Reconfigurable Computing: A Survey of Architectures and Synthesis Tools V1.0

D. de Leeuw Duarte
Student ID: 1027875

Parallel and Distributed Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Science & Technology B.V., Delft

October 19, 2005

Contents

1 Introduction **5**
- 1.1 Overview . . . 5
- 1.2 Related work . . . 6
- 1.3 Background and motivation . . . 6
- 1.4 Assessing reconfigurable computing platforms . . . 10

2 Reconfigurable architectures **12**
- 2.1 System properties . . . 12
  - 2.1.1 Reconfigurability . . . 12
  - 2.1.2 Coupling . . . 12
  - 2.1.3 Granularity . . . 13
  - 2.1.4 Reconfiguration time . . . 13
- 2.2 Architectures . . . 13
  - 2.2.1 Field Programmable Gate Arrays . . . 14
  - 2.2.2 Garp . . . 14
  - 2.2.3 PipeRench . . . 15
  - 2.2.4 Raw . . . 17
  - 2.2.5 Chimaera . . . 18
  - 2.2.6 Other architectures . . . 19
- 2.3 Conclusion . . . 19

3 Synthesis tools **21**
- 3.1 Language considerations . . . 21
  - 3.1.1 Structural vs. behavioral modeling . . . 22
  - 3.1.2 Declarative vs. impertative programming . . . 22
- 3.2 Models of computation . . . 23
  - 3.2.1 Gate level description . . . 23
  - 3.2.2 Data flow network . . . 24
  - 3.2.3 Communicating sequential processes (CSP) . . . 24
  - 3.2.4 Custom instruction set architecture . . . 25
- 3.3 Synthesis tools . . . 25
  - 3.3.1 XST . . . 25
  - 3.3.2 DEFACTO . . . 27
  - 3.3.3 SA-C . . . 28
  - 3.3.4 Machines . . . 29
  - 3.3.5 Lava . . . 31
  - 3.3.6 GCC/Chimaera . . . 33
  - 3.3.7 Other synthesis tools . . . 34
- 3.4 Conclusion . . . 35

4 Designing synthesis tools **37**
4.1 Design strategies . . . 37
4.2 Discussion . . . 39
4.3 Conclusion . . . 40

5 Conclusion **41**

Acknowledgements

Before you is the end product of a research assignment that was carried out as part of the Computer Science MSc. curriculum at Delft University of Technology. The project was completed under the supervision and guidance of dr. ir. Arjan van Gemund of the Parallel and Distributed Systems Group and ir. Sidney Cadot of Science & Technology B.V., Delft, to whom I wish to extend my thanks. In addition, I wish to thank Science & Technology B.V. for providing the additional resources to complete this project.

Chapter 1

Introduction

Since its conception in the 1940's, John Louis von Neumann's sequential stored program computer model has become the predominant paradigm in computer design. Whether it is the ARM7TDMI spinning digital music files in today's portable digital music players or the PowerPC 440 FP2 crunching astronomic amounts of data in IBM's BlueGene/L supercomputer, the principles of operation are largely the same. Given the appropriate software, the classic Von Neumann design can be deployed on any type of computational problem.

As any person with hands-on Swiss army knife experience can attest, the most flexible tool doesn't necessarily make for the most effective one. Despite its proven agility, all except the most purist of survivalists would prefer a dedicated can opener over the pocket utensil when challenged with a hermetic tin of beans. In computing, the situation is not much different. While stored program computers can easily be adapted to perform a variety of tasks, their design is inevitably suboptimal for any specific problem. It is for this reason that the idea of *custom computing devices* should appeal to the academically curious.

There are more urgent reasons to think about application specific computing devices than curiosity, however. These reasons come in the form of both threat and opportunity. On the one hand, steady evolution in the field of reprogrammable logic devices has delivered the tools for the implementation of custom computing machines to our doorsteps. Inexpensive 'reconfigurable fabrics' have become a technical and commercial reality over the past decade. On the other hand, we see conventional CPU design approaching its physical limits, leaving this new technology waiting to be exploited as an alternative for low budget, high performance computing. With the reconfigurable hardware already in place, all that remains to be desired are proper development tools that can make reconfigurable fabrics accessible for routine application development. Fundamental to the development of such tools is the resolution of one question: how does one transform an algorithm into a hardware design?

1.1 Overview

This text is the end result of an exploratory journey into the field of reconfigurable computing. Over a period of several months, the author immersed himself in the body of literature surrounding this subject. The aim of this report is to give an account of that educational experience, without the pretense of achieving complete coverage of the subject matter. It is the author's hope, however, that the reader will gain an appreciation of the challenges that lie at the heart of this interesting topic.

This report is structured as follows. Section 2 contains an in-depth look at hardware or-

ganization in reconfigurable systems. Section 3 reviews the proposed solutions to the most prominent challenge in reconfigurable computing today: the design of convenient programming languages and efficient compilers for reconfigurable platforms. Section 4 aims to capture some of the early signs of convergence in this area, in the form of a series of design strategies for synthesis tools. Section 5 concludes this work with a summary of the findings, and a research proposal for an MSc. thesis project. These main themes will be preceded by a short historical background (section 1.3), and the presentation of an elementary framework for systems evaluation (section 1.4) in the remainder of this introductory section.

1.2 Related work

Reconfigurable computing systems have seen a fair amount of research activity in the last decade. A number of surveys have appeared, which discuss hardware organization and synthesis tool design in a manner similar to this work. Compton and Hauck [19] have made a taxonomy of the hardware organization and synthesis tool designs of RC systems. In contrast to this work, which considers a variety of hardware platforms, the scope of their study was limited to FPGA based systems. Hartenstein [28] has compiled short technical summaries of twenty reconfigurable hard- and software platforms. The accent of his work is on the enumeration of existing systems, rather than the identification of common properties of the designs and/or the fundamental difficulties of reconfigurable computing. Sima et al [41] have taken a formal approach to systems classification, which is more exact, yet less intuitive than ours. Mangione-Smith et al [36] have written an excellent shorter introductory survey about reconfigurable computing, which deliberately lacks the practical details and difficulties that are at the center of our discussion.

1.3 Background and motivation

The topic of reconfigurable computing is no exception to the rule that new ideas and technologies tend to build upon existing ones. When studying this exciting new field, it is easy to lose oneself in the myriad of details and valuable insights from related topics, such as compiler construction, language design, parallel algorithms, computation theory and semiconductor engineering. At these levels of detail, our goals and the possible means of achieving them are easily confused. To maintain a proper perspective, it is instructive to place this technology in light of the greater goals that drive the disciplines of computer science and engineering. In addition to establishing a good starting point, this will identify the major design parameters that govern the development of reconfigurable computing systems.

In the remainder of this section, the historical goals and concepts of theoretical computer science and computer engineering will be reviewed. This brief historical overview will establish the motivation and goals of reconfigurable computing as an alternative to conventional architectures.

Problems

Computer science is the study of systematic solutions to well formulated problems. The fundamental question of what constitutes a problem in this context is not trivially answered[1], but for our purposes we will equate *problems* to *binary relations over sets*. The elements of the domain of such a relation are the *instances* of the problem. The elements of the codomain that are paired to an instance by the relation are called its *solutions*. The

[1] At best, we encounter intuitive definitions in the literature [37]. Some classic texts simply avoid a definition altogether [33, 40].

anatomy of a problem in computer science is illustrated in Figure 1.1.

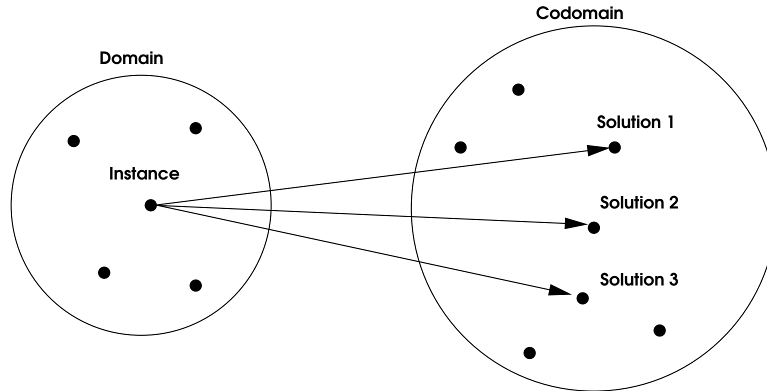

Figure 1.1: A problem in computer science.

To demonstrate the applicability of this definition, consider the classic Travelling Salesman Problem:

Example 1.3.1 (Traveling Salesman Problem (TSP)). The objective of *TSP* is to find *Hamiltonian cycles* in graphs. Paraphrased slightly, the problem is to evaluate the binary relation that pairs each graph to all —if any— of its Hamiltonian cycles. The domain of *TSP* is the set of all graphs. Since Hamilton cycles are themselves graphs, this is also the codomain of the relation. Any particular simple graph is an *instance* of *TSP*. If G is such an instance, any graph that is a Hamilton cycle in G is a *solution* of G for *TSP*.

Algorithms

Intuitively, our reasons for equating problems to binary relations become clear when we try to find a solution to a particular problem instance. It is one thing to write down the definition of *TSP*, but to find an actual Hamilton cycle in an actual graph is quite another. Computer science is largely concerned with the development of efficient procedures that can find solutions to problem instances within finite time and with absolute certainty. Informally, these procedures are what we refer to when we speak of *algorithms*. While a formal definition of the concept 'algorithm' is nearly as evasive as that of the concept 'problem', we know of several widely accepted models of computation that can be used for this purpose. For the sake of this discussion, we will adopt the formalism of *partial recursive functions* [37]. We thus define an *algorithm* for a problem P as a *partial recursive function* f which exhibits the following properties:

- $f \subseteq P$.
- $f(x)$ is defined for all x where there exists at least one y such that $(x, y) \in P$.

If f is an algorithm for P then f is said to *solve* P. It follows from our definition that different algorithms for P may yield different solutions for particular instances. We know from complexity theory that different algorithms for P may differ in terms of runtime efficiency. Additionally, an algorithm may or may not be suitable for (partial) parallel execution. Last but not least, we know from computability theory that there are P for which no algorithms exist. For an introduction to partial recursive functions, computability theory and complexity analysis, we refer to [37].

If we consider the set of all problems and the set of all partial recursive functions, the

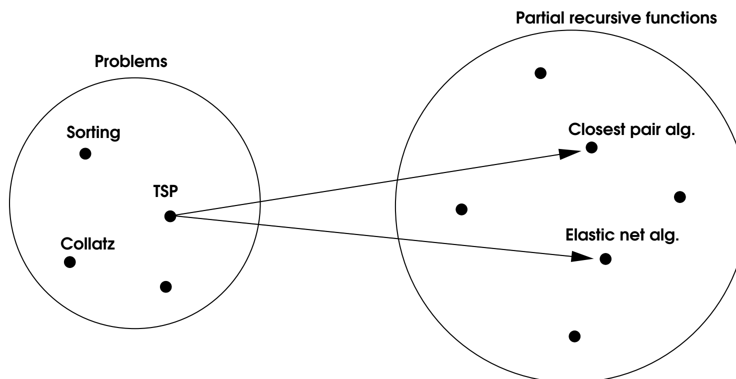

Figure 1.2: The *ALG* problem.

binary relation *ALG* that pairs each problem to the partial recursive functions which are its algorithms is itself a problem (see Figure 1.2). In fact, we would like to pose that computer science is the discipline of finding and studying solutions to the instances of the *ALG* problem. Since the *ALG* problem is not algorithmically solvable, we need to resort to less deterministic techniques in order to tackle it. *ALG* is thus a *design problem*. While it is not the central subject of our investigations, the *ALG* problem is relevant to us, as its solutions define the domain of another important problem. That problem, which is discussed in the next paragraph, provides the essential motivation for our interest in reconfigurable computing architectures.

