Reconfigurable computing: architectures and design methods

T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk and P.Y.K. Cheung

Abstract: Reconfigurable computing is becoming increasingly attractive for many applications. This survey covers two aspects of reconfigurable computing: architectures and design methods. The paper includes recent advances in reconfigurable architectures, such as the Alters Stratix II and Xilinx Virtex 4 FPGA devices. The authors identify major trends in general-purpose and special-purpose design methods. It is shown that reconfigurable computing designs are capable of achieving up to 500 times speedup and 70% energy savings over microprocessor implementations for specific applications.

1 Introduction

Reconfigurable computing is rapidly establishing itself as a major discipline that covers various subjects of learning, including both computing science and electronic engineering. Reconfigurable computing involves the use of reconfigurable devices, such as field programmable gate arrays (FPGAs), for computing purposes. Reconfigurable computing is also known as configurable computing or custom computing, since many of the design techniques can be seen as customising a computational fabric for specific applications [1].

Reconfigurable computing systems often have impressive performance. Consider, as an example, the point multiplication operation in elliptic curve cryptography. For a key size of 270 bits, it has been reported [2] that a point multiplication can be computed in 0.36 ms with a reconfigurable computing design implemented in an XC2V6000 FPGA at 66 MHz. In contrast, an optimised software implementation requires 196.71 ms on a dual-xeon computer at 2.6 GHz; so the reconfigurable computing design is more than 540 times faster, while its clock speed is almost 40 times slower than the Xeon processors. This example illustrates a hardware design implemented on a reconfigurable computing platform. We regard such implementations as a subset of reconfigurable computing, which in general can involve the use of runtime reconfiguration and soft processors.

Is this speed advantage of reconfigurable computing over traditional microprocessors a one-off or a sustainable trend? Recent research suggests that it is a trend rather than a one-off for a wide variety of applications: from image processing [3] to floating-point operations [4].

Sheer speed, while important, is not the only strength of reconfigurable computing. Another compelling advantage is reduced energy and power consumption. In a reconfigurable system, the circuitry is optimised for the application, such that the power consumption will tend to be much lower than that for a general-purpose processor. A recent study [5] reports that moving critical software loops to reconfigurable hardware results in average energy savings of 35% to 70% with an average speedup of 3 to 7 times, depending on the particular device used.

Other advantages of reconfigurable computing include a reduction in size and component count (and hence cost), improved time-to-market, and improved flexibility and upgradability. These advantages are especially important for embedded applications. Indeed, there is evidence [6] that embedded systems developers show a growing interest in reconfigurable computing systems, especially with the introduction of soft cores which can contain one or more instruction processors [7–12].

In this paper, we present a survey of modern reconfigurable system architectures and design methods. Although we also provide background information on notable aspects of older technologies, our focus is on the most recent architectures and design methods, as well as the trends that will drive each of these areas in the near future. In other words, we intend to complement other survey papers [13–17] by:

(i) providing an up-to-date survey of material that appears after the publication of the papers mentioned above;
(ii) identifying explicitly the main trends in architectures and design methods for reconfigurable computing;
(iii) examining reconfigurable computing from a perspective different from existing surveys, for instance classifying design methods as special-purpose and general-purpose;
(iv) offering various direct comparisons of technology options according to a selected set of metrics from different perspectives.

2 Background

Many of today's computationally intensive applications require more processing power than ever before. Applications such as streaming video, image recognition

© IEE, 2005
IEE Proceedings online no. 20045086
doi: 10.1049/ip-cdt:20045086
Paper first received 14th July and in revised form 9th November 2004

T.J. Todman, O. Mencer and W. Luk are with the Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK
G.A. Constantinides and P.Y.K. Cheung are with the Department of Electrical and Electronic Engineering, Imperial College London, Exhibition Rd, South Kensington, London SW7 2BT, UK
S.J.E. Wilton is with the Department of Electrical and Computer Engineering, University of British Columbia, 2356 Main Mall, Vancouver, British Columbia, Canada V6T 1Z4
E-mail: tjt97@doc.ic.ac.uk

and processing, and highly interactive services are placing new demands on the computation units that implement these applications. At the same time, the power consumption targets, the acceptable packaging and manufacturing costs, and the time-to-market requirements of these computation units are all decreasing rapidly, especially in the embedded hand-held devices market. Meeting these performance requirements under the power, cost and time-to-market constraints is becoming increasingly challenging.

In the following, we describe three ways of supporting such processing requirements: high-performance microprocessors, application-specific integrated circuits and reconfigurable computing systems.

High-performance microprocessors provide an off-the-shelf means of addressing processing requirements described earlier. Unfortunately for many applications, a single processor, even an expensive state-of-the-art processor, is not fast enough. In addition, the power consumption (100 W or more) and cost (possibly thousands of dollars) state-of-the-art processors place them out of reach for many embedded applications. Even if microprocessors continue to follow Moore's Law so that their density doubles every 18 months, they may still be unable to keep up with the requirements of some of the most aggressive embedded applications.

Application-specific integrated circuits (ASICs) provide another means of addressing these processing requirements. Unlike a software implementation, an ASIC implementation provides a natural mechanism for implementing the large amount of parallelism found in many of these applications. In addition, an ASIC circuit does not need to suffer from the serial (and often slow and power-hungry) instruction fetch, decode and execute cycle that is at the heart of all microprocessors. Furthermore, ASICs consume less power than reconfigurable devices. Finally, an ASIC can contain just the right mix of functional units for a particular application; in contrast, an off-the-shelf microprocessor contains a fixed set of functional units which must be selected to satisfy a wide variety of applications.

Despite the advantages of ASICs, they are often infeasible or uneconomical for many embedded systems. This is primarily due to two factors: the cost of producing an ASIC often due to the mask's cost (up to $1 million [18]), and the time to develop a custom integrated circuit, can both be unacceptable. Only the very highest-volume applications would the improved performance and lower per-unit price warrant the high nonrecurring engineering (NRE) cost of designing an ASIC.

A third means of providing this processing power is a reconfigurable computing system. A reconfigurable computing system typically contains one or more processors and a reconfigurable fabric upon which custom functional units can be built. The processor(s) executes sequential and noncritical code, while code that can be efficiently mapped to hardware can be 'executed' by processing units that have been mapped to the reconfigurable fabric. Like a custom integrated circuit, the functions that have been mapped to the reconfigurable fabric can take advantage of the parallelism achievable in a hardware implementation. Also like an ASIC, the embedded system designer can produce the right mix of functional and storage units in the reconfigurable fabric, providing a computing structure that matches the application.

Unlike an ASIC, however, a new fabric need not be designed for each application. A given fabric can implement a wide variety of functional units. This means that a reconfigurable computing system can be built out of off-the-shelf components, significantly reducing the long design-time inherent in an ASIC implementation. Also unlike an ASIC, the functional units implemented in the reconfigurable fabric can change over time. This means that as the environment or usage of the embedded system changes, the mix of functional units can adapt to better match the new environment. The reconfigurable fabric in a handheld device, for instance, might implement large matrix multiply operations when the device is used in one mode, and large signal processing functions when the device is used in another mode.

Typically, not all of the embedded system functionality needs to be implemented by the reconfigurable fabric. Only those parts of the computation that are time-critical and contain a high degree of parallelism need to be mapped to the reconfigurable fabric, while the remainder of the computation can be implemented by a standard instruction processor. The interface between the processor and the fabric, as well as the interface between the memory and the fabric, are therefore of the utmost importance. Modern reconfigurable devices are large enough to implement instruction processors within the programmable fabric itself: soft processors. These can be general purpose, or customised to a particular application; application specific instruction processors and flexible instruction processors are two such approaches. Section 4.3.2 deals with soft processors in more detail.

Other devices show some of the flexibility of reconfigurable computers. Examples include graphics processor units and application specific array processors. These devices perform well on their intended application, but cannot run more general computations, unlike reconfigurable computers and microprocessors.

Despite the compelling promise of reconfigurable computing, it has limitations of which designers should be aware. For instance, the flexible routing on the bit level tends to produce large silicon area and performance overhead when compared with ASIC technology. Hence for large volume production of designs in applications without the need for field upgrade, ASIC technology or gate array technology can still deliver higher performance design at lower unit cost than reconfigurable computing technology. However, since FPGA technology tracks advances in memory technology and has demonstrated impressive advances in the last few years, many are confident that the current rapid progress in FPGA speed, capacity and capability will continue, together with the reduction in price.

It should be noted that the development of reconfigurable systems is still a maturing field. There are a number of challenges in developing a reconfigurable system. We describe three of such challenges below.

First, the structure of the reconfigurable fabric and the interfaces between the fabric, processor(s) and memory must be very efficient. Some reconfigurable computing systems use a standard field-programmable gate array [19–24] as a reconfigurable fabric, while others adopt custom-designed fabrics [25–36].

Another challenge is the development of computer-aided design and compilation tools that map an application to a reconfigurable computing system. This involves determining which parts of the application should be mapped to the fabric and which should be mapped to the processor, determining when and how often the reconfigurable fabric should be reconfigured, which changes the functional units implemented in the fabric, as well as the specification of algorithms for efficient mappings to the reconfigurable system.

In this paper, we provide a survey of reconfigurable computing, focusing our discussion on both the issues described above. In the following Section, we provide a survey of various architectures that are found useful for reconfigurable computing; material on design methods will follow.

3 Architectures

We shall first describe system-level architectures for reconfigurable computing. We then present various flavours of reconfigurable fabric. Finally we identify and summarise the main trends.

3.1 System-level architectures

A reconfigurable system typically consists of one or more processors, one or more reconfigurable fabrics, and one or more memories. Reconfigurable systems are often classified according to the degree of coupling between the reconfigurable fabric and the CPU. Compton and Hauck [14] present the four classifications shown in Fig. 1*a–d*. In Fig. 1*a*, the reconfigurable fabric is in the form of one or more stand-alone devices. The existing input and output mechanisms of the processor are used to communicate with the reconfigurable fabric. In this configuration, the data transfer between the fabric and the processor is relatively slow, so this architecture only makes sense for applications in which a significant amount of processing can be done by the fabric without processor intervention. Emulation systems often take on this sort of architecture [37, 38].

Figures 1*b*, *c* show two intermediate structures. In both cases, the cost of communication is lower than that of the architecture in Fig. 1*a*. Architectures of these types are described in [28, 29, 33, 35, 39–42].

Next, Fig. 1*d* shows an architecture in which the processor and the fabric are very tightly coupled; in this case, the reconfigurable fabric is part of the processor itself; perhaps forming a reconfigurable sub-unit that allows for the creation of custom instructions. Examples of this sort of architecture have been described in [30, 32, 36, 43].

Figure 1*e* shows a fifth organisation. In this case, the processor is embedded in the programmable fabric. The processor can either be a 'hard' core [44, 45], or can be a 'soft' core which is implemented using the resources of the programmable fabric itself [7–12].

