software, and re-running the synthesis.

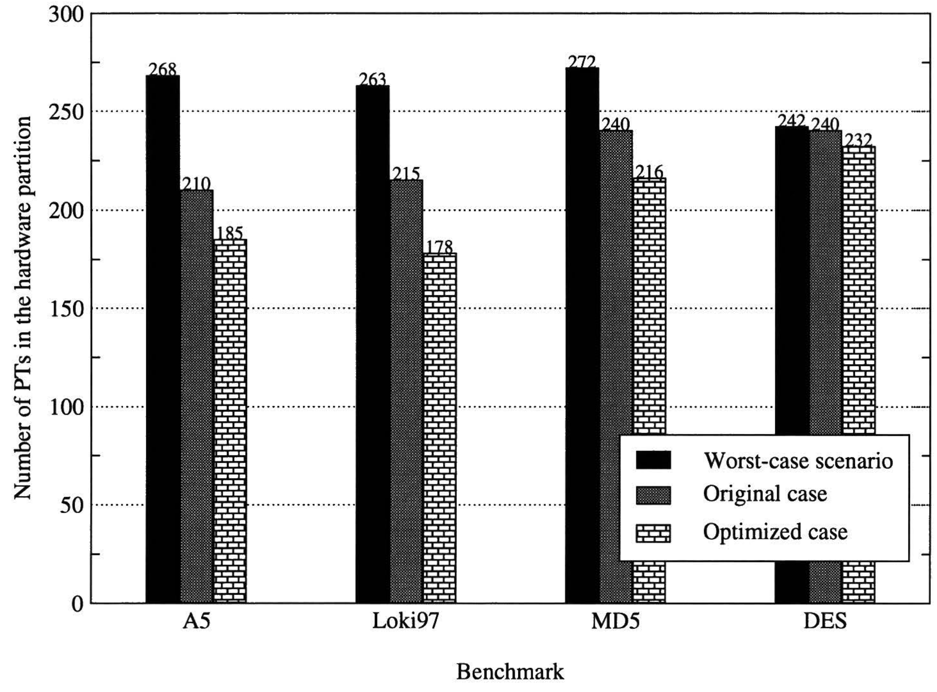


Figure 8.1: Benchmark results in terms of absolute PT reduction.

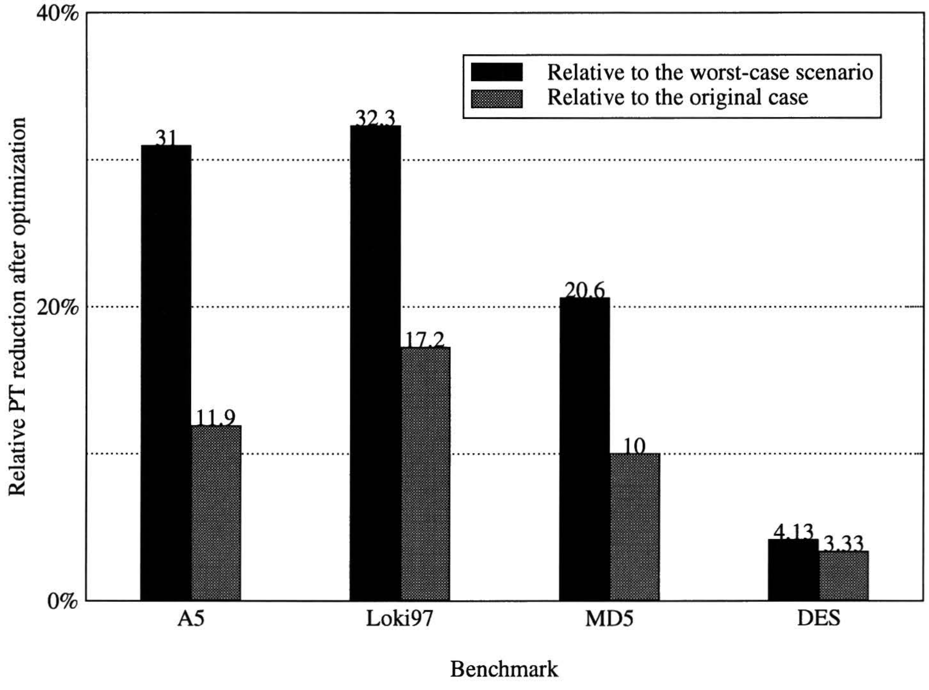


Figure 8.2: Benchmark results in terms of relative PT reduction.