Implementations

When faced with an instance of a problem, an algorithm is of course only half of the tool that is needed to systematically find solutions. We also require a physical artifact which can *perform* the algorithm. To be more precise, we require an artifact that is capable of preserving the input/output relation of the algorithm, in addition to its theoretical performance characteristics. We call such an artifact an *implementation* of the algorithm. Implementations can take diverse physical forms. As an example, we consider an arbitrary algorithm f which solves *TSP*. A stored program computer which is preloaded with an appropriate sequence of instructions could be considered an implementation of f. Alternatively, a properly trained mathematician behind a desk, or a trained monkey for that matter, could also be considered an implementation of f. Neither implementation would be more efficient in terms of time complexity, but under normal circumstances the computer would likely have a much shorter turnaround time, due to the number of algorithmic steps it can perform per unit of time.

The relation between algorithms and their implementations is of course a problem, as per our earlier definition of the word. This problem, which we will call the *IMPL* problem for the sake of reference, is depicted in Figure 1.3. It drives the discipline of computer engineering, just as the *ALG* problem drives the discipline of computer science. As theorists strive to find the most efficient algorithms for computing problems, engineers do their part by seeking to create the fastest implementations of algorithms[2]. Like the *ALG* problem, *IMPL* is a design problem, as we have no repeatable procedures for creating the best possible physical implementation of a given algorithm.

Together, the disciplines of computer science and computer engineering attack the com-

[2]In reality, the perimeter between the two disciplines is less clear. The design and implementation of programming languages and their compilers, for example, is the traditional domain of computer scientists.

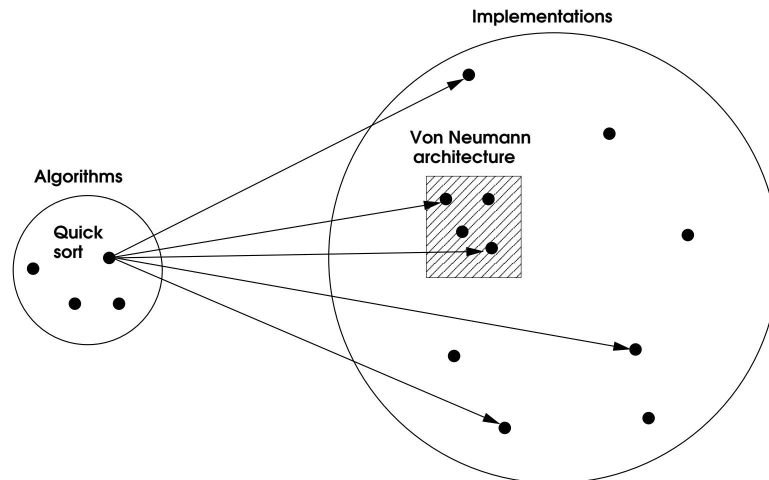

Figure 1.3: The *IMPL* problem.

bined problem of finding problem-solving automata which exhibit the shortest possible turnaround times for a given problem. In the next paragraph, we will consider some of the practical limitations that inhibit the development of optimal problem-solving automata. Reconfigurable computing systems will be addressed later on as opportunities for relaxing some of these limitations.

Adaptable computing systems

The synthesis of physical computing artifacts is a resource intensive process. For this reason, computer engineering has naturally gravitated towards systems which are adaptable to new tasks. The most successful attempts at such adaptable systems have been based on the sequential stored program computer model, as proposed by Von Neumann. The stored program computer offers a small virtual design space in which engineers have the freedom to create ‘soft’ implementations of algorithms with relative ease.

While the Von Neumann architecture can be adapted to perform any algorithm, it exposes its reconfigurability in a rather primitive manner. In order to implement an algorithm using such an architecture, i.e. to solve the *IMPL* problem, engineers need to express the algorithms in terms of the rather small set of problems which the hardware can actually solve efficiently, i.e. those corresponding to the instructions in its instruction set. The penalties of this intellectual detour, both on implementation time and on performance of the resulting systems, can be quite dramatic.

While it has remained the de facto standard in computing, the downsides of the Von Neumann architecture have long been acknowledged. Gerald Estrin and John Pasta first addressed the performance problems of the architecture in 1959. They started the first research program into reconfigurable computing architectures at the University of California at Los Angeles (UCLA) one year later. The direct incentive for their effort was the growing concern about computational problems which could not be solved efficiently using Von Neumann computers on the one hand, and the industry’s failure to produce radical alternatives on the other [21]. John Backus was probably the first to address the detrimental effects of the Von Neumann architecture on our reasoning about programs, in his now classic 1977 Turing Award lecture [8]. In his own words:

> Not only is [the system bus] a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied

> to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand.

Despite the wide support for his position at the time, and despite the development of elegant declarative programming paradigms since, word-at-a-time programming has hardly been dismissed from our engineering practices. An important reason for its persistence is the fact that, underneath it all, our computing hardware is still instruction based. While declarative languages may provide much better formalisms for reasoning about algorithms, imperative programming languages tend to map better onto sequential logic. With the recent arrival of cheap, reconfigurable VLSI technology —hardware which is not inherently instruction-based— word-at-a-time programming at least deserves another round of thorough questioning.

1.4 Assessing reconfigurable computing platforms

In the foregoing, we briefly restated the goals of computer science and computer engineering at large. We pointed out the fundamental barriers that prevent engineers from further approaching these goals through established means and we hinted at the possibility of attacking these barriers by means of newly emergent, reconfigurable hardware. We will now define of the concept *reconfigurable computing system* and state the design criteria for such systems.

Definition 1.4.1 (Reconfigurable computing system). For the remainder of this text, we define *reconfigurable computing systems*, in the narrow sense, as computing systems that allow explicit rerouting of low level hardware signals. In a broader sense, we define *reconfigurable computing systems* as those macroscopic computing devices which are at least partly comprised of such reconfigurable hardware. We consider the tools and programming languages that allow for the development of applications for such architectures as part of the system in this broader definition.

Design objectives

In our review in Section 1.3, we established that the idealized goal of computer engineering is to create physical implementations of algorithms which exhibit minimal effective turnaround time behaviour. Reconfigurable computing systems are, above all else, tools that should enhance the effectiveness of engineers at attaining this goal. In order to be useful as such, we feel that RC systems should aim to satisfy the following idealized design criteria:

Design criterion 1 (Admittance). A reconfigurable computing system should *admit* efficient hardware utilization for arbitrary algorithms.

In essence, any prefabricated computing platform imposes constraints on the options of engineers. The admittance criterion states that reconfigurable computing systems should minimize these constraints, in order to allow the user to create customized, efficient designs for arbitrary algorithms.

Design criterion 2 (Accessibility). A reconfigurable computing system, in the broad sense of definition 1.4.1, should provide an *accessible* development platform.

Whereas the admittance criterion only states that it should be possible to create efficient implementations of algorithms in theory, the accessibility criterion demands that there should also be practical means of developing such implementations, i.e. at the expense of reasonable amounts of intellectual effort, time and money.

Design criterion 3 (Transparency). A development platform for reconfigurable computing applications should be *transparent* to the developer.

In order to improve accessibility, developers generally rely upon tools that allow for algorithms to be expressed in abstract formalisms, rather than low-level detail. The transparency criterion demands that these tools should not introduce performance surprises, i.e. that the developer should be able to make reasonable estimations of the runtime behavior of their program during the development phase.

Chapter 2

Reconfigurable architectures

In the first of the three complementary views on reconfigurable computing presented in this work, we will focus exclusively on the hardware organization of RC platforms. Under the assumption that the reader is already familiar with traditional hardware organization, i.e. the Von Neumann architecture, we will limit ourselves to a treatment of what are, in our opinion, the key aspects of hardware organization in RC systems. In Section 2.1, we present these key aspects and provide a short explanation. In Section 2.2, we will examine a number of actual hardware designs, in order to gain a feel for what these aspects mean in a practical sense. Section 2.3 finishes this chapter with a conclusion of our findings.

2.1 System properties

Hardware platforms for reconfigurable computing systems come in various shapes and sizes. On a broad scale, we can observe four distinctive properties which can be adopted as a means of classification. In this section, we will present and discuss these characteristics one by one.

2.1.1 Reconfigurability

The definition of RC systems in 1.4.1 allows for platforms which combine traditional fixed components with reconfigurable computing fabrics. The balance between these two types of logic circuits typifies a hardware platform. We will refer to this property as the *reconfigurability* of a system. Reconfigurability is a macroscopic system property.

Reconfigurability can be viewed as a continuous scale. On the extreme end of the scale are systems which are fully adaptable, i.e. systems which are completely comprised of reconfigurable fabrics. On the other extreme end are non-reconfigurable architectures, such as traditional Von Neumann computers. Somewhere in between these extremes are 'fixed plus variable' systems, which combine fixed components, such as conventional sequential processors or finer grained components, with reconfigurable routing- or computational fabrics.

2.1.2 Coupling

Given the distinction between fully and partially reconfigurable architectures, we can make further distinctions amongst systems of the latter type, based on the nature of the communication facilities which exist between the reconfigurable fabric and the fixed core. We will make a broad distinction between two types of systems. On *loosely coupled* systems, the reconfigurable fabric executes computations with a certain degree of autonomy, i.e. as a

small parallel task. On *tightly coupled systems*, the reconfigurable fabric functions as an integral component of the CPU, i.e. as a special extension to the ALU on which custom instructions can be implemented.

2.1.3 Granularity

Up to this point, we have presented two characteristics which describe systems on a macroscopic level. The final pair of characteristics will deal with the microscopic level of RC systems: the reconfigurable fabric. The first and arguably the foremost characteristic of this fabric is its *granularity*. The concept of granularity deals with the 'size' of the logic elements that can explicitly be reconfigured by the programmer. On the extremely fine-grained end of the spectrum, we have fabrics that allow explicit reconfiguration of single logic gates. Moving towards the coarser end of the spectrum, we encounter systems which offer control over larger operational components, e.g. adders, multipliers, shift registers, etc. RC fabrics can have homogeneous granularity, but hybrid fabrics are also common.

Granularity has a number of important implications. Foremost, it has an obvious bounding effect on the admittance of the system as an application development platform. Fine-grained fabrics allow for the implementation of detailed, problem specific designs, which implies a higher potential for efficient hardware utilization. There is a flip side to this coin however, as coarser grained components can be optimized for performance on a level deeper than that of logic gates. Through clever design tricks, for example, coarse specialized components can be made which exhibit shorter propagation delays than logically equivalent implementations with reconfigurable logic gates. For this reason, we see a trend towards heterogeneous fabrics, which offer a mixture of reconfigurable gates and commonly used arithmetic- and routing components.

2.1.4 Reconfiguration time

The fourth and final systems characteristic is also related to the reconfigurable fabric on a microscopic scale. The *reconfiguration time* is the amount of time that is required to load a new circuit description into the fabric. In our definition of reconfiguration time, we exclude any compilation- or preprocessing time, i.e. we are only interested in the time it takes to physically change the hardware to a new configuration.

In relation to reconfiguration time, we make a broad distinction between two classes of systems. On the one hand we have systems which are *statically reconfigurable*. A statically reconfigurable fabric is a fabric that cannot change its configuration at runtime. Conversely, *runtime reconfigurable* fabrics can be loaded with new configurations during the execution of a program. Runtime reconfigurability is a useful property, since it can help to alleviate the problem of limited hardware resources. With the gained power, of course, comes increased management complexity.