A summary of the above organisations can be found in Table 1. Note that the bandwidth is the theoretical maximum available to the CPU: for example, in Chess [30], we assume that each block RAM is being accessed at its maximum rate. Organisation (a) is by far the most common, and accounts for all commercial reconfigurable platforms.

3.2 Reconfigurable fabric

The heart of any reconfigurable system is the reconfigurable fabric. The reconfigurable fabric consists of a set of reconfigurable functional units, a reconfigurable interconnect, and a flexible interface to connect the fabric to the rest of the system. In this Section, we review each of these components, and show how they have been used in both commercial and academic reconfigurable systems.

A common theme runs through this entire section: in each component of the fabric, there is a tradeoff between flexibility and efficiency. A highly flexible fabric is typically much larger and much slower than a less flexible fabric. On the other hand, a more flexible fabric is better able to adapt to the application requirements.

In the following discussions, we will see how this tradeoff has influenced the design of every part of every

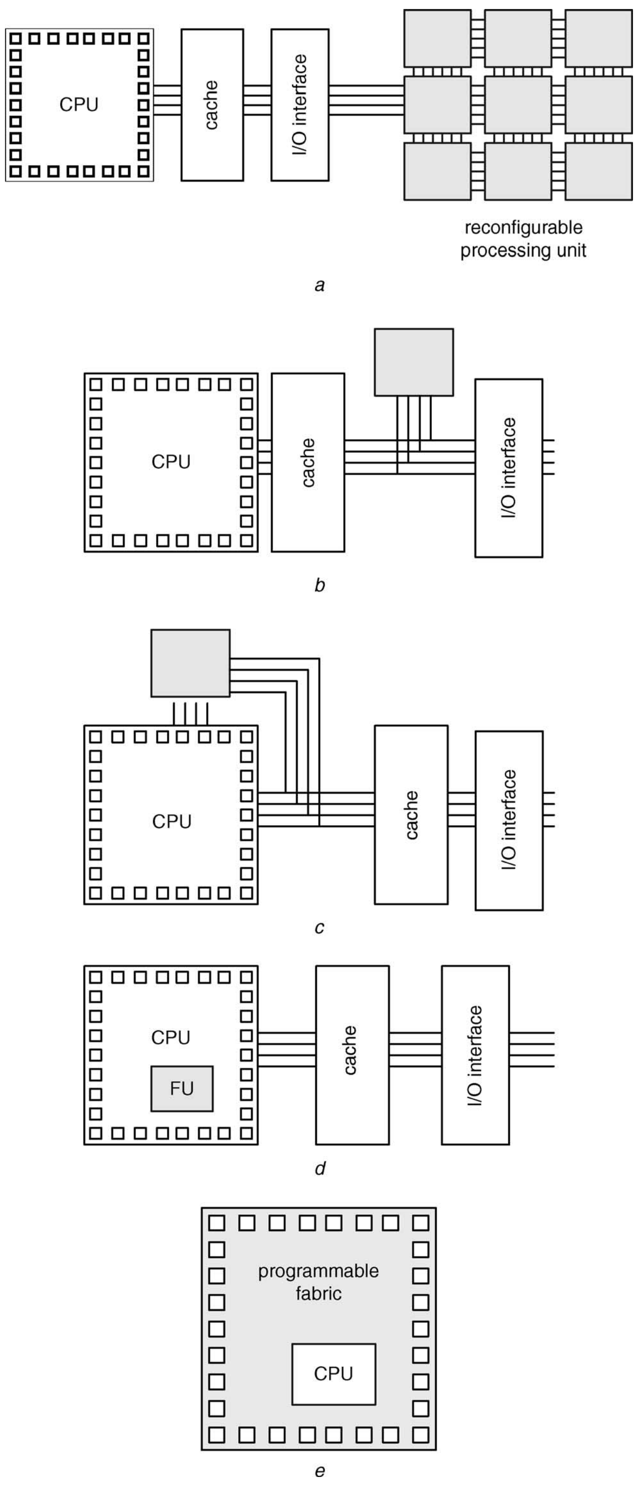


Fig. 1 *Five classes of reconfigurable systems*

The first four are adapted from [14]
a External stand-alone processing unit
b Attached processing unit
c Co-processor
d Reconfigurable functional unit
e Processor embedded in a reconfigurable fabric

reconfigurable system. A summary of the main features of various architectures can be found in Table 2.

3.2.1 Reconfigurable functional units:

Reconfigurable functional units can be classified as either coarse-grained or fine-grained. A fine-grained functional unit can typically implement a single function on a single (or small number) of bits. The most common kind of fine-grained

Table 1: Summary of system architectures

Class	CPU to memory bandwidth, MB/s	Shared memory size	Fine grained or coarse grained	Example application
(*a*) External stand-alone processing unit				
RC2000 [46]	528	152 MB	Fine grained	Video processing
(*b*)/(*c*) Attached processing unit/co-processor				
Pilchard [47]	1064	20 kbytes	Fine grained	DES encryption
Morphosys [35]	800	2048 bytes	Coarse grained	Video compression
(*d*) Reconfigurable functional unit				
Chess [30]	6400	12288 bytes	Coarse grained	Video processing
(*e*) Processor embedded in a reconfigurable fabric				
Xilinx Virtex II Pro [24]	1600	1172 kB	Fine grained	Video compression

Table 2: Comparison of reconfigurable fabrics and devices

Fabric of device	Fine grained or coarse grained	Base logic component	Routing architecture	Embedded memory	Special Features
Actel ProASIC+ [19]	Fine	3-input block	Horizontal and vertical tracks	256×9 bit blocks	Flash-based
Altera Excalibur [44]	Fine	4-input lookup tables	Horizontal and vertical tracks	2 kbit memory blocks	ARMv4T embedded processor
Altera Stratix II [20]	Fine/coarse	8-input adaptive logic module	Horizontal and vertical tracks	512 bits, 4 kbits, and 512 kbit blocks	DSP blocks
Garp [29]	Fine	Logic or arithmetic functions on four 2-bit input words	2-bit buses in horizontal and vertical columns	External to fabric	
Xilinx Virtex II Pro [45]	Fine	4-input lookup tables	Horizontal and vertical tracks	18 kbit blocks	Embedded multipliers, PowerPC 405 processor
Xilinx Virtex II [24]	Fine	4-input lookup tables	Horizontal and vertical tracks	18 kbit blocks	Embedded multipliers
DReAM [48]	Coarse	8-bit ALUs	16-bit local and global buses	Two 16×8 dual port memory	Targets mobile applications
Elixent D-fabrix [27]	Coarse	4-bit ALUs	4-bit buses	256×8 memory blocks	
HP Chess [30]	Coarse	4-bit ALUs	4-bit buses	256×8 bit memories	
IMEC ADRES [31]	Coarse	32-bit ALUs	32-bit buses	Small register files in each logic component	
Matrix [32]	Coarse	8-bit ALUs	Hierarchical 8-bit buses	256×8 bit memories	
MorphoSys [35]	Coarse	ALU and multiplier, and shift units	Buses	External to fabric	
Piperench [28]	Coarse	8-bit ALUs	8-bit buses	External to fabric	Functional units arranged in 'stripes'
RaPiD [26]	Coarse	ALUs	Buses	Embedded memory blocks	
Silicon Hive Avispa [34]	Coarse	ALUs, shifters, accumulators and multipliers	Buses	Five embedded memories	

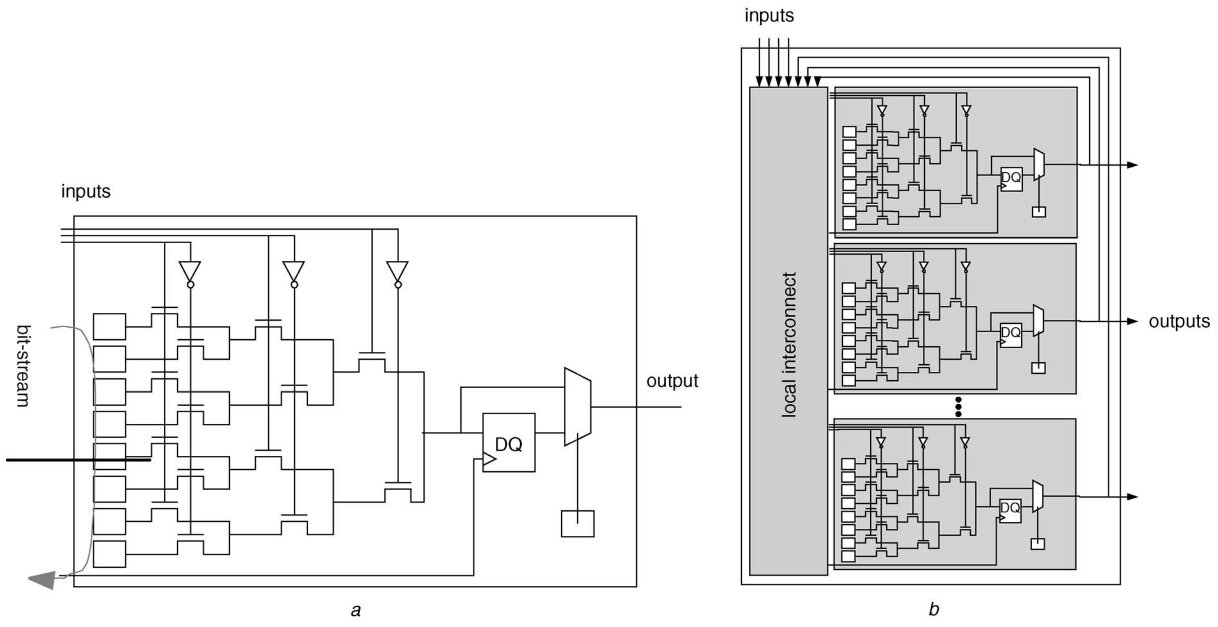


Fig. 2 *Fine-grained reconfigurable functional units*

a Three-input lookup table
b Cluster of lookup tables

functional units are the small lookup tables that are used to implement the bulk of the logic in a commercial field-programmable gate array. A coarse-grained functional unit, on the other hand, is typically much larger, and may consist of arithmetic and logic units (ALUs) and possibly even a significant amount of storage. In this Section, we describe the two types of functional units in more detail.

Many reconfigurable systems use commercial FPGAs as a reconfigurable fabric. These commercial FPGAs contain many three to six input lookup tables, each of which can be thought of as a very fine-grained functional unit. Figure 2*a* illustrates a lookup table; by shifting in the correct pattern of bits, this functional unit can implement any single function of up to three inputs – the extension to lookup tables with larger numbers of inputs is clear. Typically, lookup tables are combined into clusters, as shown in Fig. 2*b*. Figure 3 shows clusters in two popular FPGA families. Figure 3*a* shows a cluster in the Altera Stratix device; Altera calls these clusters 'logic array blocks' [20]. Figure 3*b* shows a cluster in the Xilinx architecture [24]; Xilinx calls these clusters 'configurable logic blocks' (CLBs). In the Altera diagram, each block labelled 'LE' is a lookup table, while in the Xilinx diagram, each 'slice' contains two lookup tables. Other commercial FPGAs are described in [19, 21–23].