8.6 Final considerations

The impact of proposition 8.2.1, on page 110, has been quantified by the results presented in the previous section. As a consequence, corollary 4.2.2, on page 50, has been substantiated. Note that the logic minimization opportunities discussed in this chapter are a direct consequence of the basic design principle of ConCISe, illustrated by proposition 4.2.1, on page 47. This partially illustrates the conceptual integrity of the entire approach.

9 Conclusions

"*And the message? Despite assertions to the contrary, the lode of discovery is far from worked out.*"
John Maddox, in "What remains to be discovered".

These conclusions are divided in three sections: A summary of the previous chapters of this dissertation; Some possibilities for future work that can potentially extend the applicability of the concepts and innovations elaborated upon in this thesis; And my visions regarding trends and perspectives of reconfigurable computing in particular, and of the role it plays in computing in general. The reader will notice that the conclusions *per se* are distributed among all the three sections.

9.1 Summary

We have seen that Field Programmable Logic (FPL) devices are hardware circuits that can be customized after fabrication to perform a certain function. Just as in programmable processors, control data loaded in memory elements of the FPL controls the behavior of the computationally-active circuit elements. The configuration plane of FPL, however, has orders of magnitude more memory bits than the instruction words of typical programmable processors (even VLIWs). This allows for a much finer granularity of control over data-path elements, in a true computing-in-space (data-flow computing) fashion. This way, the intrinsic parallelism of functions can be exploited in a FPL implementation.

We have analyzed three different levels of parallelism in a FPL-based, computing-in-space approach. They were: boolean, bit-level parallelism, in which individual bits of operands are processed independently and in parallel, and boolean logic minimization opportunities can be exploited; Lateral parallelism, or Instruction-Level Parallelism, in which data-independent operations are processed concurrently; And time parallelism, or loop-level pipelining, in which the computations of multiple iterations of a loop are overlapped. We have seen that modern, general-purpose processors can exploit most of the available lateral parallelism, but are very limited with respect to bit-level and time parallelism.

In this thesis, we introduced ConCISe ("Compiler-driven, CPLD-based Instruction Set accelerator"), a programmable RISC processor that features a Complex Programmable Logic Device (CPLD) as a Reconfigurable Functional Unit (RFU). ConCISe targets embedded encryption applications. Just like the ALU, ConCISe's RFU must execute in a single clock cycle, during the execution stage of the RISC pipeline. However, unlike the ALU, the RFU can implement segments of the application's critical path, where bit manipulations are intense, in a true computing-in-space fashion. The mapping in space exploits the available boolean, bit-level parallelism and improves performance. For this reason, the approach is orthogonal to other processor design techniques like bigger caches or multiple instruction issue slots in VLIWs. We have also introduced a general think-model that can provide some basic guidelines for a designer to evaluate the cost-effectiveness of reconfigurable computing approaches like ConCISe.

The ConCISe processor reconfigures its own RFU prior to executing an application, such that a set of application-specific instruction set extensions is available during execution. Unlike other reconfigurable processors, ConCISe does not use run-time RFU reconfiguration, due to its complexity and associated reconfiguration latency overheads. Instead, ConCISe encodes several custom operations in a single RFU configuration. The approach is simpler, more reliable, easier to verify, and eliminates reconfiguration overheads. Tests conducted have shown that the RFU proposed is large enough to implement most of the custom operations necessary to map the bit manipulations in the applications' critical paths. The price is that more FPL is necessary in ConCISe's RFU than in similar reconfigurable processors. It boils down to a trade-off between the amount of silicon (cheaper with each new process technology) and the simplicity of design, verification, and programming (ever more critical issues in today's semiconductors industry).