2.2 Architectures

In the foregoing, we introduced four aspects of hardware organization, uniquely pertaining to reconfigurable computing architectures. We will now direct our attention to the study of a number of actual hardware designs. The designs that are discussed in this section represent a wide variety of different approaches to RC hardware design.

2.2.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) are reconfigurable VLSI components that allow for the implementation of arbitrary sequential and combinatorial circuits. In its simplest form, an FPGA consists of an array of configurable logic blocks (CLB), a fixed set of I/O blocks and a flexible routing mesh. An FPGA 'program' consists of a functional specification for each of the CLBs and a connectivity description for the routing mesh. FPGAs are widely used in the capacity of application-specific integrated circuits (ASIC). Example applications include realtime digital signal processing and data encryption.

Reconfigurability and coupling

An FPGA is a single VLSI component. In order to be suitable as a generic computing platform, it needs to be mounted on a printed circuit board that provides I/O facilities. A number of inexpensive FPGA development boards are currently available on the market, which commonly provide a selection of IO connectors —ranging from PS2 ports, VGA ports and serial (RS232) ports to Ethernet connectors and beyond— which are connected directly to the IO blocks of the FPGA. While these boards are designed with development for embedded applications in mind, they are suitable for use as simple reconfigurable computing platforms. An FPGA mounted on such a board is a prime example of what we consider to be a *fully reconfigurable* architecture, i.e. any subsystem decomposition or internal structuring is completely up to the application developer. As an aside, we mention that some vendors are currently offering partially reconfigurable systems which combine fixed cores and FPGAs on a single die.

Granularity

The configurable logic blocks (CLB) on an FPGA are fine-grained components. A CLB can typically be used to replace two to six simple logic gates or a single flip flop in a gate level circuit design. In addition to these generic CLBs, most of the commercially available FPGAs contain larger, coarse-grained blocks which provide commonly used functionality. As an example, the members of the Xilinx Spartan-3 family of FPGAs offer 18-kilobit synchronized RAM blocks, 18-bit multiplier blocks and a small number of specialized clock management blocks in addition to their generic CLBs [46]. Other vendors offer similar packages. FPGAs can thus be considered *fine-grained*, *heterogeneous* computing platforms, as per our discussion in 2.1.3.

Reconfiguration time

The reconfiguration times of FPGAs are model specific. Traditionally, FPGAs required a complete reset from an external bit stream which took several seconds to perform, making them *statically reconfigurable* only. At present, some FPGA models support limited runtime reconfiguration, e.g. the Xilinx Virtex-4 family [47].

2.2.2 Garp

Garp [15] is the reconfigurable computing architecture from the University of California at Berkeley. The Garp architecture consists of a fixed MIPS [4] core, augmented with a reconfigurable array. The reconfigurable array was designed with the acceleration of loops in general purpose programs in mind.

The anatomy of Garp's reconfigurable array is similar to that of FPGAs, i.e. it is comprised of reconfigurable logic blocks and a flexible routing mesh. These logic blocks array can perform logical and arithmetic operations on operands which are at most two bits in

size. More complex operational components can be composed of these smaller units by means of the routing mesh. The configuration of the reconfigurable array is controlled by the program that runs on the sequential core. From the programmer's point of view, the array is available as a support utility, which can be deployed for demanding computations.

Reconfigurability

The Garp architecture is a fixed-plus-variable architecture. In the case of Garp, the motivation for this configuration stems from the observation that most sequential algorithms spend the bulk of their time in relatively small pieces of looping code. Instead of applying complex custom computing approaches to complete programs, the Garp architecture focuses on the acceleration of these loops by means of the reconfigurable array, and leaves the mundane algorithmic tasks to the conventional sequential logic. The Garp architecture is thus a *partially reconfigurable* platform.

Coupling

The coupling of the Garp system is similar to that of a conventional sequential core with an assisting coprocessor. The VLSI core, which contains the MIPS CPU, connects to the reconfigurable array over high bandwidth connection with a moderately high latency, i.e. while high volume data transmission is possible, the latency is too high to communicate data on a per-instruction basis. Data is moved to and from the array using special move instructions, which take an unspecified but small number of cycles to execute.The array communicates its status to the core through an interrupt mechanism, allowing the core to continue parallel execution during computations on the array.

Both the MIPS core and the reconfigurable array have direct access to the main memory subsystem. The MIPS core has a conventional instruction cache. The reconfigurable array has a configuration cache, that allows the array to recall recently used configurations.

Granularity

The reconfigurable array of the Garp architecture is a *fine-grained*, *homogeneous* reconfigurable fabric. As mentioned in the introduction, the configurable logic blocks of the fabric can be used to implement operations with at most two bits of input. Similar to FPGAs, the CLBs of the array can double as small latches or registers. Contrary to many current FPGAs, the array contains no coarse-grained operational units.

Reconfiguration time

The Garp architecture was designed with the intent to optimize small loops in conventional programs. In order for this approach to be feasible, reconfiguration times had to be kept acceptably low. In order to achieve this, the reconfigurable array of the architecture has a high bandwidth connection with main memory and a private configuration cache. The size of the configuration cache is model-specific, but it is intended to hold at least several, e.g. about four, complete configurations or many smaller ones. The array can be reconfigured from the cache in approximately five clock cycles.

2.2.3 PipeRench

PipeRench [25] is Carnegie Mellon University's answer to the reconfigurable computing challenge. The system, which is particularly suitable for stream based media processing applications, was designed with the benefit of hindsight on FPGA-based reconfigurable

computing machines. Specifically, the PipeRench fabric aims to beat FPGA based machines on five counts, three of which are discussed here:

Forward compatibility Configuration bit streams for FPGAs are model-specific. There is no portable binary format for FPGA core images, which necessitates recompilation for models with different gate capacities. The PipeRench executable format was designed to be forward compatible with larger and faster implementations of the architecture.

Resource constraints FPGA developers are often confronted with the limited gate capacity of the hardware. The PipeRench system encompasses an innovative 'hardware virtualization' technique, which is used to abstract away from the physical limitations of the hardware, allowing transparent reuse of the available processing components.

Compilation time Compared to the compilation times of sequential programs, the build times of FPGA-based applications are relatively long. The hardware virtualization mechanism of PipeRench has the side effect of relaxing the difficulties of compilation, because the compiler no longer has to optimize for efficient space usage.

The other two claimed improvements of PipeRench are improvements in *configuration time* and *logic granularity*. For the sake of consistency with our treatment of other platforms, we discuss these aspects in the appropriate sections below.

Reconfigurability

Technically, the PipeRench design is a pure reconfigurable fabric. It is thus in itself a fully adaptable system. Unlike FPGAs however, the PipeRench fabric is not suitable for complete system on chip implementations. The designers of the system envisioned PipeRench as a coprocessor, i.e. it requires an additional computing system in order to be used as a full computing platform. A complete PipeRench-enhanced computing system should thus be classified as a *partially reconfigurable* system.

Coupling

As mentioned, PipeRench was designed to function as an attached coprocessor in a general purpose computing system. The coprocessor is intended to be used as a *loosely coupled* system, which processes longer instruction sequences with relative autonomy.

Granularity

One of the key design points of the PipeRench fabric is its radically different approach to granularity compared to FPGAs. According to its designers, a key improvement of PipeRench over FPGAs is that its configurable logic units are designed for computation purposes, whereas FPGAs are designed for the more difficult task of random logic replacement. The PipeRench fabric is thus an example of a relatively coarse-grained fabric.

We will now look into the fabric of PipeRench in some detail. The PipeRench chip is comprised of so called *stripes* which represent pipeline stages in a computation. One stripe contains a number of configurable processing elements, which need to be thought of as ALUs in terms of granularity. All processing elements inside a physical stripe need to be configured simultaneously, i.e. in the designer's terminology: the elements in a stripe maintain their static function while a virtual stripe resides in a physical stripe.

Subsequent stripes in the PipeRench fabric are connected by an interconnection network. The processing elements from a stripe can access the computed values of any processing

element in the preceding stripe, or registered and unregistered values from elements in the same stripe, through this network. Finally, each processing element contains a set of pass registers which can be read by the subsequent stripe. If the processing element doesn't provide a value for a pass register, the value is copied from the preceding process element's corresponding register.

Reconfiguration time

A powerful feature of the PipeRench fabric is its extremely short configuration time. A physical stripe on the fabric can be reconfigured in a single clock cycle. Its short reconfiguration time enables the PipeRench architecture to simulate long virtual pipelines on hardware with fewer physical stripes with minimal overhead.

2.2.4 Raw

The Reconfigurable Architecture Workstation ("Raw") microprocessor is an experimental reconfigurable fabric from Massachusetts Institute of Technology [44], which was designed for general purpose computing. The Raw system was built on the observation that on-chip wire delays are becoming the limiting factor in modern CPU designs, to an extent where they can and should no longer be abstracted away from the developer. The proposed solution is an architecture which explicitly exposes wire delays to the programmer.

The Raw processor consists of 16 identical programmable tiles on a single silicon die. Each of these tiles contains a single-issue MIPS core, a floating point unit, a data cache and a software managed instruction cache. The size of the tiles is chosen such that a logic signal can propagate an entire tile within one clock cycle.

The reconfigurable nature of the Raw architecture exposes itself through the routing mesh that connects the tiles, which is explicitly programmable. To the developer, communication between two adjacent tiles is visible as a network hop. The programmer has explicit control over the organization of the computation, in order to minimize the total amount of hops. The Raw architecture can thus be viewed as a symmetric multiprocessor on a single chip, with a partially reconfigurable communication grid.

Reconfigurability

In concept, the Raw architecture is more similar to a conventional multiprocessing cluster than it is to a reconfigurable computer. Contrary to other systems, the reconfigurable subsystem of the Raw architecture is a communications subsystem only, i.e. it does not serve computation explicitly. Given the granularity of the tiles, it is reasonable to classify Raw as a partially reconfigurable system, with the connotation that it approaches the limits of what can still be called a reconfigurable computer.

Coupling

In relation to other partially reconfigurable systems, we have looked at coupling in terms of the communication facilities between fixed and variable subsystems. On the Raw architecture, this view becomes distorted, as the communication facilities *are* the reconfigurable subsystem.

The reconfigurable network of the Raw architecture consists of a combination of four separate networks. In each of these networks, a tile connects to its four immediate neighbors through 32-bit, full duplex connections. Two of these four networks have to be configured by the application programmer at compile time. The remaining two networks can be reconfigured at runtime.

Granularity

The Raw architecture separates itself from its peers by taking a completely different approach to reconfigurability. In other systems, signals can be routed between relatively low-level operational blocks. The Raw architecture offers no such blocks, or rather, the blocks are so extremely coarse-grained that they are in fact completely autonomous sequential processors.

Reconfiguration time

As mentioned under the coupling heading, the reconfigurable network of the Raw architecture has variable reconfiguration time characteristics. Two of the four communication grids cannot be reconfigured at runtime, whereas the other two can. The exact number of cycles necessary for rerouting is not specified. Given the strong focus of the system towards single cycle operations, it would be likely that the routing can be reconfigured in a single MIPS instruction, i.e. one cycle.

2.2.5 Chimaera

The Chimaera Reconfigurable Functional Unit [29] is a general purpose reconfigurable computing architecture. Contrary to the other systems reviewed here, Chimaera combines a classic fixed architecture with a reconfigurable fabric on a single die. According to its designers, multiple die approaches suffer from a communication bottleneck between the fixed- and variable logic components, necessitating the transport of larger units of code to the reconfigurable fabric to maintain computational efficiency.

Reconfigurability

Chimaera is a fixed-plus-variable architecture. The design specifies a small reconfigurable array, which attaches to a conventional sequential host processor.