Reconfigurable fabrics containing lookup tables are very flexible, and can be used to implement any digital circuit. However, compared to the coarse-grained structures in Section 3.2.2, these fine-grained structures have significantly more area, delay and power overhead. Recognising that these fabrics are often used for arithmetic purposes, FPGA companies have added additional features such as carry-chains and cascade-chains to reduce the overhead when implementing common arithmetic and logic functions. Figure 4 shows how the carry and cascade chains, as well as the ability to break a 4-input lookup table into four two-input lookup tables, can be exploited to efficiently implement carry-select adders [20]. The multiplexers and the exclusive-or gate in Fig. 4 are included as part of each logic array block, and need not be implemented using other lookup tables.

The example in Fig. 4 shows how the efficiency of commercial FPGAs can be improved by adding

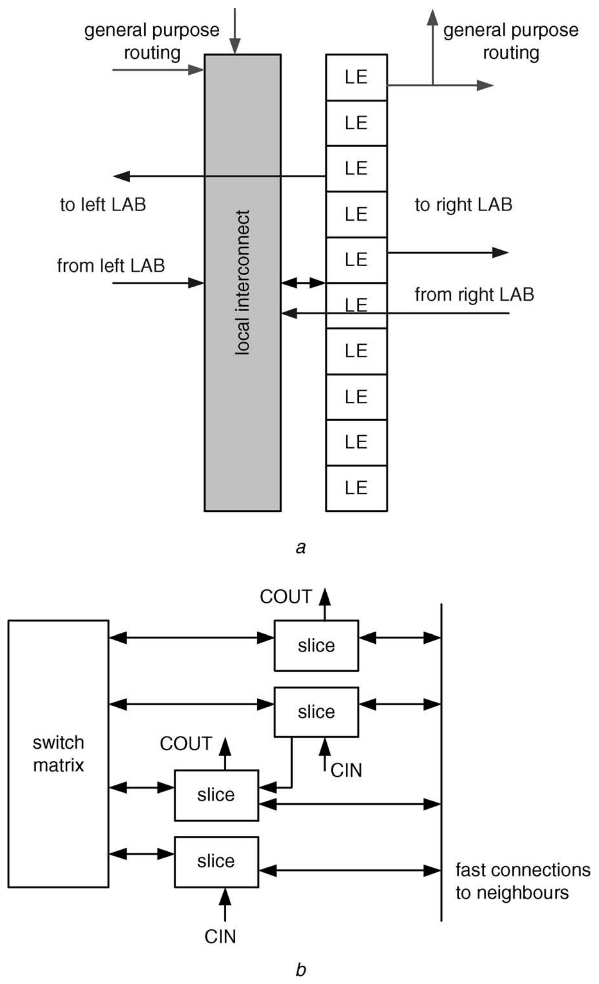


Fig. 3 *Commercial logic block architectures*

a Altera logic array block [20]
b Xilinx configurable logic block [24]

architectural support for common functions. We can go much further than this, though, and embed significantly larger, but far less flexible, reconfigurable functional units. There are two kinds of devices that contain coarse-grained

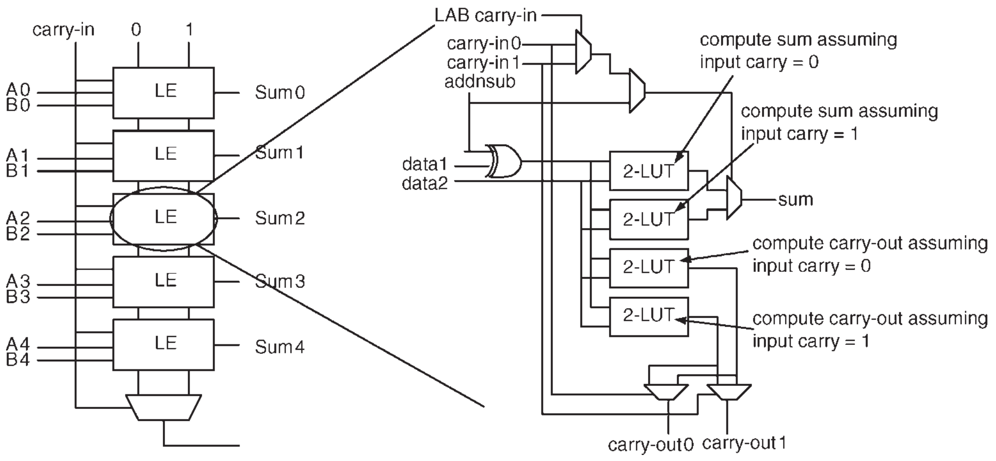

carry-in
0
1
A0
B0
LE
Sum0
A1
B1
LE
Sum1
A2
B2
LE
Sum2
A3
B3
LE
Sum3
A4
B4
LE
Sum4
LAB carry-in
carry-in0
carry-in1
addnsub
data1
data2
2-LUT
2-LUT
2-LUT
2-LUT
compute sum assuming input carry = 0
compute sum assuming input carry = 1
sum
compute carry-out assuming input carry = 0
compute carry-out assuming input carry = 1
carry-out0
carry-out1

Fig. 4 *Implementing a carry-select adder in an Altera Stratix device [20]*
'LUT' denotes 'lookup table'

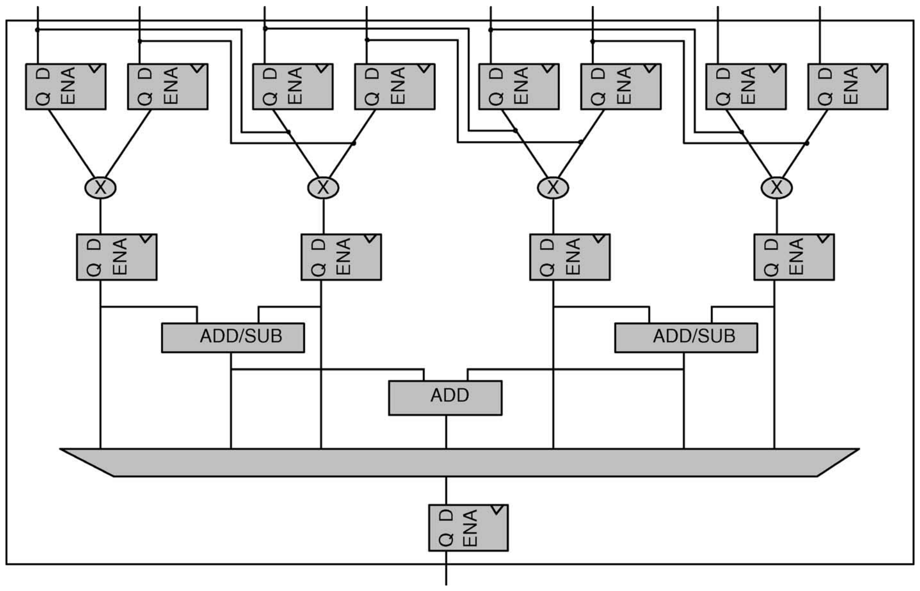


Fig. 5 *Altera DSP block [20]*

functional units; modern FPGAs, which are primarily composed of fine-grained functional units, are increasingly being enhanced by the inclusion of larger blocks. As an example, the Xilinx Virtex device contains embedded 18-bit by 18-bit multiplier units [24]. When implementing algorithms requiring a large amount of multiplication, these embedded blocks can significantly improve the density, speed and power of the device. On the other hand, for algorithms which do not perform multiplication, these blocks are rarely useful. The Altera Stratix devices contain a larger but more flexible embedded block, called a DSP block, shown in Fig. 5 [20]. Each of these blocks can perform accumulate functions as well as multiply operations. The comparison between the two devices clearly illustrates the flexibility and overhead tradeoff; the Altera DSP block may be more flexible than the Xilinx multiplier, however, it consumes more chip area and runs somewhat slower.

The commercial FPGAs described above contain both fine-grained and coarse-grained blocks. There are also devices which contain only coarse-grained blocks [25, 26, 28, 30, 31, 35]. An example of a coarse-grained architecture is the ADRES architecture shown in Fig. 6 [31]. Each reconfigurable functional unit in this device contains a 32-bit ALU which can be configured to implement one of several functions including addition, multiplication and

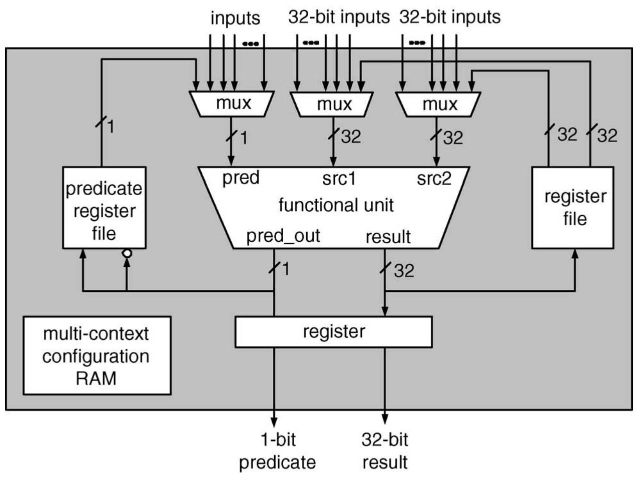


Fig. 6 *ADRES reconfigurable functional unit [31]*

logic functions, with two small register files. Clearly, such a functional unit is far less flexible than the fine-grained functional units described earlier; however, if the application requires functions which match the capabilities of the ALU, these functions can be very efficiently implemented in this architecture.

3.2.2 Reconfigurable interconnects: Regardless of whether a device contains fine-grained functional units, coarse-grained functional units, or a mixture of the two, the functional units needed to be connected in a flexible way. Again, there is a tradeoff between the flexibility of the interconnect (and hence the reconfigurable fabric) and the speed, area and power-efficiency of the architecture.

As before, reconfigurable interconnect architectures can be classified as fine-grained or coarse-grained. The distinction is based on the granularity with which wires are switched. This is illustrated in Fig. 7, which shows a flexible interconnect between two buses. In the fine-grained architecture in Fig. 7*a*, each wire can be switched independently, while in Fig. 7*b* the entire bus is switched as a unit. The fine-grained routing architecture in Fig. 7*a* is more flexible, since not every bit needs to be routed in the same way; however, the coarse-grained architecture in Fig. 7*b* contains far fewer programming bits, and hence suffers much less overhead.

Fine-grained routing architectures are usually found in commercial FPGAs. In these devices, the functional units are typically arranged in a grid pattern, and they are connected using horizontal and vertical channels. Significant research has been performed in the optimisation of the topology of this interconnect [49, 50]. Coarse-grained routing architectures are commonly used in devices containing coarse-grained functional units. Figure 8 shows two examples of coarse-grained routing architectures: (a) the Totem reconfigurable system [25]; (b) the Silicon Hive reconfigurable system [34], which is less flexible but faster and smaller.