The core of ConCISe is its programming tool-set, capable of automatically partitioning an application into software and hardware, and of automatically synthesizing the hardware partition in the RFU hardware. Since the ease and speed of (re-)programming encryption processors are crucial factors in today's fast-changing Internet world, the tool-set comes to fulfill an important demand. Unlike prior works, ConCISe uses innovative graph-based techniques during partitioning, trying to maximize performance and the utilization of RFU resources. These techniques have been inspired by the field of compiler technology and blended with high-level hardware synthesis concepts.

Some new theoretical basis needed by the techniques employed have also been laid down. In particular, an injection that allows one to transfer a partial graph covering problem from the domain of Directed Acyclic Graphs (DAGs) to an abstract domain of directed trees has been introduced. The covering problem can be more efficiently and easily solved in the tree domain. The theoretical results are more general than the ConCISe framework itself, and could be a useful tool for DAG-based instruction selection problems normally encountered in compiler technology.

Benchmark results show that the ConCISe approach is a cost-effective investment. For some benchmarks, an RFU-extended MIPS processor exhibited a level of performance more than 1.5 time that of the same processor without the RFU. Given that all recurring costs of ConCISe reside in the RFU, which in turn represents only a small investment in silicon, this is a very promising result. In addition, silicon is an ever cheaper commodity in today's semiconductors world. The core of ConCISe is its programming tool-set, which represents a non-recurring cost. In this context, the benchmark results indicate that the approach can be an attractive solution for high-volume, encryption-enabled, embedded electronics.

The static encoding of several custom operations in a single RFU configuration, ConCISe's basic design principle, unfolds into interesting opportunities at the hardware synthesis and logic optimization levels of the tool-set. The logic corresponding to different operations can be statically cross-minimized by the tools, saving silicon or allowing for more custom operations to be mapped onto hardware. We have written a logic minimization module for ConCISe that is based on techniques of input encoding. We have used this module to demonstrate and quantify the impact of cross-minimization in the hardware partitions mapped onto ConCISe's RFU.

9.2 Applicability

Throughout this dissertation, we have positioned ConCISe as a solution for a specific application domain, embedded encryption. Note that this does not pose a strong limitation to the applicability of the concepts and innovations discussed in the previous chapters. In today's Internet world, in which network connectivity no longer applies only to desktop PCs, but also to phones, TVs, domestic appliances and, in a near future, even to one's clothes, data protection and privacy concerns are paramount. In this scenario, embedded encryption is, alone, already a very large application domain, which only tends to grow.

Still, ConCISe's basic concepts can be extended to other application domains. As already pointed out before, the ideas in chapter 6 are more general than the framework that motivated their development in the first place. ConCISe's overall graph-based partitioning strategies, as described in Chapter 5, could also be extended to DSP applications in the following way. In stream-oriented DSP, like multimedia applications, time parallelism (i.e. loop-level pipelining) is crucial to meet the typical high throughput requirements. In this case, and given that multiplies and accumulates are typical DSP operations, a CPLD would no longer be a suitable choice for the FPL accelerator. Instead, one could use some coarser-grained FPGA architecture. Other front-end compiler optimizations not implemented in ConCISe, like aggressive load-store elimination and loop transformations, would also be necessary. Finally, low-level mapping techniques like re-timing, resource-sharing, and time-folding [69], would be important to extract the available parallelism. In general lines, these are the directions in which future work will be carried out.

9.3 Visions and trends

ConCISe's tool-set is an early sign of a future in which high-level synthesis tools and compilers will converge. Actually, one could say that the division between the two has always been somewhat artificial. Compilers map a high-level specification of an application onto a piece of hardware, mostly in time. High-level synthesis tools map a high-level specification of an application onto a piece of hardware, mostly in space. Reconfigurable processors like ConCISe support both styles of computation, in time and space, blurring the border between these tools.