Coupling

The Chimaera design is an example of a tightly coupled system. The RFU communicates with the sequential logic on a per-instruction basis. From a programmer's perspective, RFU operations should be viewed as special instructions, rather than separate miniature tasks, as was the case with the Garp- and PipeRench designs.

At the heart of Chimaera's communication infrastructure lies the shadow register file. It is a collection of registers which mirror the content of a subset of the host processor's registers. The shadow registers connect directly to the reconfigurable array. They serve as inputs for RFU-based computations only. The output of an RFU operation is written directly to a single host register.

Granularity

Chimaera's reconfigurable array is a *homogeneous*, *fine-grained* reconfigurable fabric. Its design was inspired by that of the Triptych [13]- and Altera FLEX 8000 [5] FPGA fabrics.

Reconfiguration time

The Chimaera system can be reconfigured at runtime. In addition to the mentioned shadow register file and the reconfigurable array, the RFU contains a configuration cache- and prefetch unit, which is used to compensate for costly RFU misses. The reconfigurable

array can contain an unspecified number of RFU operations simultaneously, depending on its capacity.

2.2.6 Other architectures

In this section, we have looked at a small subset of the architectures that are currently being proposed. While these architectures have been chosen to reflect some of the bigger trends in RC hardware design, there are many interesting variations on these themes, which are equally worthy of investigation. Rather than citing an arbitrarily long list of names, we advise the interested reader to consult the concise list of reconfigurable architectures maintained by Rami Abielmona [2], and the list of FPGA-based computing machines [3], maintained by Dr. S. Guccione of Xilinx, Inc.

2.3 Conclusion

In this chapter we presented four system properties that can be used to classify reconfigurable computing architectures. In a series of case studies, we applied this simple model to a number of actual architectures. A summary of our findings is given in Table 2.3, and in the discussion that follows.

Platform	Reconfigurability	Coupling	Granularity	Reconfiguration time
FPGA	Full	N/A	Fine/Heterogeneous	Model specific
Garp	Partial	Loose	Fine	Runtime
PipeRench	Partial	Loose	Moderately coarse	Runtime
Raw	Routing only	Tight	Coarse	Partially runtime
Chimaera	Partial	Tight	Fine	Runtime

Table 2.1: Hardware platform summary

Reconfigurability

We have considered systems with varying levels of reconfigurability. Advancing on our discussion on the difficulties of code generation for reconfigurable architectures, we conclude that a partially reconfigurable setup, where the fixed logic handles mundane computational tasks while the reconfigurable fabric is used for the implementation of demanding 'micro algorithms', holds a great advantage over fully reconfigurable systems in terms of developer accessibility.

Coupling

We have considered 'fixed plus variable' architectures with different degrees of coupling between the sequential logic and the reconfigurable fabric. We have seen that tightly coupled systems, i.e. systems where communication between subsystems occurs on a per-instruction basis, are easier to program than loosely coupled systems, where the reconfigurable fabric executes its tasks with a certain degree of autonomy. We have also seen that tight coupling inhibits the exploitation of parallelism on the reconfigurable fabric, due to the fact that the fabric is forced to run in lock step with the fixed sequential logic.

Granularity

We have considered reconfigurable architectures of different granularities. Amongst these, FPGAs represent the finer end of the spectrum, whereas the Raw architecture is a repre-

sentative of a more coarse grained approach. Fine grained approaches have the advantage of allowing more detailed control, and thus a more efficient utilization of resources than coarser grained systems. A disadvantage of such systems is that complex operational circuits (numeric adders, multipliers, etc.) tend to exhibit longer propagation delays when implemented in fine grained hardware. Heterogeneous systems, e.g. systems that offer both fine grained logic and a selection of larger components offer a satisfactory compromise. The extremely coarse grained approach of the Raw architecture, where each logic block is a complete sequential core, offers few advantages over conventional parallel systems.

Reconfiguration time

We have considered both statically and dynamically (runtime) reconfigurable fabrics in this chapter. We have seen that dynamic reconfigurability is not only a desirable property, but that it is the only effective way of dealing with larger programs on hardware of limited capacity.

Chapter 3

Synthesis tools

Reconfigurable computing platforms offer an unprecedented amount of freedom to the programmer in comparison to Von Neumann machines. While this freedom brings opportunity, it also demands greater care and effort during development. It is the function of the *synthesis tool* to offer abstraction from the low level details of the hardware, in order to make the implementation of algorithms a tractable problem for the human programmer. Relying on the synthesis tool to offer abstraction from the complexity of the underlying platform does not, however, solve the problem of exploiting the added flexibility. It merely shifts the burden from the application developer to the synthesis tool designer.

In this chapter, we will look at the unique difficulties of developing compiler technology for reconfigurable targets. In Section 3.1, we discuss the problem with an emphasis on programming languages and paradigms, i.e. the we discuss synthesis tools from the 'user' perspective. In Section 3.2, we take a target oriented look and discuss the models of computation that synthesis tools use as intermediate targets. In Section 3.3, we will evaluate the practical implications of language design and computation models by looking at details of some existing synthesis tool designs.

3.1 Language considerations

A synthesis tool should help developers to transform their thoughts about algorithms into efficient implementations. The first task of the tool is thus to capture the thoughts of the developer in an accurate way. The source language is the vehicle for this influx of information. If anything, it should be expressive and powerful enough to allow the developer to express their thoughts about algorithms in a straightforward manner.

Source language design is, however, not completely user-centric. On the other end of the synthesis tool, we find the reconfigurable target architecture, with its structural semantics. At some point during the synthesis process, behavior will have to be mapped onto structure. This task —the bridging of the 'semantic gap'— is a monumental task; one that may prove to be too heavy for application developers or synthesis tool designers to bear on their own. Conventional wisdom then suggests that the burden may be spread over both parties. This desire for intellectual load balancing counteracts the wish for a fully abstract, developer friendly source language. In the design of a source language for a synthesis tool, the constraints of the underlying hardware should thus be taken into consideration.

Apart from the usual cosmetic issues, source language design revolves around two central issues: the declarative vs. imperative programming debate and the tradeoff between structural and behavioral modeling. The former of these is a known issue from conventional

compiler technology, but one that may have a different outcome with respect to reconfigurable architectures. The latter is a tradeoff that is unique to reconfigurable computing. We will now discuss them both in some detail.

3.1.1 Structural vs. behavioral modeling

At the lowest level, reconfigurable computing architectures distinguish themselves from their sequential counterparts through exposure of their internal structure. Effectively, this means that the programmer is expected to describe the signal interconnections between the available logic components. Low level reconfigurable programming thus deals with structural organization of logic elements, rather than the functional description of algorithms. Reconfigurable platforms can be programmed directly at this level, using a rudimentary language in which structure can be expressed. In this scenario, the burden of crossing the semantic gap is completely on the application developer. We refer to this style of modeling as *structural modeling*.

One of the desirable properties of a synthesis tool is that it can abstract away from the structural nature of reconfigurable computing. If so desired, we could develop languages in which the programmer exclusively deals with functional, or even imperative[1] descriptions. This style of programming, which is similar to that of conventional programming languages, is referred to as *behavioral modeling*. A purely behavioral modeling language requires an intelligent tool, which is capable of negotiating the semantic gap automatically.

As we have mentioned earlier, application development and synthesis tool design are both complex tasks with conflicting interests. This dissonance is without doubt the single most important problem in reconfigurable computing today. With little overstatement, the holy grail of reconfigurable computing is to create a synthesis tool which combines a fully behavioral source language with code generation capabilities that match or surpass handcrafted structural designs in quality. As is typical of holy grails, such a tool has remained conspicuously elusive.

3.1.2 Declarative vs. impertative programming

As synthesis technology attempts to move toward the ideal of behavioral modeling, the decade-old question 'which behavioral programming paradigm is best?' towers at the horizon. At the risk of reiterating one of the most heated debates in computer science, we will argue here that there are substantial arguments to reevaluate the results of this discussion in light of reconfigurable computing systems.

To put our discussion into perspective, we recall that the word *paradigm*, in general, denotes a set of *beliefs* or assumptions about the world which is shared by a group of people. In programming, for example, the Object Oriented paradigm (OO) encompasses the belief that all properly structured programs are internally composed out of autonomous units which communicate with each other through implementation-independent interfaces. When a programmer shares this belief with the compiler designer[2], the source language may reflect this common assumption by offering specialized support for object creation and inter-object communication. A paradigm in computer programming is thus a set of assumptions about the nature of programming, which are accepted as truths by the language designer and the programmer alike.

As is customary with belief systems, some paradigms in programming are based on mutually conflicting assumptions. The canonical example of such a near-religious dispute in

[1] Note, for example, the sequential programming model provided in the VHDL 'process' environment.

[2] Apparently, not all programmers do.

computing is the ‘declarative vs. imperative’ debate, which has kept academics and engineers quarreling for nearly half a century. The declarative paradigm assumes that the programmer’s position is to tell the computer ‘what’ it is that is to be computed, independent of the ‘how’ of the computation. The imperative paradigm assumes almost the opposite: the imperative programmer conveys the ‘how’, leaving the ‘what’ fully implied. In imperative programming, the position of the programmer is to reduce the computational task to a series of trivial steps. In declarative programming, the objective is to reduce the problem to a series of elementary declarative statements that can be resolved by the runtime environment or the compiler.

To an extent, a stalemate has been reached in the declarative vs. imperative debate, leaving the conclusion that declarative programs are generally shorter to write and less prone to errors, whilst imperative programs are considerably easier to compile into efficient code for Von Neumann target architectures. The latter argument, then, is the exact reason why we should be wary of imperative languages with respect to reconfigurable architectures: on non-sequential hardware, the main argument for their justification simply doesn’t apply.

3.2 Models of computation

In Section 2, we painted the field of reconfigurable hardware platforms as a wide and varied landscape, which up to this point has displayed few signs of convergence. With this image freshly in mind, it is evident that a generalized description of the *output* of synthesis tools is hardly attainable. Matters are complicated further by the fact that synthesis tools are typically located at the top of a larger tool stack, in which each subsequent layer represents a virtual target platform in itself; each with its own distinct semantics. In order to understand the transition from behavioral- to structural description, it is nonetheless important to have an general impression of what it is that synthesis tools produce as output.

With the connotation that a complete coverage is beyond the scope of this text, we will take an alternative approach to describing what it is that synthesis tools generate. We will present a number of abstract models of computation, which share the property that they can easily be translated into a hardware specific structural description. This approach will allow us to focus on the transition of the semantic gap, whilst avoiding any platform specific details.

We will consider the following models of computation:

- The *gate level description*.
- The *data-flow network*.
- *Communicating sequential processes* (CSP).
- The *custom instruction set architecture*.

These models provide a good general idea of what the output of many actual synthesis tools looks like.

3.2.1 Gate level description

In its idealized form, a reconfigurable computing platform allows for explicit reprogramming of individual logic gates and their interconnections, without resource constraints and performance penalties. The first of our four target architecture models is a direct reflection of this idealized flexibility: the *gate level description*. A gate level description synthesizer translates its source program into a directed, single edged graph, which directly describes

elementary logic gates, IO ports, and their interconnections, which collectively implement the desired functionality. In a gate level description, clock signals are explicitly visible, in the form of ordinary component input signals. Similarly, any latching functionality has to be synthesized explicitly.

The gate level description is a model of computation that can be mapped onto fine-grained reconfigurable hardware with relative ease. It should be noted, however, that a series of non-trivial transformations remain before a hardware configuration bit stream can be generated from such a description[3]. The gate level description is a commonly used stepping stone in synthesis tool stacks for FPGA and CPLD targets.