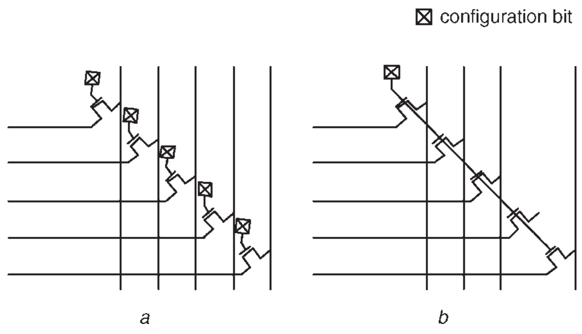
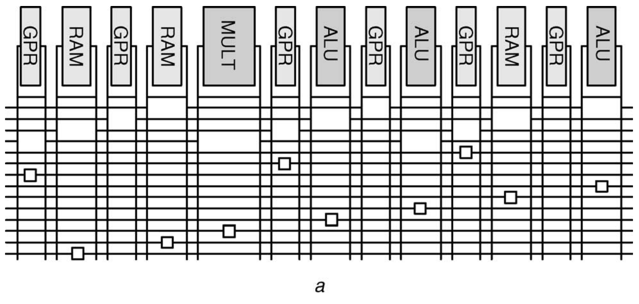


Fig. 7 *Routing architectures*
a Fine-grained
b Coarse-grained



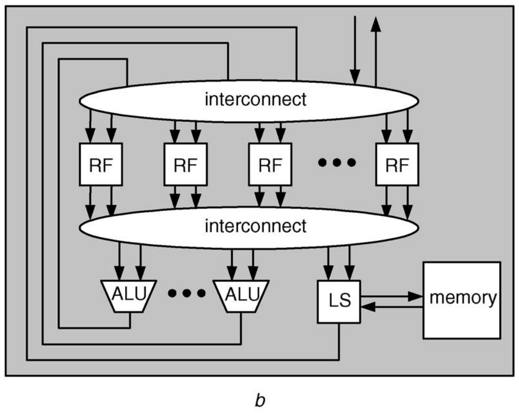


Fig. 8 *Example coarse-grained routing architectures*
a Totem coarse-grained routing architecture [25]
b Silicon Hive coarse-grained routing architecture [34]

3.2.3 Emerging directions: Several emerging directions will be covered in the following. These directions include low-power techniques, asynchronous architectures and molecular microelectronics:

- *Low-power techniques*: Early work explores the use of low-swing circuit techniques to reduce the power consumption in a hierarchical interconnect for a low-energy FPGA [51]. Recent work involves: (a) activity reduction in power-aware design tools, with energy saving of 23% [52]; (b) leakage current reduction methods such as gate biasing and multiple supply-voltage integration, with up to two times leakage power reduction [53]; and (c) dual supply-voltage methods with the lower voltage assigned to noncritical paths, resulting in an average power reduction of 60% [54].
- *Asynchronous architectures:* There is an emerging interest in asynchronous FPGA architectures. An asynchronous version of Piperench [28] is estimated to improve performance by 80%, at the expense of a significant increase in configurable storage and wire count [55]. Other efforts in this direction include fine-grained asynchronous pipelines [56], quasi delay-insensitive architectures [57], and globally asynchronous locally synchronous techniques [58].
- *Molecular microelectronics:* In the long term, molecular techniques offer a promising opportunity for increasing the capacity and performance of reconfigurable computing architectures [59]. Current work is focused on developing programmable logic arrays based on molecular-scale nanowires [60, 61].

3.3 Architectures: main trends

The following summarises the main trends in architectures for reconfigurable computing.

3.3.1 Coarse-grained fabrics: As reconfigurable fabrics are migrated to more advanced technologies, the cost (in terms of both speed and power) of the interconnect part of a reconfigurable fabric is growing. Designers are responding to this by increasing the granularity of their logic units, thereby reducing the amount of interconnect needed. In the Stratix II device, Altera moved away from simple 4-input lookup tables, and used a more complex logic block which can implement functions of up to 7 inputs. We should expect to see a slow migration to more complex logic blocks, even in stand-alone FPGAs.

3.3.2 Heterogeneous functions: As devices are migrated to more advanced technologies, the number of transistors that can be devoted to the reconfigurable logic fabric increases. This provides new opportunities to embed complex nonprogrammable (or semi-programmable) functions, creating heterogeneous architectures with both general-purpose logic resources and fixed-function embedded blocks. Modern Xilinx parts have embedded 18 by 18 bit multipliers, while modern Altera parts have embedded DSP units which can perform a variety of multiply/accumulate functions. Again, we should expect to see a migration to more heterogeneous architectures in the near future.

3.3.3 Soft cores: The use of 'soft' cores, particularly for instruction processors, is increasing. A 'soft' core is one in which the vendor provides a synthesisable version of the function, and the user implements the function using the reconfigurable fabric. Although this is less area- and speed-efficient than a hard embedded core, the flexibility and the ease of integrating these soft cores makes them attractive. The extra overhead becomes less of a hindrance as the number of transistors devoted to the reconfigurable fabric increases. Altera and Xilinx both provide numerous soft cores, including soft instruction processors such as NIOS [7] and Microblaze [12]. Soft instruction processors have also been developed by a number of researchers, ranging from customisable JVM and MIPS processors [10] to ones specialised for machine learning [8] and data encryption [9].

4 Design methods

Hardware compilers for high-level descriptions are increasingly recognised to be the key to reducing the productivity gap for advanced circuit development in general, and for reconfigurable designs in particular. This Section looks at high-level design methods from two perspectives: special-purpose design and general-purpose design. Low-level design methods and tools, covering topics such as technology mapping, floor-planning, and place and route, are beyond the scope of this paper – interested readers are referred to [14].

4.1 General-purpose design

This Section describes design methods and tools based on a general-purpose programming language such as C, possibly adapted to facilitate hardware development. Of course, traditional hardware description languages like VHDL and Verilog are widely available, especially on commercial reconfigurable platforms.

A number of compilers from C to hardware have been developed. Some of the significant ones are reviewed here. These range from compilers which only target hardware to those which target complete hardware/software systems; some also partition into hardware and software components.

We can classify different design methods into two approaches: the annotation and constraint-driven approach, and the source-directed compilation approach. The first approach preserves the source programs in C or C++ as much as possible and makes use of annotation and constraint files to drive the compilation process. The second approach modifies the source language to let the designer to specify, for instance, the amount of parallelism or the size of variables.

4.1.1 Annotation and constraint-driven approach: The systems mentioned below employ annotations in the source-code and constraint flies to control the optimisation process. Their strength is that usually only minor changes are needed to produce a compilable program from a software description – no extensive restructuring is required. Five representative methods are SPC [62], Streams-C [63], Sea Cucumber [64], SPARK [65] and Catapult-C [66].

SPC [62] combines vectorisation, loop transformations and retiming with automatic memory allocation to improve performance. SPC accelerates C loop nests with data dependency restrictions, compiling them into pipelines. Based on the SUIF framework [67], this approach uses loop transformations, and can take advantage of runtime reconfiguration and memory access optimisation. Similar methods have been advocated by other researchers [68, 69].

Streams-C [63] compiles a C program to synthesisable VHDL. Streams-C exploits coarse-grained parallelism in stream-based computations; low-level optimisations such as pipelining are performed automatically by the compiler.

Sea Cucumber [64] compiles Java programs to hardware using a similar scheme to Handel-C, which we detail in Section 4.1.2. Unlike Handel-C, no language extensions are needed; like Streams-C, users must call a library, in this case based on communicating sequential processes (CSP [70]). Multiple circuit implementations of the library primitives enable tradeoffs.

SPARK [65] is a high-level synthesis framework targeting multimedia and image processing. It compiles C code with the following steps: (a) list scheduling based on speculative code motions and loop transformations; (b) resource binding pass with minimisation of interconnect; (c) finite state machine controller generation for the scheduled datapath; (d) code generation producing synthesisable register-transfer level VHDL. Logic synthesis tools then synthesise the output.

Catapult C synthesises register transfer level (RTL) descriptions from unannotated C++, using characterisations of the target technology from RTL synthesis tools [66]. Users can set constraints to explore the design space, controlling loop pipelining and resource sharing.

4.1.2 Source-directed compilation approach: A different approach adapts the source language to enable explicit description of parallelism, communication and other customisable hardware resources such as variable size. Examples of design methods following this approach include ASC [71], Handel-C [72], Haydn-C [73] and Bach-C [77].

ASC [71] adopts C++ custom types and operators to provide a C++ programming interface on the algorithm, architecture, arithmetic and gate levels. This enables the user to program on the desired level for each part of the application. Semi-automated design space exploration further increases design productivity, while supporting the optimisation process on all available levels of abstraction. The object-oriented design enables efficient code-reuse, and includes an integrated arithmetic unit generation library [74]. A floating-point library [75] provides over 200 different floating point units, each with custom bitwidths for mantissa and exponent.

Handel-C [72] extends a subset of C to support flexible width variables, signals, parallel blocks, bit-manipulation operations, and channel communication. A distinctive feature is that timing of the compiled circuit is fixed at one cycle per C statement. This allows Handel-C programmers to schedule hardware resources manually. Handel-C compiles to a *one-hot* state machine using a token-passing scheme developed by Page and Luk [76]; each assignment of the program maps to exactly one control flip-flop in the state machine. These control flip-flops capture the flow of control (represented by the token) in the program: if the control flip-flop corresponding to a particular statement is active, then control has passed to that statement, and the circuitry compiled for that statement is activated. When the statement has finished execution, it passes the token to the next statement in the program.

Haydn-C [73] extends Handel-C for component-based design. Like Handel-C, it supports description of parallel blocks, bit-manipulation operations and channel

Table 3: Summary of general-purpose hardware compilers

System	Approach	Source language	Target language	Target architecture	Example applications
Streams-C [63]	Annotation/ constraint-driven	C + library	RTL VHDL	Xilinx FPGA	Image contrast enhancement, pulsar detection [78]
Sea Cucumber [64]	Annotation/ constraint-driven	Java + library	EDIF	Xilinx FPGA	none given
SPARK [65]	Annotation/ constraint-driven	C	RTL VHDL	LSI, Altera FPGAs	MPEG-1 predictor, image tiling
SPC [62]	Annotation/ constraint-driven	C	EDIF	Xilinx FPGAs	String pattern matching, image skeletonisation
ASC [71]	Source-directed compilation	C++ using class library	EDIF	Xilinx FPGAs	Wavelet compression, encryption
Handel-C [72]	Source-directed compilation	Extended C	Structural VHDL, Verilog, EDIF	Actel, Altera Xilinx FPGAs	Image processing, polygon rendering [79]
Haydn-C [73]	Source-directed compilation	Extended C	Extended C (Handel-C)	Xilinx FPGAs	FIR filter, image erosion
Bach-C [77]	Source-directed compilation	Extended C	Behavioural and RTL VHDL	LSI FPGAs	Viterbi decoders, image processing

communication. The principal innovation of Haydn-C is a framework of optional annotations to enable users to describe design constraints, and to direct source-level transformations such as scheduling and resource allocation. There are automated transformations so that a single high-level design can be used to produce many implementations with different trade-offs. This approach has been evaluated using various case studies, including FIR filters, fractal generators and morphological operators. The fastest morphological erosion design is 129 times faster and 3.4 times larger than the smallest design.