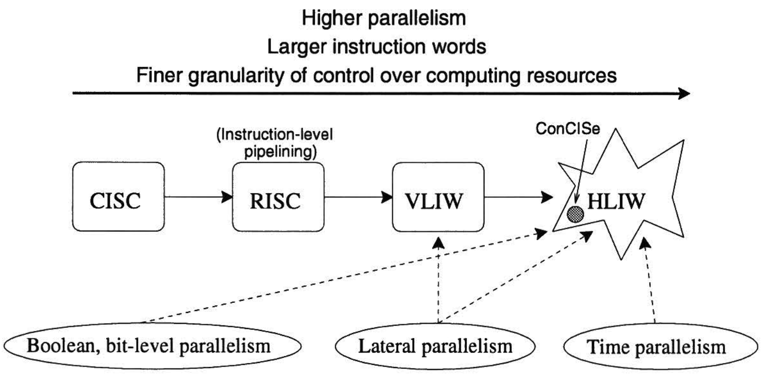

Higher parallelism
Larger instruction words
Finer granularity of control over computing resources
(Instruction-level pipelining)
ConCISe
CISC
RISC
VLIW
HLIW
Boolean, bit-level parallelism
Lateral parallelism
Time parallelism

Figure 9.1: General caricature of the evolution of programmable processors, in terms of parallelism.

In a sense, because the FPL configuration plane is analogous to an instruction word with thousands of bits, reconfigurable processors like ConCISe could be called "HLIW processors", for "Hyper Long Instruction Word"[1]. See Figure 9.1. For being so long, these instruction words take time to be transfered from the background memory and, therefore, cannot be changed (reconfigured) as frequently as instructions are issued in standard processors. In the case of ConCISe, the "hyper long instruction word" is loaded prior to execution and does not change dynamically.

Note that this is a unifying perspective. Hardware and software converge into HLIW processors where parallelism can be exploited at all levels. Computing-in-space and computing-in-time can be traded off as required by the application domain. Instead of compilers and synthesis tools, general *function mapping* tools are used.

In this context, ConCISe would be an instance of a HLIW processor that can only exploit boolean, bit-level parallelism (key in the embedded encryption domain), not loop-level pipelining or ILP. Its innovative tool-set approach points out ways in which compilers and synthesis tools can be integrated in a synergistic way, and its basic design principle suggests a cost-effective approach to properly balance the computational load between space and time.

[1]Another option could be "Hardware Language-based Instruction Word", since these "words" are synthesized from hardware descriptions.

Bibliography

[1] S. Trimberger. *Field-Programmable Gate Array Technology*, Kluwer, MA, 1994.

[2] J. H. Jenkins. *Designing With FPGAs and CPLDs*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1994.

[3] G. J. Hekstra *et al.* TriMedia CPU64 Design Space Exploration, *Proc. IEEE Intl. Conf. Computer Design*, pp. 599-606, Austin, 1999.

[4] S. Hauck. The Roles of FPGAs in Reprogrammable Systems, *Proc. of the IEEE*, Volume 86, Issue 4, pp. 615-638, April 1998.

[5] H. Mangione *et al.* Seeking Solutions in Configurable Computing, *IEEE Computer*, 30(12), pp. 38-43, December 1997.

[6] B. Radunović *et al.* A Survey of Reconfigurable Computing Architectures, *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1482, Springer-Verlag), pp. 376-385, Tallinn, 1998.

[7] L. Wirbel. Reconfigurable Computing Hits Software Radio, *Electronic Engineering Times*, Issue 1045, pp. 1, 14-15, January 25, 1999.

[8] P. Masters and P. Athanas. Reconfigurable Computing Offers Options For 3G, *Wireless Systems Design*, pp 20-25, January 1999.

[9] G. Brebner. Field-Programmable Logic: Catalyst for New Computing Paradigms, *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1482, Springer-Verlag), pp. 49-58, Tallinn, 1998.

[10] P. Laity and S. Lam. Combining FPGA Cores Can Extend DSP Design Performance, *Wireless Systems Design*, pp. 51-54, July 1998.

[11] *XC4000E and XC4000X Series Field Programmable Gate Arrays*, Product Specification, Version 1.6, Xilinx Corporation, May 14, 1999.