3.2.2 Data flow network

In our discussion of hardware architectures, we have seen a distinction between fine-grained and coarse-grained approaches. We expect to see similar distinctions in abstract synthesis targets. The *data flow network*, then, can be viewed as a coarse-grained variation on the gate level description. Data flow networks are directed acyclic graphs of coarse-grained operational components (e.g. adders, multipliers, etc.), which operate in lock step according to a global clock. Latching is implicit at each step, as the connections between successive operators are expected to be registered.

The choice of the data flow network as a model of computation has a number of pros and cons as compared to the finer grained gate level description. On the upside, this approach simplifies synthesis considerably, due both to the fact that the number of vertices in a data flow graph is an order of magnitude smaller than that of a functionally equivalent gate level description, and the fact that no bit-level synthesis has to be performed[4]. The inevitable downside is that no bit level synthesis *can* be performed either, thus inhibiting effective bit-level parallelism. Another drawback is the fact that data flow networks have difficulty with maintaining state over longer periods of time, as data is forced through the acyclic graph at each step.

The data flow network is a natural target for synthesis on coarse-grained RC platforms. Aside from its native habitat, the model can be used as an intermediate step in tool stacks for finer grained platforms. This latter approach is especially potent in combination with the augmented instruction set architecture approach, as discussed in Section 3.2.4.

3.2.3 Communicating sequential processes (CSP)

Gate level descriptions and data flow graphs are both examples of low level target architecture abstractions. In these models, the atomic units of the underlying hardware are exposed to the synthesis tool directly. By contrast, the *communicating sequential processes* model signifies a mid-level approach. In this model, which is loosely based on Hoare's formalism [30], the synthesis tool maps the input program onto a set of sequential processes, which are intended to operate concurrently. Communication between these processes occurs either synchronously or asynchronously through specified data connections.

The CSP model of computation is somewhat higher in level than the models we have discussed so far. In general, it will be necessary to have an additional strategy to transform the processes into actual structural designs. Depending on the complexity of the processes, one of two approaches can be taken:

[3] These transformations, which may include gate-to-CLB mappings and 'place-and-route' stages, are hardware specific. Some of them will be discussed during the subsequent case studies.

[4] ALU configurations can be defined statically.

Concurrent state machines Given the finite memory resources, sequential processes in CSP are effectively state machines. If the number of states in a process is small, e.g. in the order of tens, a generic state machine template can be used for code generation.

Hybrid modeling If the complexity of the processes does not facilitate a direct mapping onto simple state machines, i.e. when the number of possible states of a process becomes too high, due for example to the presence of data structures internal to the process, an additional model of computation can be used to describe the sequential task.

The CSP approach trades some of the expressive power of low level models for a simplification of the synthesis problem.

3.2.4 Custom instruction set architecture

The final model of computation discussed here has its roots firmly in traditional Von Neumann computing. The *custom instruction set architecture* model of computation is for the better part identical to that of the Von Neumann approach: the input program is transformed to a logically equivalent sequence of primitive operations. These operations form the *instruction set architecture* of the target. In reconfigurable computing, the compiler has a say in the design of this architecture, i.e. it can synthesize a custom instruction set in addition to the instruction sequence itself.

The custom instruction set architecture model assumes implicitly that bit-level parallelism is where reconfigurable computing will shine, as other forms of parallelism are effectively ignored. The best fit for this model are the hardware platforms which combine sequential logic with a closely coupled, i.e. low latency/high bandwidth connected, reconfigurable fabric. On platforms that allow greater flexibility, the assumption that all interesting opportunity for parallelism is exhibited at the bit-level is, at best, naive.

3.3 Synthesis tools

After the theoretical treatment of Sections 3.1 and 3.2, we will now examine synthesis tools from a more practical perspective. In this section, we will look at a selection of actual synthesis tool designs. Given our angle, we will not be interested in the finest of details. Rather, we will aim for general impressions that will allow us to gain an understanding of how language designs and models of computation affect synthesis tool design in practice.

3.3.1 XST

XST (Xilinx Synthesis Technology) is the entry-level, proprietary synthesis tool from Xilinx, Inc., a large manufacturer and developer of FPGA products. XST occupies the top layer in a stack of tools, the Integrated Synthesis Environment (ISE), which collectively transform developer input into a model-specific FPGA configuration bit stream. While our main focus is on the top level synthesis tool, we will discuss the subsequent layers of the ISE stack in some detail, in order to gain a holistic understanding of the synthesis process.

Source language

FPGA designs are commonly developed in so-called *hardware definition languages* (HDL). The HDL world is dominated by two languages: VHDL and Verilog. XST supports both. In this discussion, our focus will be on VHDL.

The VHSIC[5] Hardware Description Language —VHDL for short— grew out of the desire for a standardized description format for large scale digital systems, in the late 1970's. The development of VHDL, which was largely carried out by the U.S. Department of Defense, culminated in the adoption of an IEEE standard (IEEE 1076) in 1987, and subsequent revisions in 1993 and 2001. At its inception, synthesis of logic designs was not part of the task description of the language, i.e. it was used primarily for simulation and verification purposes. These roots are still visible in modern VHDL-based synthesis tools like XST, in the sense that not all valid VHDL code is synthesizable into hardware. Application developers are thus generally confined to a relatively small synthesizable subset of the language, which, unfortunately, is not standardized across different synthesis tools.

As a hardware description language, VHDL does little to hide the structural semantics of the underlying platform. A VHDL 'program' consists of an explicit external description of a single physical component, which represents the top level of the application. The input- and output signal 'pins' of this top level component may be connected to the physical IO facilities of the FPGA through a simple name resolution mechanism.

Once a component is defined from the outside (using an 'entity' construct), its inner workings are to be specified in the body of an accompanying 'architecture' block. The architecture block allows for a mixture of structural and behavioral statements. As an example of the former, smaller components can be instantiated and composed inside the architecture body to produce the larger component's functionality. Theoretically, algorithms can be expressed completely in structural terms, through successive decomposition in ever smaller components, requiring simple behavioral constructs only for the implementation of simple logic gates. In practice, however, behavioral constructs are used to accomplish slightly more high level tasks.

The behavioral constructs of VHDL deserve further elaboration. In its simplest form, a behavioral statement in an architecture body assigns a value to an output signal. These values can be computed from input- or internal signals or variables using simple arithmetic operations or elementary conditional constructs. Consider the following examples:

```
-- Simple signal assignment
foo <= 3;

-- Arithmetic on input signals
bar <= baz + quux;

-- Switch/case like construct
with quux select
     zow <= 7 when 0,
            1 when 1,
            8 when 2,
            2 when others;
```

Contrary to the imperative programmer's intuition, statements such as the above are executed concurrently, not sequentially. The output signals `bar` and `zow` are, for example, updated concurrently as the input signal `quux` changes value. Signal assignments such as these are examples of pure behavioral, declarative programming, as the synthesis tool is responsible for the generation of an appropriate circuit. The developer can be assured, however, that the generated logic for these statements will be combinatorial, not sequential.

In addition to this concurrent form, VHDL allows a second type of behavioral code, which is more akin to imperative programming: the 'process' construct. A *process* construct is a sequential statement block, which is executed each time one of the signals in its *sensitivity list* changes value. The body of a process can contain many of the familiar constructs from imperative programming, such as `if/else`-statements, loops and switches. While the programming model of the process construct is purely sequential, the synthesis tool is at

[5]Very High Speed Integrated Circuit

liberty to generate an equivalent combinatorial circuit, whenever possible. An example of process code is given below:

```
-- A driver for a blinking LED
foo: process( clock )
begin
    if ( rising_edge( clock ) ) then
        case state is
            when LED_ON =>
                state <= LED_OFF;
                led  <= '0';

            when LED_OFF =>
                state <= LED_ON;
                led <= '1';
        end case;
   end if;
end process foo;
```

A more thorough introduction to VHDL can be found in [7].

Model of computation

Given its position at the top of a larger tool stack, it is not surprising that the synthesis target of XST resembles one of our purely abstract models of computation rather closely: the gate level description. Specifically, XST generates NGC[6] netlists as output. A *netlist* is a description of physical components and their interconnections. The netlists that are output by XST contain only elementary logic components, such as gates and flip-flops, and possibly 'black box' components, such as adders and multipliers, if these are supported by the hardware. At this stage, little to no consideration is given to the actual configurable logic blocks of the target hardware, aside from the aforementioned high level components.

The transformation from a gate level description to a netlist of actual, model specific CLBs is the responsibility of the next layer in the ISE tool stack: the *mapper*. The mapper identifies and replaces small groups of gates or a single flip-flop with an equivalent CLB.

The next layer in the ISE stack is occupied by the *place and route* tool. This tool assigns each 'virtual' CLB to a physical unit on the FPGA surface, i.e. it performs *placement*. The configuration of the flexible routing mesh is generated simultaneously, in order to realize the required interconnections, whilst minimizing propagation delays. After the place-and-route phase, only one trivial transformation remains: the generation of a configuration bit stream for the FPGA.

In a VHDL program, parallelism is explicitly defined by the programmer at the statement level. The synthesis tool performs no special transformations to extract implicitly parallel constructs.

3.3.2 DEFACTO

The DEFACTO system —a Design Environment For Adaptive Computing TechnOlogy— is a synthesis tool that maps conventional high-level programming languages, in particular the C language, onto FPGA based computing platforms [20]. DEFACTO is partially funded by the Defense advanced Research Project Agency (DARPA) and Boeing Satellite Systems.

[6]NGC netlists are a proprietary format that resembles, and can be converted to, the standardized EDIF netlist format. Xilinx uses this format to facilitate better communication between XST and the tools in the lower layers of the ISE stack.

Source language

DEFACTO may be regarded as the canonical example of what is known in the literature as the 'dusty deck'[7] approach to reconfigurable computing: it is a direct attempt at generating FPGA bitstreams from legacy C code. From a language perspective, the programming model is thus purely behavioral and purely imperative.

Model of computation

The DEFACTO synthesizer generates code for a fixed-plus-variable architecture with a moderately close coupling between the FPGA fabric and the sequential host processor, e.g. a PCI-based FPGA coprocessor setup. After conventional code optimizations, the first phase in the code transformation process is to partition the application into a host part and an FPGA part. After this partitioning, the synthesis target for the FPGA-branch of the application is a behavioral VHDL program. DEFACTO can thus be viewed as a bridging synthesizer between C and VHDL. Its model of computation is not one of the models discussed in Section3.2. Rather, it is the mixed structural/behavioral model of VHDL. The actual transition from behavioral to structural design is thus carried out by the VHDL synthesis tool.

Since the VHDL target language is a language which contains explicit parallel constructs, the main task of the DEFACTO compiler is to extract the implicit parallelism from the C source code and to make it explicit. This is accomplished through a rather awkward procedure. After eliminating conditional statements[8], the suite employs 'heroic' compiler techniques to identify implicit loop parallelism. The sheer number of different techniques for this purpose that the authors describe is impressive, and somewhat worrying. After the loop analysis, a lengthy and unpredictable procedure determines the extent to which the loops are to be unrolled, to minimize runtime and optimize space utilization. This last procedure, which is dubbed *estimation* and/or *automatic design space exploration*, involves an iterative process of parameter tweaking, recompilation and automated benchmarking.

3.3.3 SA-C

The Single Assignment-C [38] compiler is the product of the Cameron project [18]; a collaborative research effort which aims to improve the accessibility of FPGA-based reconfigurable computing platforms. The Cameron project was started in May of 1999 at Colorado State University. It currently runs in collaboration with the University of California at Riverside and Khoral Research, Inc.