Bach-C [77] is similar to Handel-C but has an untimed semantics, only synchronising between parallel threads on synchronous communications between them, possibly giving greater scope for optimisation. It also allows asynchronous communications but otherwise resembles Handel-C, using the same basic one-hot compilation scheme.

Table 3 summarises the various compilers discussed in this Section, showing their approach, source and target languages, target architecture and some example applications. Note that the compilers discussed are not necessarily restricted to the architectures reported; some can usually be ported to a different architecture by using a different library of hardware primitives.

4.2 Special-purpose design

Within the wide variety of problems to which reconfigurable computing can be applied, there are many specific problem domains which deserve special consideration. The motivation is to exploit domain-specific properties: (a) to describe the computation, such as using MATLAB for digital signal processing, and (b) to optimise the implementation, such as using word-length optimisation techniques described later.

We shall begin with an overview of digital signal processing and relevant tools which target reconfigurable implementations. We then describe the word-length optimisation problem, the solution to which promises rich rewards; an example of such a solution will be covered. Finally we summarise other domain-specific design methods which have been proposed for video and image processing and networking.

4.2.1 *Digital signal processing:*

One of the most successful applications for reconfigurable computing is real-time digital signal processing (DSP). This is illustrated by the inclusion of hardware support for DSP in the latest FPGA devices, such as the embedded DSP blocks in Altera Stratix II chips [20].

DSP problems tend to share the following properties: design latency is usually less of an issue than design throughput, algorithms tend to be numerically intensive but have very simple control structures, controlled numerical error is acceptable, and standard metrics, such as signal-to-noise ratio, exist for measuring numerical precision quality.

DSP algorithm design is often initially performed directly in a graphical programming environment such as Mathworks' MATLAB Simulink [80]. Simulink is widely used within the DSP community, and has been recently incorporated into the Xilinx System Generator [81] and Altera DSP builder [82] design flows. Design approaches such as this are based on the idea of data-flow graphs (DFGs) [83].

Tools working with this form of description vary in the level of user intervention required to specify the numerical properties of the implementation. For example, in the Xilinx System Generator flow [81], it is necessary to specify the number of bits used to represent each signal, the scaling of each signal (*namely* the binary point location), and whether to use saturating or wrap-around arithmetic [84].

Ideally, these implementation details could be automated. Beyond a standard DFG-based algorithm description, only one piece of information should be required: a lower-bound on the *output* signal to quantisation noise acceptable to the user. Such a design tool would thus represent a truly 'behavioural' synthesis route, exposing to the DSP engineer only those aspects of design naturally expressed in the DSP application domain.

4.2.2 *The word-length optimisation problem:*

Unlike microprocessor-based implementations where the word-length is defined a priori by the hard-wired architecture of the processor, reconfigurable computing based on FPGAs allows the size of each variable to be customised to produce the best tradeoffs in numerical accuracy, design size, speed and power consumption. The use of such custom data representation for optimising designs is one of the main strengths of reconfigurable computing.

Given this flexibility, it is desirable to automate the process of finding a good custom data representation. The most important implementation decision to automate is the selection of an appropriate word-length and scaling for each signal [85] in a DSP system. Unlike microprocessor-based implementations, where the word-length is defined a priori by the hard-wired architecture of the processor, reconfigurable computing allows the word-length of each signal to be customised to produce the best tradeoffs in numerical accuracy, design size, speed, and power consumption. The use of custom data representation is one of the greatest strengths.

It has been argued that, often, the most efficient hardware implementation of an algorithm is one in which a wide variety of finite precision representations of different sizes are used for different internal variables [86]. The accuracy observable at the outputs of a DSP system is a function of the word-lengths used to represent all intermediate variables in the algorithm. However, accuracy is less sensitive to some variables than to others, as is implementation area. It is demonstrated in [85] that, by considering error and area information in a structured way using analytical and semi-analytical noise models, it is possible to achieve highly efficient DSP implementations.

In [87] it has been demonstrated that the problem of word-length optimisation is NP-hard, even for systems with special mathematical properties that simplify the problem from a practical perspective [88]. There are, however, several published approaches to word-length optimisation. These can be classified as heuristics offering an area / signal quality tradeoff [86, 89, 90], approaches that make some simplifying assumptions on error properties [89, 91], or optimal approaches that can be applied to algorithms with particular mathematical properties [92].

Some published approaches to the word-length optimisation problem use an analytic approach to scaling and/or error estimation [90, 93, 94], some use simulation [89, 91], and some use a hybrid of the two [95]. The advantage of analytical techniques is that they do not require representative simulation stimulus, and can be faster; however, they tend to be more pessimistic. There is little analytical work on supporting data-flow graphs containing cycles, although in [94] finite loop bounds are supported, while [88] supports cyclic data-flow when the nodes are of a restricted set of types, extended to the semi-analytic technique with fewer restrictions in [96].

Some published approaches use worst-case instantaneous error as a measure of signal quality [90, 91, 93], whereas some use signal-to-noise ratio [86, 89].

The remainder of this Section reviews in some detail particular research approaches in the field.

The Bitwise Project [94] proposes propagation of integer variable ranges backwards and forwards through data-flow graphs. The focus is on removing unwanted most-significant bits (MSBs). Results from integration in a synthesis flow indicate that area savings of between 15% and 86% combined with speed increases of up to 65% can be achieved compared to using 32-bit integers for all variables.

The MATCH Project [93] also uses range propagation through data-flow graphs, except that variables with a fractional component are allowed. All signals in the model of [93] must have equal fractional precision; the authors propose an analytic worst-case error model to estimate the required number of fractional bits. Area reductions of 80% combined with speed increases of 20% are reported when compared to a uniform 32-bit representation.

Wadekar and Parker [90] have also proposed a methodology for word-length optimisation. Like [93], this technique also allows controlled worst-case error at system outputs; however, each intermediate variable is allowed to take a word-length appropriate to the sensitivity of the output errors to quantisation errors on that particular variable. Results indicate area reductions of between 15% and 40% over the optimum uniform word-length implementation.

Kum and Sung [89] and Cantin *et al.* [91] have proposed several word-length optimisation techniques to tradeoff system area against system error. These techniques are heuristics based on bit-true simulation of the design under various internal word-lengths.

In Bitsize [97, 98], Abdul Gaffar *et al.* propose a hybrid method based on the mathematical technique known as automatic differentiation to perform bitwidth optimisation. In this technique, the gradients of outputs with respect to the internal variables are calculated and then used to determine the sensitivities of the outputs to the precision of the internal variables. The results show that it is possible to achieve an area reduction of 20% for floating-point designs, and 30% for fixed-point designs, when given an output error specification of 0.75% against a reference design.

A useful survey of algorithmic procedures for word-length determination has been provided by Cantin *et al.* [99]. In this work, existing heuristics are classified under various categories. However the ‘exhaustive’ and ‘branch-and-bound’ procedures described in [99] do not necessarily capture the optimum solution to the word-length determination problem, due to nonconvexity in the constraint space: it is actually possible to have a *lower* error at a system output by reducing the word-length at an internal node [100]. Such an effect is modelled in the MILP approach proposed in [92].

A comparative summary of existing optimisation systems is provided in Table 4. Each system is classified according to the several defining features described below.

- Is the word-length and scaling selection performed through analytical or simulation-based means?
- Can the system support algorithms exhibiting cyclic data flow (such as infinite impulse response filters)?
- What mechanisms are supported for most significant bit (MSB) optimisations (such as ignoring MSBs that are known to contain no useful information, a technique determined by the scaling approach used)?
- What mechanisms are supported for least significant bit (LSB) optimisations? These involve the monitoring of word-length growth. In addition, for those systems that support error-tradeoffs, further optimisations include the quantisation (truncation or rounding) of unwanted LSBs.
- Does the system allow the user to tradeoff numerical accuracy for a more efficient implementation?

4.2.3 An example optimisation flow:

One possible design flow for word-length optimisation, used in the Right-Size system [96] is illustrated in Fig. 9 for Xilinx FPGAs. The inputs to this system are a specification of the system behaviour (e.g. using Simulink), a specification of the acceptable signal-to-noise ratio at each output, and a set of representative input signals. From these inputs, the tool automatically generates a synthesisable structural description of the architecture and a bit-true behavioural VHDL testbench, together with a set of expected outputs for the provided set of representative inputs. Also generated is

Table 4: Comparison of world-length and scaling optimisation systems and methods

System	Analytical/simulation	Cyclic data flow?	MSB-optimisation	LSB-optimisations	Error tradeoff	Comments
Benedetti [101]	analytical	none	through interval arithmetic	through 'multi-interval' approach	no error	can be pessimistic
Stephenson [94, 102]	analytical	for finite loop bounds	through forward and backward range propagation	none	no error	less pessimistic than [101] due to backwards propagation
Nayak [93]	analytical	not supported for error analysis	through forward and backward range propagation	through fixing number of fractional bits for all variables	user-specified or inferred absolute bounds on error	fractional parts have equal word-length
Wadekar [90]	analytical	none	through forward rang propagation	through genetic algorithm search for suitable word-lengths	user-specified absolute bounds	uses Taylor series at limiting values to determine error propagation
Keding [103, 104]	hybrid	with user intervention	through user-annotations and forward range propagation	through user-annotations and forward world-length propagation	not automated	possible truncation error
Cmar [95]	hybrid for scaling simulation for error	with user intervention only	through combined simulation and forward range propagation	word-length bounded through hybrid fixed or floating simulation	not automated	less pessimistic than [101] due to input error propagation
Kum [89, 105–107]	simulation (hybrid for multiply-accumulate signals in [89, 105])	yes	through measurement of variable mean and standard deviation	through heuristics based on simulation results	user-specified bounds and metric	long simulation time possible
Constantinides [85]	analytical	yes	through tight analytical bounds on signal range and automatic design of saturation arithmetic	through heuristics based on an analytic noise model	user-specified bounds on noise power and spectrum	only applicable to linear time-invariant systems
Constantinides [96]	hybrid	yes	through simulation	through heuristics based on a hybrid noise model	user-specified bounds on noise power and spectrum	only applicable to differentiable nonlinear systems
Abdul Gaffar [97, 98]	hybrid	with user intervention	through simulation based range propagation	through automatic differentiation based dynamic analysis	user-specified bounds and metric	covers both fixed-point and floating-point

a makefile which can be used to automate the post-Right-Size synthesis process.