[12] *Coprocessor Field Programmable Gate Arrays, AT6000 Series*, Data Sheet, Atmel Corporation, October 1999.

[13] A. DeHon. The Density Advantage of Configurable Computing, *IEEE Computer*, Vol. 33, No. 4, pp. 41-49, April 2000.

[14] A. DeHon. *Reconfigurable Architectures for General-Purpose Computing*, Ph.D. Dissertation, Massachusetts Institute of Technology, October 1996.

[15] D. Bursky. Tool Suite Enables Designers to Craft Customized Embedded Processors, *Electronic Design*, pp. 33-38, February 8, 1999.

[16] C. Iseli and E. Sanchez. Spyder: A SURE (SUperscalar and REconfigurable) Processor. *The Journal of Supercomputing*, 9, pp. 231-252, 1995.

[17] B. L. Hutchings and M. J. Wirthlin. Implementation Approaches for Reconfigurable Logic Applications, *Proc. Of the 5th Intl. Workshop on Field Programmable Logic and Applications* (Lecture Notes in Computer Science 975, Springer-Verlag), pp. 419-428, Oxford, August 1995.

[18] M. J. Wirthlin and B.L. Hutchings. DISC: The Dynamic Instruction Set Computer, *Proc. FPGAs for Fast Board Development and Reconfigurable Computing*, John Schewel Editor, Proc. SPIE 2607, pp. 92-103, 1995.

[19] M. J. Wirthlin *et al.* The Nano Processor: A Low Resource Reconfigurable Processor, *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 23-30, Napa, April 1994.

[20] S. Hauck *et al.* The Chimaera Reconfigurable Functional Unit, *Proc. IEEE Symp. On FPGAs for Custom Computing Machines*, pp. 87-96, Napa, April 1997.

[21] R. Razdan and M.D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units, *Proc. 27th Annual IEEE/ACM Intl. Symp. On Microarchitecture*, pp. 172-180, November 1994.

[22] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, 1994.

[23] R. Razdan, K. Brace, and M.D. Smith. PRISC Software Acceleration Techniques, *Proc. 1994 IEEE Intl. Conf. On Computer Design*, pp. 145-149, October 1994.

[24] S. Sawitzki, A. Gratz, and R.G. Spallek. Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays, *Proc. Of Field- Programmable Logic and Applications* (Lecture Notes in Computer Science 1482, Springer-Verlag), pp. 411-415, Tallinn, August/September, 1998.

[25] R. D. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 126-135, Los Alamitos, April 1996.

[26] J. R. Hauzer and J. Wawrzynek. GARP: A MIPS Processor with a Reconfigurable Coprocessor, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 12-21, April 1997.

[27] T. J. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing, *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1482, Springer-Verlag), pp. 248-257, Tallinn, 1998.

[28] T. J. Callahan *et al.* Fast Module Mapping and Placement for Datapaths in FPGAs, *Proc. Intl. Symp. on Field Programmable gate Arrays*, pp. 123-132, Monterey, February 1998.

[29] L. Pozzi. *Methodologies for the Design of Application-Specific Reconfigurable VLIW Processors*, Ph.D. Thesis, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 2000.

[30] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis, *IEEE Computer*, 26(3), pp. 11-18, March 1993.

[31] C. Ebeling *et al.* RaPiD: Reconfigurable Pipelined Datapath, *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1142, Springer-Verlag), pp. 126-135, Darmstadt, 1996.

[32] C. Ebeling *et al.* Configurable Computing: The Catalyst for High-Performance Architectures, *Proc. IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors*, pp. 364-372, Zurich, July 1997.

[33] H. Singh *et al.* MorphoSys: An Integrated Re-configurable Architecture, *The Application of Information Technologies (Computer Science) to Mission Systems*, NATO Research and Technology Organization, pp. 10/1-11, Monterey, April 1998.

[34] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 157-166, Los Alamitos, April 1996.

[35] R. Bittner, P. Athanas and M. Musgrove, Colt: An Experiment in Wormhole Runtime Reconfiguration, *Proc. of the SPIE, The International Society for Optical Engineering*, Vol. 2914, pp. 187-194, Boston, November 1996.