Before venturing into the details of the SA-C language and its model of computation, it deserves mention and praise that its designers have been amongst the few to illustrate the potential of their solution with hard empirical data. In [38], the authors provide experimental data from a number of image processing applications. In these experiments, the effective turnaround time of an FPGA-based SA-C implementation was compared to that of an equivalent C implementation on a commodity workstation[9]. The observed application-dependent speedups ranged from a factor 8.8 (adding a constant value to each pixel in an image) to a factor 800.0 (automatic target recognition) in favor of the SA-C implementations.

[7] 'Dusty deck' is derogatory hacker jargon for legacy code. The term was devised to recall images of the days of punch-card programming.

[8] The exact procedure is similar to that used in the Machines system, which will be discussed in 3.3.4.

[9] A Xilinx Virtex E 2000 FPGA and an 800 Mhz. Intel Pentium III MMX were used in the experiments.

Source language

As its name suggests, the source language of the SA-C compiler is a relative of the C programming language. Compared to the purist approach of DEFACTO, however, it is but a distant cousin. The main distinction between SA-C and C is the 'single assignment' restriction of the language. In a single assignment language, a variable may be assigned a value only once during its lifetime. Other important restrictions include the inhibition of recursion and the lack of a pointer dereferencing operator ('`*`'). These restrictions exist to remove the direct bindings between variables and their physical (i.e. memory) locations, inherent in the C language, thus allowing an easier mapping of variables onto circuit wires.

Apart from its restrictions, SA-C introduces a number of new concepts, which serve to compensate for some of the imposed restrictions. The most prominent extensions include true multidimensional arrays, dynamic arrays and an efficient `for`-loop for data processing on an array window. An interesting addition comes in the form of variable sized integer types, e.g. `int12`, `uint128`, etc. To alleviate the single assignment restrictions, the reuse of variable names is allowed.

To aid code generation, the SA-C compiler supports a number of compiler pragmas that allow the programmer to control certain aspects of the generated code. Amongst the options are pragmas for enabling/disabling loop unrolling, strip mining and the possibility to implement a function as a lookup table. Another interesting feature is the SA-C preprocessor, which offers template/generics support, similar to that of C++ and Java.

Model of computation

The astute reader may already have noticed a correlation between the acyclic nature of the data-flow model of computation and the single assignment restriction of SA-C. In both cases, data is pushed 'forward' throughout the computation, inhibiting the possibility to reuse data storage locations. The similarities are no coincidence, as the data-flow network is indeed the SA-C compiler's model of computation.

The restrictions of the SA-C language were designed to make the synthesis process reasonably straightforward. During the initial build phase, a *data-dependence and control-flow graph* is generated. This is a hierarchical graph, which reflects the structure of the source code in terms of loops and function calls. The DDCF representation is then optimized to exploit function- and loop level parallelism, and to decrease circuit size. The DDCF representation is subsequently transformed to a data-flow graph, as per our earlier definition. In the final phases, this graph will be annotated with timing information, to allow further machine-specific optimization. Finally, structural VHDL code is generated as output.

3.3.4 Machines

Machines [42] is a programming model that was designed to bring the power of reconfigurable computing closer to the conventional application programmer. Application of the model requires an object-oriented host language and a special compiler. A C++ implementation of the model exists in the form of an unnamed compiler. The Machines model and its C++ compiler were developed at Hewlett-Packard Laboratories.

Source language

Machines is a class hierarchy that can be implemented in any conventional object-oriented programming language. At the basis of the hierarchy is the `Machine`-class, which defines the pure virtual/abstract `step()` method. The programmer creates an application by extending the `Machine` class and implementing the `step()` method. All machine instances

in an application execute concurrently. The runtime environment calls their `step()` methods at each ‘cycle’.

A child of `Machine` may contain state variables in the form of class attributes. These attributes must either be simple data types (integers, booleans, etc.) or other Machines. Machines can communicate through inputs and outputs, which can be defined by adding `input()` and `output()` methods to the child class. A Java example, adopted from [42], demonstrates how a simple low-pass filter may be implemented:

```
class LowPassFilter extends Machine
{
     int state = 0; // State variable
     protected int inputData; // 'Buffered' input.

     void input( int in ) { inputData = in; }
     void step() { state = state/2 + inputData/2; }
     int output() { return state; }
}
```

Another example demonstrates the composition of Machines into a pipeline of low-pass filters:

```
class FilterPipeline extends Machine
{
     LowPassFilter[] stages = new LowPassFilter[10];

     void step();
     int output() { return stages[9].output(); }
     void input( int in )
     {
          stages[0].input(in);
          for ( int i = 11 i < 10; i++ )
               stages[i].input( stages[i-1].output() );
     }
}
```

The Machines system combines structural semantics at the macroscopic level with behavioral semantics at the instruction level. In this approach it is not dissimilar to VHDL. The boon of Machines, then, is that it has a much friendlier face, at least to those accustomed to conventional programming languages.

As a final note on language: the authors of [42] provide data from a small comparative study of C++/Machines vs. hand-crafted VHDL, in which the Serpent block cipher [6] was used as an example application. It is unfortunate, however, that clock rates and gate utilization are the only metrics cited in the paper[10], leaving us guessing at the differences in effective turnaround time.

Model of computation

Machines is a prime example of the *communicating sequential processes* model of computation. Each user-defined `Machine` class instance is initially treated as a separate entity in a graph of communicating processes. As we mentioned at the time of its introduction, the CSP model is not a complete model of computation. An additional strategy is required to synthesize code for the sequential process bodies. In the Machines-case, the number of possible states of the concurrent processes becomes quite large, as each Machine can have an arbitrary number of internal variables. It for this reason that the system uses a hybrid

[10]Tuned C++/Machines implementations used approx. 66 percent of the gates and ran at nearly three times the clock speeds of the hand-crafted VHDL implementation.

model of computation. The *data-flow graph* model is used for the synthesis of process bodies. The transformations used are similar in nature to those used in tools like DEFACTO and SA-C. We have avoided a detailed discussion of this process until this point, because the Machines designers offer an exceptionally clear description of the two key techniques:

Static single assignment conversion In our treatment of the SA-C language, we saw that the single-assignment restriction was imposed on the programmer in order to facilitate convenient data-flow graph generation. The Machines model imposes no such restriction. Instead, it employs an elegant transformation algorithm to create a single-assignment form. The algorithm simply tags each variable in the original source code with an index. This index is incremented each time a value is assigned to the variable. In subsequent right-hand side use of the variable, the variable with the current index is used, effectively creating a new variable for each value assignment.

Predication With the static single assignment transformation, we have the means to create a data-flow graph from a series of unconditional imperative statements. Conditional constructs still pose a minor challenge, however, as the following example demonstrates:

```
void foo()
{
      int x = 3;
      if ( quux ) x = 7;
      y = x;
}
```

If we attempt to apply the static single assignment transformation to `foo()`, we encounter a problem at the point where `y` is assigned a value: which index would `x` have if `quux` were true, and which if it were false? One solution to this problem is to guard each statement in the source program with a predicate that controls its execution, e.g.:

```
void foo()
{
      int x = 3; : if( true );
      x = 7; : if(quux);
      y = x; : if(true);
}
```

This *predication* technique, which is due to Mahlke [35], can be used to eliminate any conditional construct. During the static single assignment transformation phase, the guards are simply assumed to be true. The guards are implemented in hardware during the final stages of synthesis, i.e. controlling whether a statement operates on its input data or merely propagates it unaltered.

The Machines synthesizer extracts parallelism in two ways. On the one hand, there is the explicit task-level parallelism that is explicitly specified by the developer. On the other, a certain degree of statement- and instruction-level parallelism emerges naturally from the transcription to a data flow graph. The Machines compiler does not use advanced loop optimizing techniques. Rather, it relies on the explicit task level parallelism to bring the large performance gains.

3.3.5 Lava

Lava [12] is a new tool for circuit design, synthesis and simulation. It consists of a collection of modules for the programming language Haskell [31]. Lava is an experimental tool with two current incarnations: Chalmers-Lava from Chalmers University of Technology, Sweden, and Xilinx-Lava from Xilinx, Inc. The former is aimed at formal hardware verification, whilst the latter is being developed with FPGA development in mind. According

to reports on the Chalmers website [17], both parties are moving to unify the two versions. Whether Xilinx is still actively involved is unclear, however, as the Lava page has been removed from the company's website some time during the writing of this report.

Source language

Much like the Machines model of Section 3.3.4, Lava is an 'embedded language' that uses another programming language, in this case Haskell, as a host language for circuit modeling. Contrary to the OO host languages of Machines, Lava's host language Haskell is a purely functional programming language, i.e. all computation is performed through the application of functions without side effects. Functions are first class types in Haskell.

The Lava system revolves around the definition of a `Circuit`-class[11], for types on which basic bit-level operations like `and` and `or` are defined. Extensions to the class exist for higher level circuits, which for example support arithmetic operations. A hardware circuit, then, is described by defining a function that operates on one of the data types of the Circuit class. A half adder example:

```
halfAdd (a, b) = (sum, carry)
   where
      sum   = xor2( a, b )
      carry = and2( a, b )
```

The example shows the definition of a function that accepts a tuple of bits as an input. The `xor2` and `and2` functions are defined in the Lava package. The package also offers a function for circuit simulation. The output of a test run on the `halfAdd` circuit description, fired from the Haskell listener:

```
main> simulate halfAdd(high,high)

(low,high)
```

Through function composition, circuits of arbitrary complexity can be defined. The package itself provides type classes and standard functions for higher-level circuits, e.g. for arithmetic- and list processing. Synthesizing these circuits is as convenient as simulating them:

```
main> writeVHDL "FullAdder" fullAdd

Writing to file "FullAdder.vhdl" ... Done.
```

The embedded modeling language of Lava makes effective and elegant use of the host language's features in many interesting ways. We have chosen not to spoil our simple impression by presenting these details here. For a thorough introduction, we refer to the informative Lava tutorial [34].

Model of computation

The `writeVHDL` function from the Lava package generates a gate level description. The generated VHDL code is purely structural. In the Lava system, synthesis is accomplished by evaluating the circuit description functions on a special input type. The VHDL sources are generated as a side effect[12].

[11] In functional languages, a *type class* is a means of grouping types by the operators that are defined on them. The `Ord` class, for example, consists of all types that can be ordered, i.e. that have == and < operators defined on them. The Integer, for example, would be in the `Ord`-class, whilst a color would not be, for lack of an < operator.

[12] Since Haskell is a purely functional language, side effects can only occur using the language's *monadic* facilities. Monads, unfortunately, cannot be introduced in a mere sentence. We will leave the reader with a reference to a reasonably short introduction to monads instead [1].

A very elegant property of the Lava system is the clear relation between the behavioral semantics of the language and the structure of the generated circuit. Due to the fact that functional languages are stateless, one can easily see how a complex expression can be translated to a data flow graph that carries inherent statement level parallelism. Since Lava's intended use is that of hardware definition, rather than computing, the language provides no constructs for task level parallelism.

3.3.6 GCC/Chimaera

GCC/Chimaera [48] is a C compiler for the Chimaera [29] reconfigurable architecture. The compiler was built upon the foundations of GCC [22], the familiar open source compiler framework of the GNU project. The Chimaera port of GCC was developed at Northwestern University, Illinois. It is not part of the GNU project or the official GCC distribution.

Source language

The GCC/Chimaera compiler is in many respects a fairly standard-issue GCC port. The C language front-end of the suite contains no new code. Significant features of the tool are only found at the back-end.

Model of computation

True to the intentions of Chimaera's designers, the compiler synthesizes code for a custom instruction set architecture, or rather, an augmented instruction set architecture. The generated code consists of regular RISC instructions for the architecture's fixed logic, interspersed with custom operations for the reconfigurable functional unit ('RFUOPs'). The limitation of this model of computation is that task level parallelism is not exploitable.