Application of Right-Size to various adaptive filters implemented in a Xilinx Virtex FPGA has resulted in area reduction of up to 80%, power reduction of up to 98%, and speedup of up to 36% over common alternative design methods without word-length optimisation.

4.2.4 Other design methods:

Besides signal processing, video and image processing is another area that can benefit from special-purpose design methods. Three examples will be given to provide a flavour of this approach. First, the CHAMPION system [108] maps designs captured in the Cantata graphical programming environment to multiple reconfigurable computing platforms. Second, the IGOL framework [109] provides a layered architecture for facilitating hardware plug-ins to be incorporated in various applications in the Microsoft Windows operating system, such as Premiere, Winamp, VirtualDub and DirectShow. Third, the SA-C compiler [110] maps a high-level single-assignment language specialised for image processing description into hardware, using various optimisation methods including loop unrolling, array value propagation, loop-carried array elimination and multi-dimensional stripmining.

Recent work indicates that another application area that can benefit from special-purpose techniques is networking. Two examples will be given. First, a framework has been developed to enable description of designs in the network policy language Ponder [111], into reconfigurable hardware implementations [112]. Second, it is shown [113] how descriptions in the Click networking language can produce efficient reconfigurable designs.

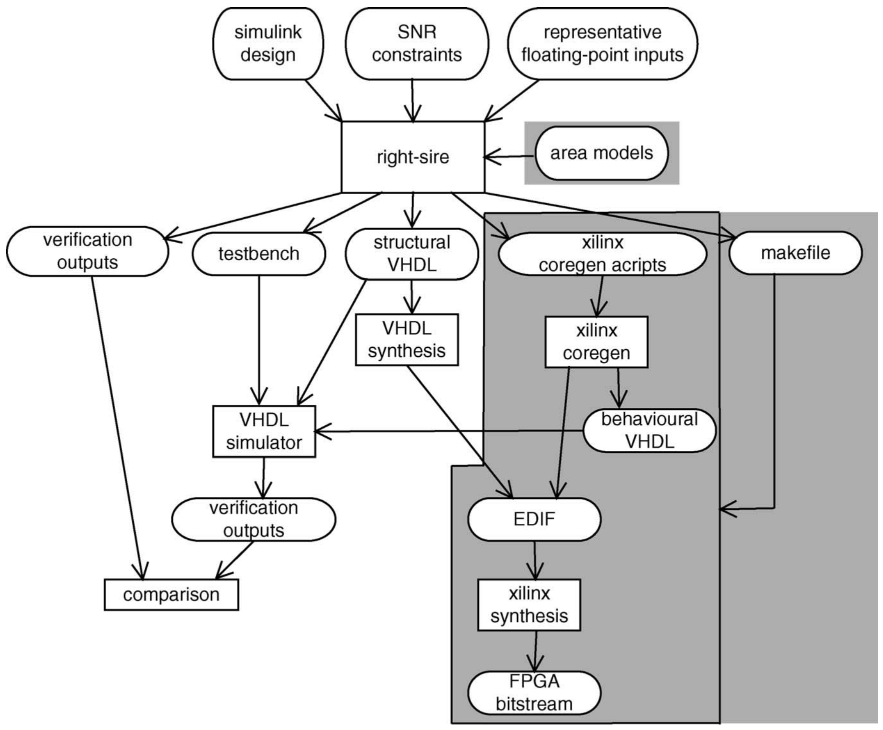


Fig. 9 *Design flow for Right-Size tool [96]*
Shaded portions are FPGA vendor-specific

4.3 Other design methods

In the following, we describe various design methods in brief.

4.3.1 Run-time customisation:

Many aspects of runtime reconfiguration have been explored [14], including the use of directives in high-level descriptions [114]. Effective runtime customisation hinges on appropriate design-time preparation for such customisation. To illustrate this point, consider a runtime customisable system that supports partial reconfiguration: one part of the system continues to be operational, while another part is being reconfigured. As FPGAs get larger, partial reconfiguration is becoming increasingly important as a means of reducing reconfiguration time. To support partial reconfiguration, appropriate circuits must be built at fabrication time as part of the FPGA fabric. Then, at compile time, an initial configuration bitstream and incremental bitstreams have to be produced, together with runtime customisation facilities which can be executed, for instance, on a microprocessor serving as part of the runtime system [115]. Runtime customisation facilities can include support for condition monitoring, design optimisation and reconfiguration control.

Opportunities for runtime design optimisation include: (a) runtime constant propagation [116], which produces a smaller circuit with higher performance by treating runtime data as constant, and optimising them principally by Boolean algebra: (b) library-based compilation – the DISC compiler [117] makes use of a library of precompiled logic modules which can be loaded into reconfigurable resources by the procedure call mechanism; (c) exploiting information about program branch probabilities [118]; the idea is to promote utilisation by dedicating more resources to branches which execute more frequently. A hardware compiler has been developed to produce a collection of designs, each optimised for a particular branch probability; the best can be selected at runtime by incorporating observed branch probability information from a queueing network performance model.

4.3.2 Soft instruction processors:

FPGA technology can now support one or more soft instruction processors implemented using reconfigurable resources on a single chip; proprietary instruction processors, like MicroBlaze and Nios, are now available from FPGA vendors. Often such processors support customisation of resources and custom instructions. Custom instructions have two main benefits. First, they reduce the time for instruction fetch and decode, provided that each custom instruction replaces several regular instructions. Second, additional resources can be assigned to a custom instruction to improve performance. Bit-width optimisation, described in Section 4.2, can also be applied to customise instruction processors at compile time. A challenge of customising instruction processors is that the tools for producing and analysing instructions also need to be customised. For instance, the *flexible instruction processor* framework [10] has been developed to automate the steps in customising an instruction processor and the corresponding tools. Other researchers have proposed similar approaches [119].

Instruction processors can also run declarative languages. For instance, a scalable architecture [8], consisting of multiple processors based on the Warren Abstract Machine, has been developed to support the execution of the Progol system [120], based on the declarative language Prolog. Its effectiveness has been demonstrated using the mutagenesis data set containing 12 000 facts about chemical compounds.

4.3.3 Multi-FPGA compilation:

Peterson *et al.* have developed a C compiler which compiles to multi-FPGA systems [121]. The available FPGAs and other units are specified in a library file, allowing portability. The compiler can generate designs using speculative and lazy execution to improve performance, and ultimately they aim to partition a single program between host and reconfigurable resource (hardware/software codesign). Duncan *et al.* have developed a system with similar capabilities [122]. This is also retargetable, using hierarchical architecture descriptions. It synthesises a VLIW architecture that can be partitioned across multiple FPGAs. Both methods can split designs across several FPGAs, and are retargetable via hardware description libraries. Other C-like languages that have been developed include MoPL-3, a C extension supporting data procedural compilation for the Xputer architecture which comprises an array of reconfigurable ALUs [123], and spC, a systolic parallel C variant for the Enable + + board [124].

4.3.4 Hardware/software codesign:

Several research groups have studied the problem of compiling C code to both hardware and software. The Garp compiler [125] is intended to accelerate plain C, with no annotations to help the compiler, making it more widely applicable. The work targets one architecture only: the Garp chip, which integrates a RISC core and reconfigurable logic. This compiler also uses the SUIF framework. The compiler uses a technique first developed for VLIW architectures called hyperblock scheduling, which optimises for instruction-level parallelism across several common paths, at the expense of rarer paths. Infeasible or rare paths are implemented on the processor with the more common, easily parallelisable paths synthesised into logic for the reconfigurable resource. Similarly, the NAPA C compiler targets the NAPA architecture [126], which also integrates a RISC processor reconfigurable logic. This compiler can also work on plain C code but the programmer can add C pragmas to indicate large-scale parallelism and the bit-widths of variables to the code. The compiler can synthesise pipelines from loops.

4.3.5 Annotation-free compilation:

Some researchers aim to compile a sequential program, without any annotations, into efficient hardware design. This requires analysis of the source program to extract parallelism for an efficient result, which is necessary if compilation from languages such as C is to compete with traditional methods for designing hardware. One example is the work of Babb *et al.* [127], targeting custom, fixed-logic implementation while also applicable to reconfigurable hardware. The compiler uses the SUIF infrastructure to do several analyses to find what computations affect exactly what data, as far as possible. A tiled architecture is synthesised, where all computation is kept as local as possible to one tile. More recently, Ziegler *et al.* [128] have used loop transformations in mapping loop nests onto a pipeline spanning several FPGAs. A further effort is given by the Garp project [125].

4.4 Emerging directions

4.4.1 Verification:

As designs are becoming more complex, techniques for verifying their correctness are becoming increasingly important. Four approaches are described: (i) The InterSim framework [129] provides a means of combining software simulation and hardware prototyping. (ii) The Lava system [130] can convert designs into a form suitable for input to a model checker; a number of FPGA design libraries have been verified in this way [131]. (iii) The Ruby language [132] supports correctness-preserving transformations, and a wide variety of hardware designs have been produced. (iv) The Pebble [133] hardware design language has been formally specified [134], so that provably-correct design tools can be developed.

4.4.2 Customisable hardware compilation:

Recent work [135] explains how customisable frameworks for hardware compilation can enable rapid design exploration, and reusable and extensible hardware optimisation. The framework compiles a parallel imperative language like Handel-C, and supports multiple levels of design abstraction, transformational development, optimisation by compiler passes, and metalanguage facilities. The approach has been used in producing designs for applications, such as signal and image processing, with different tradeoffs in performance and resource usage.

4.5 Design methods: main trends

We summarise the main trends in design methods for reconfigurable computing below.

4.5.1 Special-purpose design:

As explained earlier, special-purpose design methods and tools enable both high-level design and domain-specific optimisation. Existing methods, such as those compiling MATLAB Simulink descriptions into reconfigurable computing implementations [81, 82, 93, 96, 97, 136], allow application developers without electronic design experience to produce efficient hardware implementations quickly and effectively. This is an area that would assume further importance in future.

4.5.2 Low-power design:

Several hardware compilers aim to minimise the power consumption of their generated designs. Examples include special-purpose design methods such as Right-Size [96] and PyGen [136], and general-purpose methods that target loops for configurable hardware implementation [5]. These design methods, when combined with low-power architectures [54] and power-aware low-level tools [52], can provide significant reduction in power consumption.

4.5.3 High-level transformation:

Many hardware design methods [62, 65, 110] involve high-level transformations: loop unrolling, loop restructuring and static single assignment are three examples. The development of powerful transformations for design optimisation will continue for both special-purpose and general-purpose designs.

5 Summary

This paper surveys two aspects of reconfigurable computing: architectures and design methods. The main trends in architectures are coarse-grained fabrics, heterogeneous functions and soft cores. The main trends in design methods are special-purpose design methods, low-power techniques and high-level transformations. We wonder what a survey paper on reconfigurable computing, written in 2015, will cover?