[36] R. W. Hartenstein *et al.* Using the KressArray for Configurable Computing, *Proc. of the SPIE, The International Society for Optical Engineering*, Vol. 3526, pp. 150-161, Boston, November 1998.

[37] E. Waingold *et al.* Baring It All to Software: Raw Machines, *IEEE Computer*, 30(9), pp. 86-93, September 1997.

[38] D. A. Clarck and B. L. Hutchings. Supporting FPGA Microprocessors Through Retargettable Software Tools, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 195-203, Los Alamitos, April 1996.

[39] D. C. Cronquist *et al.* Specifying and Compiling Applications for RaPiD, *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pp. 116-125, Monterey, April 1998.

[40] M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures, *Proc. of Reconfigurable Architectures Workshop* (Reconfigurable Architectures - High Performance by Configware, ITpress Verlag), Geneva, April 1997.

[41] M. Weinhardt. Integer Programming for Partitioning in Software Oriented Codesign, *Proc. Of the 5th Intl. Workshop on Field Programmable Logic and Applications* (Lecture Notes in Computer Science 975, Springer-Verlag), pp. 227-234, Oxford, August 1995.

[42] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann, 1996;

[43] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design, The Hardware/Software Interface*, Morgan Kaufmann, 1998.

[44] *Integrating Product-Term Logic in APEX 20K Devices*, Application Note 112, Version 1.0, Altera Corporation, April 1999.

[45] *XPLA Professional User's Manual*, Xilinx Corporation.

[46] *Using ABEL with Xilinx CPLDs*, Application Note XAPP075, Version 1.0, Xilinx Corporation, January 1997.

[47] B. Kastrup, K. Nowak and J. van Meerbergen. Seeking (the right) Problems for the Solutions of Reconfigurable Computing. *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1673, Springer-Verlag), pp. 520-525, Glasgow, 1999.

[48] B. Kastrup *et al.* ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator, *Proc. Intl. Symp. on Field-Programmable Custom Computing Machines*, pp. 92-101, IEEE Computer Society Press, Napa, April 1999.

[49] B. Kastrup *et al.* Compiling Applications for ConCISe: An Example of Automatic HW/SW Partitioning and Synthesis. *Proc. Of Field-Programmable Logic and Applications* (Lecture Notes in Computer Science 1896), pp. 695-706, Villach, August 2000.

[50] B. Kastrup. Automatic Hardware Synthesis for a Hybrid Reconfigurable CPU Featuring Philips CPLDs, *Proc. of Workshop on Reconfigurable Computing*, held in conjunction with the Intl. Symp. On Parallel Architectures and Compilation Techniques, pp. 5-10, Paris, October 1998.

[51] B. Kastrup and O. Moreira. A Novel Approach to Minimising the Logic of Combinatorial Multiplexing Circuits in Product-Term-Based Hardware. *Proc. of the Intl. Symp. on Digital Systems Design*, EUROMICRO 2000, pp. 164-171, IEEE Computer Society Press, Maastricht, September 2000.

[52] D. Matsumoto. Intel Prods Industry to Secure Networked PCs, *Electronic Engineering Times*, pp. 24, January 25, 1999.

[53] B. Schneier and D. Whiting. Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor, *Proc. of the International Workshop on Fast Software Encryption*, pp. 242-259, Springer-Verlag, January 1997.

[54] N. Lange. Single-Chip Implementation of a Cryptosystem for Financial Applications, *Proc. Financial Cryptography First International Conference*, pp.135-144, Anguilla, February 1997.

[55] *XCR3960, 960 macrocell SRAM CPLD*, Data Sheet, Xilinx Corporation, July 1998.

[56] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*, Second Edition, John Wiley & Sons, Inc., 1996.

[57] ftp://ftp.psy.uq.oz.au/pub/Crypto/DES/

[58] http://www.gladman.uk.net/

[59] D. P. Leach and A. P. Malvino. *Digital Principles and Applications*, Glencoe McGraw Hill, March 1994.

[60] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*, Wiley, February 1990.