From the compiler's perspective, an RFUOP can be any operation on multiple input registers, as long as the resulting value fits in a single output register. We will now discuss the source-to-silicon transformation in some detail, in order to understand where and how the RFUOPs are synthesized.

The GCC framework has a flexible design, which facilitates different source languages and targets. Internally, the framework uses a target- (and source-) independent intermediate representation: the *register transfer language*, or RTL. The GCC framework performs generic early optimizations (e.g. constant folding, common subexpression elimination, etc.) on the RTL representation, before the Chimaera-specific optimizations are performed. These latter optimizations, which are also performed on the RTL representation, serve to identify the RFUOPs.

The identification of RFUOPs is accomplished in three steps, named and briefly described as follows:

Control localization is a technique that replaces basic blocks with multiple input/output *macro instructions*. In the special case where only one output register is needed, these macro instructions can be replaced with RFUOPs immediately. Other macro instructions, i.e. those which require multiple output registers, are saved for further optimization during the *instruction combination* phase.

SWAR optimization or 'SIMD Within A Register' optimization identifies opportunities to split the bits of a group of m input registers into multiple smaller n-bit fields for collective SIMD processing. This can, for example, be useful for repetitive processing on large arrays. In this phase, candidate loops are replaced with a series

of pseudo-instructions, which can be recognized during the instruction combination phase.

Instruction combination is the process of determining which instructions can be implemented on the Chimaera functional unit, including but not limited to the special macro instructions and SWAR sequences identified during the first two phases[13]. A configuration is then generated for each replaceable sequence.

For all its effort, the GCC/Chimaera compiler achieves an average speedup of 2.6 over '-O2' optimized non-Chimaera code for a set of benchmark problems [48].

3.3.7 Other synthesis tools

Although the case studies were selected to illustrate a wide variety of approaches, their number is inevitably small in comparison to the sheer number of experimental synthesis tools and strategies that have been proposed in recent years. To an extent, all of these tools represent unique approaches, and a study at least ten times the size of this one would be needed to cover them all. We provide a quick impression:

A number of systems adopt the 'OO hardware modeling' approach of the Machines system, in combination with a modeling framework in a standard- or slightly adapted sequential programming language. These languages typically target communicating sequential processes. The C++ based SystemC [43] and the Java 1.1 based JHDL [9] are mature examples of the Machines-approach. Celoxica's Handel-C [**?**] is a slightly more abstract variation. It uses a C dialect with explicit parallel constructs and coarse-grained communicating sequential processes.

While not technically synthesis tools for reconfigurable platforms, there exist a family of 'reactive programming' languages and compilers that have an execution model which is similar to the simple CSP model (i.e. the concurrent state machines model). These languages have mostly been developed in the mid- to late 1980's, for the express purpose of embedded systems development. They can easily be adapted for use on fine-grained reconfigurable computers. The most important languages in this family are Esterel [11], Lustre [16], Signal [10] and the visual programming system Statecharts [27]. Aside from their common model of computation, these languages are all radically different.

Variations on the 'custom instruction set' theme of Chimaera come in the form of Spyder [32], which utilizes user-defined, hardware accelerated operators within a C++ framework, and the NAPA C [24] compiler, which maps a C dialect with special compiler directives onto the NAPA1000 combined RISC/FPGA architecture. The compiler of the PipeRench system combines the data-flow network model with a custom instruction set architecture.

The data-flow target model of SA-C is used in a number of hardware platforms and their associated development tools. Interest in these architectures bloomed in the 1980's. Prominent examples include the synthesis tools of the Manchester Prototype Dataflow Computer [26], Monsoon [39] and the Sigma-1 [49] machine.

At the base of our discussion of synthesis tools was a collection of loosely defined models of computation. It is worth mentioning that a number of formal models for concurrent computation exist, notably Petri nets and Kahn process networks. These models are sometimes used as abstract targets in synthesis tools. The Compaan/Laura system [50], for example,

[13]At present, only simple arithmetic and shift operations are supported on the RFU.

generates Kahn networks from Matlab source code, as an intermediate target for FPGA based computation.

3.4 Conclusion

A good synthesis tool must generate code that exploits the hardware's opportunities for parallelism effectively. Ideally, it must hide the structural nature of the hardware and extract implicit parallelism from the source program. These idealized goals can hardly be achieved in practice. Synthesis tools that generate effective code require some concessions in the source language, i.e. either explicit constructs for parallelism or for structural programming. In table 3.4, we have listed a summary of properties of the reviewed synthesis tools. The table lists the programming paradigm and the model of computation of each synthesis tool, and an indication whether the source language has explicit constructs for parallel programming and/or structural modeling.

Tool	Paradigm	MOC	Parallelism	Structure
XST	Mixed	Gate level description	Explicit	Explicit
DEFACTO	Imperative	Data flow network	Implicit	Implicit
SA-C	Imperative	Data flow network	Explicit	Implicit
Machines	OO	CSP/Data flow network	Explicit	Explicit
Lava	Functional	Gate level description	Implicit	Explicit
GCC/Chimaera	Imperative	Custom instruction set	Implicit	Implicit

Table 3.1: Synthesis tool summary

In Section 1.4, we presented three criteria by which we could assess the quality of RC development platforms. We will now conclude our study of synthesis tools by relating the theoretical discussion of Sections 3.1 and 3.2, and the case studies of Section 2.2 to these assessment criteria.

Admittance We recall from Section 1.4 that *admittance* is the extent to which a platform allows for the implementation of efficient applications. Centering purely on synthesis tool design, we can assess the admittance of a development platform by comparing the performance of a optimized reference application to that of an implementation of the same algorithm on a conventional architecture. It is unfortunate that we have had to observe that most of the reviewed publications on synthesis tool designs lacked meaningful performance data. Given that performance is the central issue in reconfigurable computing, this is quite inexcusable. This leaves us to conclude that there is insufficient data available to make solid general statements about the effects of source languages and/or models of computation on the quality of the generated code. Based on the small amount of data that is available to us, however, we can conclude that:

- The custom instruction set model of computation, as demonstrated by the Chimaera GCC compiler, does not admit very efficient implementations, due to the fact that the model does not allow larger task units to be executed in parallel.
- The impressive speedups cited by the authors of the SA-C system prove that the data flow model does admit, at least for selected applications, implementations which are more efficient than what can be achieved on conventional architectures.
- Based on knowledge from conventional parallel programming and on the previous point, it is to be expected that a hybrid CSP/data flow network model will yield equally good results.

Accessibility To paraphrase our definition from Section 1.4, the accessibility criterion states that the tool suite for an RC platform should facilitate application development within a reasonable timeframe, compared to development on traditional architectures. The accessibility of a development platform is, in our opinion, dependent upon three factors:

- The semantics of the source language.
- The turnaround time of the synthesis tool.
- The availability of debugging facilities.

When it comes to accessibility, there is a watershed to be made between the 'dusty deck' compilers, such as DEFACTO and, to a lesser extent, the systems which use a conventional language as a host language on the one hand, and the hardware description languages, like VHDL and Verilog, on the other. The benefits of the former group over the latter are evident:

- The familiarity of a traditional language makes a platform immediately accessible to a large number of existing developers.
- Programs can easily be evaluated on traditional workstations for simulation purposes.
- Development times lie in the same ranges as those of traditional applications.
- Developers do not have to deal with hardware related details, such as timing signals, which have nothing to do with the algorithm under consideration.

In relation to source language semantics and debugging facilities, there is no added value to using a HDL over any of the higher level languages for computing purposes. As the DEFACTO system proves, however, there is a considerable penalty to be paid in synthesis tool turnaround time if the use of a higher level language requires elaborate design space exploration techniques in order to yield acceptably efficient code.

Transparency The transparency criterion from Section 1.4 stated that the designs generated by a synthesis tool should harbor no unpleasant performance surprises. There is a clear lesson to be learnt here from the DEFACTO system, with its arcane synthesis process. It is that pure dusty deck imperative code is too difficult to compile into efficient hardware without having to resort to elaborate and unpredictable procedures. While the idea of having a C-to-hardware compiler is appealing from an accessibility perspective, it is virtually impossible to design such a compiler without sacrificing either transparency or admittance. Or, to paraphrase a familiar adage: "easy, fast, reliable: pick two." In conclusion, since execution speed and reliability (transparency) are the essential qualities here, we believe that it will be inevitable that a good synthesis tool will need special source language constructs to aid effective code generation for RC platforms.

Chapter 4

Designing synthesis tools

In Sections 2 and 3, we presented an overview of the considerations and tradeoffs involved in the design of reconfigurable hardware platforms and their development tools. In the subsequent conclusions, we made a case for the further investigation of fixed-plus variable hardware architectures that are based on fine-grained reconfigurable fabrics. We established that the central issue in the development of such platforms is the design of the synthesis tool. In this section, we address the design of synthesis tools in more detail, in the form of a series of design strategies. At the end of this section, we present a research proposal, based on these design strategies.

4.1 Design strategies

The quest for the perfect synthesis tool —or perhaps merely a satisfactory one— can be approached from a number of different starting points. While the result is all that matters in the end, our choice for a home base has great effect on the length and difficulty of the journey. In this section, we discuss what are in our view the four possible starting points for synthesis tool design.

The language-oriented approach

In the first of our four design strategies, we choose the source language —or one might argue, the user— as a starting point. Since the goal of synthesis tool design is to create an environment where implementations can be realized in a 'comfortable' programming language, it makes some sense to assume such a language from the outset, and to implement it in a top-down fashion. We call this approach the *language-oriented approach*, and it is summarized as follows:

1. Design or choose a programming language which provides practical behavioral semantics for the expression of algorithms.

2. Create transformation methods for each of the language's behavioral constructs.

The language-oriented approach is at the basis of many of the actual systems that we have reviewed. The 'dusty deck' tools like the DEFACTO system assume an existing programming language as a starting point, and then bravely attack its constructs with aggressive transformation- and optimization tactics, in order to arrive at a satisfactory result. Other tools, such as the OO based Machines and the Lava system, are also examples of this approach.

The target-oriented approach

The construction of a bridge may be commenced at either side of the water it crosses. While our first design strategy assumed the language of the human developer as a starting point, our second strategy can be considered its inverse. In the *target-oriented approach* to synthesis tool design, the platform's reconfigurable fabric is assumed as a given, and the language is constructed upon it in a bottom-up fashion. The approach in a series of steps:

1. Choose an model of computation which accurately captures the low level capabilities of the hardware.

2. Successively create higher-order behavioral abstractions on top of the hardware model.

While hardware description languages like VHDL and Verilog were developed with a different goal in mind, the tools that synthesize them onto hardware can be considered distorted real-world examples of the target-oriented approach. HDL synthesis tools, such as the discussed XST, generally support the low-level structural semantics of the underlying hardware, and attempt to support as many behavioral constructs as possible, whilst maintaining reasonably efficient mappings. A rare and interesting example of a purely target-oriented approach is the Crystal language [45]. We will return to this language in more detail in the discussion of Sectionsec:discussion.

The algorithm-oriented approach

Our third design strategy is somewhat orthogonal to the first two. It is based on the observation that most computational problems can be decomposed into fairly generic subproblems, such as vector multiplication, sorting, satisfiability, etc. Rather than creating a programming language in which the user can express complex algorithms, one could offer a simple composition language, combined with an ever-expanding toolkit of generic solutions to computational problems, all of which are implemented in some low-level language. The strength of this approach is that it avoids the difficult transition from programming language to hardware design by skipping the language phase altogether. Its weakness, of course, is that its usefulness is limited by the size and quality of the toolkit. We call this strategy the *algorithm-oriented* approach. It is summarized as follows.