6 Acknowledgments

Our thanks to Ray Cheung and Sherif Yusuf for their support in preparing this paper. The support of Celoxica, Xilinx and UK EPSRC (grant numbers GR/R 31409, GR/R 55931, GR/N 66599) is gratefully acknowledged.

7 References

1 Luk, W.: 'Customising processors: design-time and run-time opportunities', *Lect. Notes Comput. Sci.*, 2004, **3133**
2 Telle, N., Cheung, C.C., and Luk, W.: 'Customising hardware designs for elliptic curve cryptography', *Lect. Notes Comput. Sci.*, 2004, **3133**
3 Guo, Z., Najjar, W., Vahid, F., and Vissers, K.: 'A quantitative analysis of the speedup factors of FPGAs over processors'. Proc. Int. Symp. on FPGAs (ACM Press, 2004)
4 Underwood, K.: 'FPGAs vs. CPUs: trends in peak floating-point performance'. Proc. Int. Symp. on FPGAs (ACM Press, 2004)

5 Stitt, G., Vahid, F., and Nematbakhsh, S.: 'Energy savings and speedups from partitioning critical software loops to hardware in embedded systems', *ACM Trans. Embedded Comput. Syst.*, 2004, **3**, (1), pp. 218–232
6 Vereen, L.: 'Soft FPGA cores attract embedded developers', *Embedded Syst. Program.*, 2004, 23 April 2004, http://www.embedded.com//showArticle.jhtml?articleID=19200183
7 Altera Corp., Nios II Processor Reference Handbook, May 2004
8 Fidjeland, A., Luk, W., and Muggleton, S.: 'Scalable acceleration of inductive logic programs'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2002
9 Leong, P.H.W., and Leung, K.H.: 'A microcoded elliptic curve processor using FPGA technology', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2002, **10**, (5), pp. 550–559
10 Seng, S.P., Luk, W., and Cheung, P.Y.K.: 'Flexible instruction processors'. Proc. Int. Conf. on Compilers, Arch. and Syn. for Embedded Systems (ACM Press, 2000)
11 Seng, S.P., Luk, W., and Cheung, P.Y.K.: 'Run-time adaptive flexible instruction processors', *Lect. Notes Comput. Sci.*, 2002, **2438**
12 Xilinx, Inc., Microblaze Processor Reference Guide, June 2004
13 Bondalapati, K., and Prasanna, V.K.: 'Reconfigurable computing systems', *Proc. IEEE*, 2002, **90**, (7), pp. 1201–1217
14 Compton, K., and Hauck, S.: 'Reconfigurable computing: a survey of systems and software', *ACM Comput. Surv.*, 2002, **34**, (2), pp. 171–210
15 Luk, W., Cheung, P.Y.K., and Shirazi, N.: 'Configurable computing', in Chen, W.K. (Ed.): 'Electrical engineer's handbook' (Academic Press, 2004)
16 Schaumont, P., Verbauwhede, I., Keutzer, K., and Sarrafzadeh, M.: 'A quick safari through the reconfiguration jungle'. Proc. Design Automation Conf., ACM Press, 2001
17 Tessier, R., and Burleson, W.: 'Reconfigurable computing and digital signal processing: a survey', *J. VLSI Signal Process.*, 2001, **28**, pp. 7–27
18 Saxe, T., and Faith, B.: 'Less is more with FPGAs' EE Times, 13 September 2004 `http://www.eetimes.com/showArticle.jhtml?articleID=47203801`
19 Actel Corp., ProASIC Plus Family Flash FPGAs, v3.5, April 2004
20 Altera Corp., Stratix II Device Handbook, February 2004
21 Lattice Semiconductor Corp, ispXPGA Family, January 2004
22 Morris, K.: 'Virtex 4: Xilinx details its next generation', *FPGA Program. Logic J.*, 2004, June
23 Quicklogic Corp., Eclipse-II Family Datasheet, January 2004
24 Xilinx, Inc., Virtex II Datasheet, June 2004
25 Compton, K., and Hauck, S.: 'Totem: Custom reconfigurable array generation'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2001)
26 Ebeling, C., Conquist, D., and Franklin, P.: 'RaPiD – reconfigurable pipelined datapath', *Lect. Notes Comput. Sci. Misc.*, 1996, **1142**
27 Elixent Corporation, DFA 1000 Accelerator Datasheet, 2003
28 Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R.: 'PipeRench: a reconfigurable architecture and compiler', *Computer*, 2000, **33**, (4), pp. 70–77
29 Hauser, J.R., and Wawrzynek, J.: 'Garp: a MIPS processor with a reconfigurable processor'. IEEE Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 1997)
30 Marshall, A., Stansfield, T., Kostarnov, I., Vuillemin, J., and Hutchings, B.: 'A reconfigurable arithmetic array for multimedia applications', ACM/SIGDA Int. Symp. on FPGAs, Feb 1999, pp. 135–143
31 Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R.: 'ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix', *Lect. Notes Comput. Sci.*, 2003, **2778**
32 Mirsky, E., and DeHon, A.: 'MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 1996)
33 Rupp, C.R., Landguth, M., Garverick, T., Gomersall, E., Holt, H., Arnold, J., and Gokhale, M.: 'The NAPA adaptive processing architecture'. IEEE Symp. on Field-Programmable Custom Computing Machines, May 1998, pp. 28–37
34 Silicon Hive: 'Avispa Block Accelerator'. Product Brief, 2003
35 Singh, H., Lee, M.-H., Lu, G., Kurdahi, F., Bagherzadeh, N., and Chaves, E.: 'MorphoSys: an integrated reconfigurable system for data-parallel and compute intensive applications', *IEEE Trans. Comput.*, 2000, **49**, (5), pp. 465–481
36 Taylor, M., *et al*: 'The RAW microprocessor: a computational fabric for software circuits and general purpose programs', *IEEE Micro*, 2002, **22**, (2), pp. 25–35
37 Cadence Design Systems Inc, Palladium Datasheet, 2004
38 Mentor Graphics, Vstation Pro: High Performance System Verification, 2003
39 Annapolis Microsystems, Inc., Wildfire Reference Manual, 1998
40 Laufer, R., Taylor, R., and Schmit, H.: 'PCI-PipeRench and the SwordAPI: a system for stream-based reconfigurable computing'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 1999)
41 Vuillemin, J., Bertin, P., Roncin, D., Shand, M., Touati, H., and Boucard, P.: 'Programmable active memories: reconfigurable systems come of age', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1996, **4**, (1), pp. 56–69
42 Wittig, R.D., and Chow, P.: 'OneChip: an FPGA processor with reconfigurable logic'. IEEE Symp. on FPGAs for Custom Computing Machines, 1996
43 Razdan, R., and Smith, M.D.: 'A high performance microarchitecture with hardware programmable functional units'. *Int. Symp. on Micro-architecture*, 1994, pp. 172–180
44 Altera Corp., Excalibur Device Overview, May 2002
45 Xilinx, Inc., PowerPC 405 Processor Block Reference Guide, October 2003
46 Celoxica, RC2000 Development and evaluation board data sheet, version 1.1, 2004
47 Leong, P., Leong, M., Cheung, O., Tung, T., Kwok, C., Wong, M., and Lee, K.: 'Pilchard – a reconfigurable computing platform with memory slot interface'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2001)
48 Becker, J., and Glesner, M.: 'A parallel dynamically reconfigurable architecture designed for flexible application-tailored hardware/software systems in future mobile communication', *J. Supercomput.*, 2001, **19**, (1), pp. 105–127
49 Betz, V., Rose, J., and Marquardt, A.: 'Architecture and CAD for deep-submicron FPGAs' (Kluwer Academic Publishers, February 1999)
50 Lemieux, G., and Lewis, D.: 'Design of interconnect networks for programmable logic' (Kluwer Academic Publishers, 2004)
51 George, V., Zhang, H., and Rabaey, J.: 'The design of a low energy FPGA'. Proc. Int. Symp. on Low Power Electronics and Design, 1999
52 Lamoureux, J., and Wilton, S.J.E.: 'On the interaction between power-aware FPGA CAD algorithms', IEEE Int. Conf. on Computer-Aided Design, 2003
53 Rahman, A., Polavarapuv, V.: 'Evaluation of low-leakage design techniques for field programmable gate arrays'. Proc. Int. Symp. on Field-Programmable Gate Arrays (ACM Press, 2004)
54 Gayasen, A., Lee, K., Vijaykrishnan, N., Kandemir, M., Irwin, M.J., and Tuan, T.: 'A dual-V_{DD} low power FPGA architecture', *Lect. Notes Comput. Sci.*, 2004, **3203**
55 Kagotani, H., and Schmit, H.: 'Asynchronous PipeRench: architecture and performance evaluations'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
56 Teife, J., and Manohar, R.: 'Programmable asynchronous pipeline arrays', *Lect. Notes Comput. Sci.*, 2003, **2778**
57 Wong, C.G., Martin, A.J., and Thomas, P.: 'An architecture for asynchronous FPGAs'. Proc. Int. IEEE Conf. on Field-Programmable Technology, 2003
58 Royal, A., and Cheung, P.Y.K.: 'Globally asynchronous locally synchronous FPGA architectures', *Lect. Notes Comput. Sci.*, 2003, **2778**
59 Butts, M., DeHon, A., and Goldstein, S.: "Molecular electronics: devices, systems and tools for gigagate, gigabit chips', Proc. IEEE Int. Conf. on Computer-Aided Design, 2002
60 DeHon, A., and Wilson, M.J.: 'Nanowire-based sublithographic programmable logic arrays'. Proc. Int. Symp. on FPGAs (ACM Press, 2004)
61 Williams, R.S., and Kuekes, P.J.: 'Molecular nanoelectronics'. Proc. IEEE Int. Symp. on Circuits and Systems, 2000
62 Weinhardt, M., and Luk, W.: 'Pipeline vectorization', *IEEE Trans. Comput.-Aided Des.*, 2001, **20**, (2), pp. 234–248
63 Gokhale, M., Stone, J.M., Arnold, J., and Kalinowski, M.: 'Stream-oriented FPGA computing in the Streams-C high level language'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2000)
64 Jackson, P.A., Hutchings, B.L., and Tripp, J.L.: 'Simulation and synthesis of CSP-based interprocess communication'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
65 Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations'. Proc. Int. Conf. on VLSI Design, January 2003
66 McCloud, S.: 'Catapult C Synthesis-based design flow: speeding implementation and increasing flexibility'. White Paper, Mentor Graphics, 2004.
67 Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.-W., Tseng, C.-W., Hall, M.W., Lam, M.S., and Hennessy, J.L.: 'SUIF: an infrastructure for research on parallelizing and optimizing compilers', *SIGPLAN Not.*, 1994, **29**, (12), pp. 31–37
68 Harriss, T., Walke, R., Kienhuis, B., and Deprettere, E.: 'Compilation from Matlab to process networks realized in FPGA', *Des. Autom. Embedded Syst.*, 2002, **7**, (4), pp. 385–403
69 Schreiber, R., *et al.*: 'PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators', *J. VLSI Signal Process. Syst.*, 2002, **31**, (2), pp. 127–142
70 Hoare, C.A.R.: 'Communicating sequential processes' (Prentice Hall, 1985)
71 Mencer, O., Pearce, D.J., Howes, L.W., and Luk, W.: 'Design space exploration with a stream compiler'. Proc. IEEE Int. Conf. on Field Programmable Technology, 2003
72 Celoxica, Handel-C Language Reference Manual for DK2.0, Document RM-1003-4.0, 2003
73 De Figueiredo Coutinho, J.G., and Luk, W.: 'Source-directed transformations for hardware compilation'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2003
74 Mencer, O.: 'PAM-Blox II: design and evaluation of C++ module generation for computing with FPGAs'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2002)