[61] Estimations based on private communications with engineers at the former Programmable Products Group of Philips Semiconductors, designers of the XPLA2 CPLDs.

[62] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[63] Private communications with engineers at Philips Crypto B.V.

[64] P. Ashar, S. Devadas, and A. R. Newton. *Sequential Logic Synthesis*, Kluwer Academic Publishers, 1992.

[65] G. de Micheli. *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[66] R. Gould. *Graph Theory*, The Benjamin/Cummings Publishing Company, Inc., 1988.

[67] W. T. Tutte. *Graph Theory*, Cambridge University Press, 1984.

[68] T. H. Cormen *et al. Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1990.

[69] A. van der Werf. *Processing Unit Design*, Ph.D. dissertation, Eindhoven University of Technology, 1998.

[70] A. S. Sedra and K. C. Smith. *Microelectronic Circuits*. Saunders College Publishing, 1991.

Index

Curriculum Vitae

Bernardo Kastrup was born on 21 October, 1974, in Niterói, Rio de Janeiro, Brazil. He graduated in electronic engineering from the Federal University of Rio de Janeiro (UFRJ), Brazil, in 1997. Immediately after his University studies, he joined the European Laboratory for Particle and Nuclear Physics, CERN, in Geneva, Switzerland. There, he worked for one year with the ATLAS collaboration team, on hardware and software aspects of distributed data acquisition systems. In January 1998, he joined Philips Research in Eindhoven, the Netherlands. There, he developed the work described in this thesis as the project leader of the Embedded Reconfigurable Computing project. His current research interests include compiler technology, high-level synthesis, programmable logic, reconfigurable computing, and evolvable hardware.

Stellingen

behorende bij het proefschrift

Automatic Synthesis of Reconfigurable Instruction Set Accelerators

van Bernardo Kastrup

Technische Universiteit Eindhoven, mei 2001

1. *The added-value of reconfigurable processors is in their ability to exploit large amounts of bit-level parallelism and loop-level pipelining* (see Chapter 1 of this dissertation).

2. *A VLIW processor with distributed register files, partially-connected interconnect, and dozens of instruction issue slots, is a reconfigurable processor.*

3. *Reconfigurable processors are most cost-effective if they are not general-purpose, but application-domain-specific* (see Chapter 2 of this dissertation).

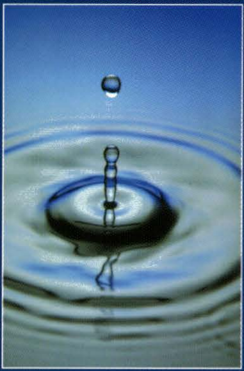
4. *Run-time reconfiguration is like alcoholic drinks: exciting, but good only with moderation* (see Chapters 3, 4, 5, and 7 of this dissertation).

5. *The technique of partial graph covering can be effectively applied to the problem of automatic hardware/software partitioning* (see Chapters 5 and 6 of this dissertation).

6. After reading a theorem and the corresponding mathematical proof, one may think the theorem was trivial and no proof was necessary. However, if one arrives at this conclusion only *after* going through the proof, *then it was not only necessary, but also well designed.*

7. So many different "fundamental particles" were being discovered in the 1950s that J. Robert Oppenheimer, father of the atomic bomb, said in frustration: "the Nobel Prize in physics shall go to the physicist who does *not* find a new particle". *The multitude of different processor architectures of today is as disturbing as the "fundamental particles" dilemma of the 1950s.*

8. The theory that brought order and understanding to the chaos of "fundamental particles" is today known as the "Standard Model". *Reconfigurable Computing can*



"Consider the surface of a pond, which can support many different types of waves and ripples. The hardware -namely the water itself - is the same in all cases, but it possesses different possible modes of excitation. Such software excitations of the same hardware can all be distinguished from each other."

Douglas Hofstadter, in "Gödel, Escher, Bach: An Eternal Golden Braid".

ISBN: 90-74445-50-0

The work described in this thesis has been carried out at the Philips Research Laboratories Eindhoven as part of the Philips Research programme