1. Devise a rudimentary algorithm-composition language.

2. Develop a toolkit full of efficient implementations for a set of common algorithms.

As mentioned, the algorithm-oriented approach can be deployed in conjunction with one of the earlier methods. In the FPGA market, so-called IP cores ('intellectual property' cores) are available, which offer efficient implementations complete algorithms. The algorithms implemented in these cores are usually somewhat larger than the examples posed above[1]. A more clear example of the idea is found in traditional parallel programming frameworks (e.g. MPI), which offer library functions for efficient parallel execution for e.g. matrix row reduction and prefix summing.

The domain-oriented approach

Our final design strategy is a radically different approach, which is again orthogonal to its precursors. It is based on the idea that similarly natured problems generally require similarly structured implementations. By making bold assumptions about the type of problems that are to be implemented using the system, the complexity of the subsequent synthesis process can be cut down considerably. This final strategy is dubbed the *domain-oriented approach*:

[1]Examples include image processing algorithms, encryption, etc.

1. Define a limited application domain for the synthesis tool.

2. Identify common properties of the problems in the domain.

3. Employ one or more of the other design strategies, using the identified properties as axiomatic assumptions about the computation.

The domain-oriented approach is a design strategy that is clearly demonstrated in synthesis tools and hardware platforms alike. An illustrative example is the Streams-C [23] language, which adopts digital signal processing applications as its problem domain. The Streams-C designers go on to assume that the algorithms in this application domain require regular, repetitive operations on steady, high volume data streams. Axiomatic assumptions such as these imply a great deal of generic architectural infrastructure, which greatly reduces the creativity requirements of the synthesis tool. Other systems that are biased toward particular application domains include PipeRench (stream-based media appplications) and DFFC (real-time vision automata). A very large group of systems are targeted at large volume data processing applications in general, i.e. ignoring intrinsically difficult problems that are not data-driven, such as SAT, number factorization, etc. These systems include: SA-C, Chimaera, Garp and, to an extent, Raw.

4.2 Discussion

Our first design strategy, the language-oriented approach, leaves room for a number of interesting questions. Starting with step 1 of the strategy, one might ask what we consider to be a language with practical behavioral semantics. In the history of computing, imperative languages, for example, have gained a large cultural stronghold in our thinking about algorithms, and many would argue that these languages are the most practical languages for the purpose, merely because of this fact. Unfortunately, we have seen in the earlier sections that implicit parallelism is difficult to exploit in imperative languages, making it hard to generate optimal hardware designs.

The flaws of the language oriented approach, however, are not limited to the imperative paradigm. They are inherent to the approach itself: one cannot, in general, design a programming language in isolation and develop a compiler that generates perfect code after the fact. We cannot do this for abstract programming languages on conventional platforms, so we should not expect to be able to do this on RC platforms either. It is for this reason that any dusty deck effort is doomed to fail.

Rather than adopting a language like C as a programming language and developing a synthesis tool around it, we should use C as an example of a language that has become highly successful in its natural habitat. Rather than taking it away from the Von Neumann architecture, we should look at what made it such a success there, and emulate its development process in the new environment. The C language was developed from the ground up, as an elegant abstraction of assembly language, which could greatly reduce the *accident* of programming, in Brooks [14]' terminology, without compromising the admittance of the platform for fast implementations. Our target oriented approach reflects this approach: we should be able to develop more effective synthesis tools if we work our way up from the hardware, rather than down from the language. In general, it is good software engineering practice to start with a simple program that actually works well and expand its abilities, than to work downward from a broad set of idealized features.

As an example of the strengths of the bottom-up approach, we have already mentioned the Crystal language [45]. It is an elegant language that allows a developer to define memory components (e.g. registers) and to express synchronous state transition rules for these

components. The strength of the Crystal language is that it admits efficient implementations to the extent that HDLs do, yet it hides much of the tedium and verbosity of true low level descriptions.

The beauty of the two final design strategies is that they can easily be combined with the target oriented approach if so desired. Once the foundations of a language have been established, it should be reasonably straightforward to offer optimized implementations of commonly used algorithms in the form of add-on modules, thus encapsulating the algorithm oriented approach in a library system. Similarly, will be straightforward to develop domain specific languages using the hardware oriented approach.

4.3 Conclusion

The goal of reconfigurable computing is to create implementations that are faster than what can be achieved on conventional platforms. Developer friendliness is important, but it is a secondary concern. In order to create convenient programming languages that do not hamper the potential for speedup, we will need to develop languages from the ground up, rather than top down. In order to do this, we will need to establish a body of programs, written in a low level language (a HDL), which demonstrate and exploit the full potential of the reconfigurable hardware. We can then proceed to develop an efficient programming language for RC platforms by doing away with the unnecessary details of these programs.

Chapter 5

Conclusion

In this report, we presented a survey of reconfigurable computing systems. We began this survey with the establishment of the motivation for the development of such systems, and of a set of criteria by means of which both the hardware designs and the application development tools of such platforms could be assessed. We have subsequently discussed the general design properties and a number of actual examples of both hardware systems and development tools.

In Section 2.3, we concluded that *fine grained*, *heterogeneous*, *fully reconfigurable* architectures, such as FPGAs, offer the best opportunities to gain insight into the fundamental strengths and weaknesses of reconfigurable computing. In section 3.4, we concluded that there is a general lack of quantitative evidence for the hypothesis that reconfigurable architectures allow for faster implementations of algorithms than conventional architectures. On a related note, we concluded that it is not possible to make strong general statements about the relative merits of different synthesis tool designs, given this lack of information.

Apart from questions surrounding performance issues, we concluded that little progress has been made in the development of an convenient, abstract programming language that is reasonably transparent to the developer. We suspect that this is in part due to the fact that many ambitious, top down approaches have been explored, in contrast to few pragmatic, bottom up ones.

In light of these results, we propose a research project in which we aim to:

- produce quantitative evidence for the hypothesis that fine-grained reconfigurable computing machines admit implementations of algorithms that are significantly faster than equivalent implementations on conventional architectures.

- seek progress in the development of a language which improves the accessibility of fine-grained reconfigurable systems as an application development platform, whilst maintaining a high degree of transparency and admittance.

We propose the following course of action:

1. We will develop a number of FPGA-based solutions to a variety of computational problems, using a traditional hardware description language.

2. We will compare the effective turnaround times of these FPGA-based implementations to equivalent software implementations of the same algorithms on a conventional architecture.

3. We will attempt to identify and generalize common algorithmic constructs from our experimental implementations, in order to produce a rudimentary programming language, using the bottom-up, hardware oriented design approach described in Section 4.1.

Bibliography

[1] *All About Monads*. http://www.nomaware.com/monads/html/.

[2] *Alphabetical List of Reconfigurable Computing Architectures*. http://www.site.uottawa.ca/ rabielmo/personal/rc.html.

[3] *List of FPGA-based Computing Machines*. http://www.io.com/g̃uccione/HW_list.html.

[4] *MIPS Processor Architecture Documentation*. http://www.mips.com/content/Documentation.

[5] Altera Corporation. *FLEX 8000 Programmable Logic Device Family Data Sheet*, 2003.

[6] R. Anderson, E. Biham, and L. Knudsen. Serpent: a flexible block cipher with maximum assurance, 1998.

[7] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[8] J. W. Backus. Can programming be liberated from the Von Neumann style? *Commun. ACM*, 21(8):613–641, 1978.

[9] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In *FCCM*, pages 175–, 1998.

[10] A. Benveniste, M. L. Borgne, and P. L. Guernic. Signal as a model for real-time and hybrid systems. In *ESOP*, pages 20–38, 1992.

[11] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[12] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, USA, 1998. ACM Press.

[13] G. Borriello, C. Ebeling, S. A. Hauck, and S. Burns. The triptych fpga architecture. *IEEE Trans. Very Large Scale Integr. Syst.*, 3(4):491–501, 1995.

[14] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[15] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The Garp architecture and C compiler. *Computer*, 33(4):62–69, 2000.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[17] Chalmers University of Technology, http://www.cs.chalmers.se/ koen/Lava/. *The Lava Homepage*.

[18] Colorado State University, http://www.cs.colostate.edu/cameron/. *The Cameron Project Homepage*.

[19] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[20] P. C. Diniz, M. W. Hall, J. Park, B. So, and H. Ziegler. Bridging the gap between compilation and synthesis in the DEFACTO system. In *LCPC*, pages 52–70, 2001.

[21] G. Estrin. Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. *IEEE Annals of the History of Computing*, 24(4):3–9, 2002.

[22] Free Software Foundation, http://gcc.gnu.org. *GCC Home Page*.

[23] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the streams-c c-to-fpga compiler: an applications perspective. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 134–140, New York, NY, USA, 2001. ACM Press.

[24] M. B. Gokhale and J. M. Stone. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 126, Washington, DC, USA, 1998. IEEE Computer Society.

[25] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *IEEE Computer*, 33(4):70–77, 2000.

[26] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[27] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., New York, NY, USA, 1998.

[28] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[29] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM '97)*, page 87, Washington, DC, USA, 1997. IEEE Computer Society.

[30] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[31] P. Hudak and J. H. Fasel. A gentle introduction to haskell. *SIGPLAN Not.*, 27(5):1–52, 1992.

[32] C. Iseli and E. Sanchez. Spyder: A SURE (SUperscalar and REconfigurable) processor. 9(3):231–252, 1995.

[33] D. E. Knuth. *The Art of Computer Programming*. Addison Wesley.

[34] M. S. Koen Claessen. A tutorial on lava: A hardware description and verification system, 2000.

[35] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of conditional branches*. PhD thesis, Champaign, IL, USA, 1997.

[36] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. A. E. Spaanenburg. Seeking solutions in configurable computing. *Computer*, 30(12):38–43, 1997.

[37] B. M. Moret. *The Theory of Computation*. Addison Wesley.

[38] W. A. Najjar, W. B. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63–69, 2003.

[39] G. M. Papadopoulos. *Implementation of a general-purpose dataflow multiprocessor*. MIT Press, Cambridge, MA, USA, 1991.

[40] R. Sedgewick. *Algorithms in C*. Addison Wesley.

[41] M. Sima, S. Vassiliadis, S. D. Cotofana, J. T. J. van Eijndhoven, and K. A. Vissers. Field-programmable custom computing machines - a taxonomy. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL 2002). Reconfigurable Computing Is Going Mainstream*, pages 79–88, September 2002.

[42] G. Snider, B. Shackleford, and R. J. Carter. Attacking the semantic gap between application programming languages and configurable hardware. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 115–124, New York, NY, USA, 2001. ACM Press.

[43] Synopsys Inc., http://www.systemc.org. *SystemC Reference Manual*.

[44] M. B. Taylor, J. S. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. "the Raw microprocessor: A computational fabric for software circuits and general-purpose programs.". *IEEE Micro*, 22(2):25–35, 2002.

[45] C. van Reeuwijk. Crystal: a simple programming language for custom computing machines. 2004.

[46] Xilinx, Inc. *Spartan-3 FPGA Family: Complete Data Sheet*, 2005.

[47] Xilinx, Inc. *Virtex-4 Family Overview*, 2005.

[48] Z. A. Ye, N. Shenoy, and P. Baneijee. A c compiler for a processor with a reconfigurable functional unit. In *FPGA '00: Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 95–100, New York, NY, USA, 2000. ACM Press.

[49] T. Yuba, K. Hiraki, T. Shimada, S. Sekiguchi, and K. Nishida. The sigma-1 dataflow computer. In *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 578–585, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[50] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. F. Deprettere. Laura: Leiden architecture research and exploration tool. In *FPL*, pages 911–920, 2003.