75 Liang, J., Tessier, R., and Mencer, O.: 'Floating point unit generation and evaluation for FPGAs'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
76 Page, I., and Luk, W.: 'Compiling occam into FPGAs' (Abingdon EE&CS Books, 1991)
77 Yamada, A., Nishida, K., Sakurai, R., Kay, A., Nomura, T., and Kambe, T.: 'Hardware synthesis with the Bach system'. Proc. IEEE ISCAS, 1999
78 Frigo, J., Palmer, D., Gokhale, M., Popkin-Paine, M.: 'Gamma-ray pulsar detection using reconfigurable computing hardware'. Proc. Symp. on Field Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
79 Styles, H., and Luk, W.: 'Customising graphics applications: techniques and programming interface'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2000)
80 Simulink, http://www.mathworks.com
81 Hwang, J., Milne, B., Shirazi, N., and Stroomer, J.D.: 'System level tools for DSP in FPGAs', *Lect. Notes Comput. Sci.*, 2001, **2147**
82 Altera Corp., DSP Builder User Guide, Version 2.1.3 rev.1, July 2003
83 Lee, E.A., and Messerschmitt, D.G.: 'Static scheduling of synchronous data flow program for digital signal processing', *IEEE Trans. Comput.*, 1987, **36**, pp. 24–35
84 Constantinides, G.A., Cheung, P.Y.K., and Luk, W.: 'Optimum and heuristic synthesis of multiple wordlength architectures', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2003, **22**, (10), pp. 1432–1442
85 Constantinides, G.A., Cheung, P.Y.K., and Luk, W.: 'Synthesis and optimization of DSP algorithms' (Kluwer Academic, Dordrecht, 2004)
86 Constantinides, G.A., Cheung, P.Y.K., and Luk, W.: 'The multiple wordlength paradigm'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2001)
87 Constantinides, G.A., and Woeginger, G.J.: 'The complexity of multiple wordlength assignment', *Appl. Math. Lett.*, 2002, **15**, (2), pp. 137–140
88 Constantinides, G.A., Cheung, P.Y.K., and Luk, W.: 'Synthesis of saturation arithmetic architectures', *ACM Trans. Des. Autom. Electron. Syst.*, 2003, **8**, (3), pp. 334–354
89 Kum, K.-I., and Sung, W.: 'Combined word-length optimization and high-level synthesis of digital processing systems', *IEEE Trans. Comput. Aided Des.*, 2001, **20**, (8), pp. 921–930
90 Wadekar, S.A., and Parker, A.C.: 'Accuracy sensitive word-length selection for algorithm optimization'. Proc. Int. Conf. on Computer Design, 1998
91 Cantin, M.-A., Savaria, Y., and Lavoie, P.: 'An automatic word length determination method'. Proc. IEEE Int. Symp. on Circuits and Systems, 2001, pp. V-53–V-56
92 Constantinides, G.A., Cheung, P.Y.K., and Luk, W.: 'Optimum wordlength allocation'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2002)
93 Nayak, A., Haldar, M., Choudhary, A., and Banerjee, P.: 'Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs'. Proc. Design Automation and Test in Europe, 2001
94 Stephenson, M., Babb, J., and Amarasinghe, S.: 'Bitwidth analysis with application to silicon compilation'. Proc. SIGPLAN Programming Language Design and Implementation, June 2000
95 Cmar, R., Rijnders, L., Schaumont, P., Vernalde, S., and Bolsens, I.: 'A methodology and design environment for DSP ASIC fixed point refinement'. Proc. Design Automation and Test in Europe, 1999
96 Constantinides, G.A.: 'Perturbation analysis for word-length optimization'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
97 Abdul Gaffar, A., Mencer, O., Luk, W., Cheung, P.Y.K., and Shirazi, N.: 'Floating-point bitwidth analysis via automatic differentiation'. Proc. Int. Conf. on Field-Programmable Technology, IEEE, 2002
98 Abdul Gaffar, A., Mencer, O., Luk, W., and Cheung, P.Y.K.: 'Unifying bit-width optimisation for fixed-point and floating-point designs'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2004)
99 Cantin, M.-A., Savaria, Y., and Lavoie, P.: 'A comparison of automatic word length optimization procedures'. Proc. IEEE Int. Symp. on Circuits and Systems, 2002
100 Constantinides, G.A.: 'High level synthesis and word length optimization of digital signal processing systems'. PhD thesis, Imperial College London, 2001
101 Benedetti, A., and Perona, B.: 'Bit-width optimization for configurable DSP's by multi-interval analysis'. Proc. 34th Asilomar Conf. on Signals, Systems and Computers, 2000
102 Stephenson, M.W.: 'Bitwise: Optimizing bitwidths using data-range propagation'. Master's Thesis, Massachussets Institute of Technology, Dept. Electrical Engineering and Computer Science, May 2000
103 Keding, H., Willems, M., Coors, M., and Meyr, H.: 'FRIDGE: A fixed-point design and simulation environment'. Proc. Design Automation and Test in Europe, 1998
104 Willems, M., Bürsgens, V., Keding, H., Grotker, T., and Meyer, M.: 'System-level fixed-point design based on an interpolative approach', Proc. 34th Design Automation Conf., June 1997
105 Kum, K., and Sung, W.: 'Word-length optimization for high-level synthesis of digital signal processing systems'. Proc. IEEE Int. Workshop on Signal Processing Systems, 1998
106 Sung, W., and Kum, K.: 'Word-length determination and scaling software for a signal flow block diagram'. Proc. IEEE Int. Conf. on Acoustics Speech and Signal Processing, 1994
107 Sung, W., and Kum, K.: 'Simulation-based word-length optimization method for fixed-point digital signal processing systems', *IEEE Trans. Signal Process.*, 1995, **43**, (12), pp. 3087–3090
108 Ong, S., Kerkiz, N., Srijanto, B., Tan, C., Langston, M., Newport, D., and Bouldin, D.: 'Automatic mapping of multiple applications to multiple adaptive computing systems'. Proc. Int. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2001)
109 Thomas, D., and Luk, W.: 'A framework for development and distribution of hardware acceleration', *Proc. SPIE - Int. Soc. Opt. Eng.*, 2002, **4867**
110 Bohm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R., and Najjar, W.: 'Mapping a single assignment programming language to reconfigurable systems', *J. Supercomput.*, 2002, **21**, pp. 117–130
111 Damianou, N., Dulay, N., Lupu, E., and Sloman, M.: 'The Ponder policy specification language', *Lect. Notes Comput. Sci.*, 2001, **1995**
112 Lee, T.K., Yusuf, S., Luk, W., Sloman, M., Lupu, E., and Dulay, N.: 'Compiling policy descriptions into reconfigurable firewall processors'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2003)
113 Kulkarni, C., Brebner, G., and Schelle, G.: 'Mapping a domain specific language to a platform FPGA'. Proc. Design Automation Conf., 2004
114 Lee, T.K., Derbyshire, A., Luk, W., and Cheung, P.Y.K.: 'High-level language extensions for run-time reconfigurable systems'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2003
115 Shirazi, N., Luk, W., and Cheung, P.Y.K.: 'Framework and tools for run-time reconfigurable designs', *IEE Proc., Comput. Digit. Tech.*, 2000, **147**, pp. 147–152
116 Derbyshire, A., and Luk, W.: 'Compiling run-time parametrisable designs'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2002
117 Clark, D., and Hutchings, B.: 'The DISC programming environment'. Proc. Symp. on FPGAs for Custom Computing Machines (IEEE Computer Society Press, 1996)
118 Styles, H., and Luk, W.: 'Branch optimisation techniques for hardware compilation', *Lect. Notes Comput. Sci.*, 2003, **2778**
119 Kathail, V., Aditya, S., Schreiber, R., Ramakrishna Rau, B., Cronquist, D.C., and Sivaraman, M.: 'PICO: automatically designing custom computers', *Computer*, 2002, **35**, (9), pp. 39–47
120 Muggleton, S.H.: 'Inverse entailment and Progol', *New Gener. Comput.*, 1995, **13**
121 Peterson, J., O'Connor, B., and Athanas, P.: 'Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures'. Int. Symp. on FPGAs for Custom Computing Machines (IEEE Computer Society Press, 1996)
122 Duncan, A., Hendry, D., and Gray, P.: 'An overview of the COBRA-ABS high-level synthesis system for multi-FPGA systems'. Proc. IEEE Symposium on FPGAs for Custom Computing Machines (IEEE Computer Society Press, 1998)
123 Ast, A., Becker, J., Hartenstein, R., Kress, R., Reinig, H., and Schmidt, K.: 'Data-procedural languages for FPL-based machines', *Lect. Notes. Comput. Sci.*, 1994, **849**
124 Högl, H., Kugel, A., Ludvig, J., Männer, R., Noffz, K., Zoz, R., 'Enable++ a second-generation FPGA processor'. IEEE Symp. on FPGAs for Custom Computing Machines (IEEE Computer Society Press, 1995)
125 Callahan, T., and Wawrzynek, J.: 'Instruction-level parallelism for reconfigurable computing', *Lect. Notes Comput. Sci.*, 1998, **1482**
126 Gokhale, M., and Stone, J.: 'NAPA C: compiling for a hybrid RISC/FPGA architecture'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 1998)
127 Babb, J., Reinard, M., Andras, Moritz, C., Lee, W., Frank, M., Barwa, S., and Amarasinghe, S.: 'Parallelizing applications into silicon'. Proc. Symp. on FPGAs for Custom Computing Machines (IEEE Computer Society Press, 1999)
128 Ziegler, H., So, B., Hall, M., and Diniz, P.: 'Coarse-grain pipelining on multiple-FPGA architectures', IEEE Symp. on Field-Programmable Custom Computing Machines, 2002, pp. 77–88
129 Rissa, T., Luk, W., and Cheung, P.Y.K.: 'Automated combination of simulation and hardware prototyping'. Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (CSREA Press, 2004)
130 Bjesse, P., Claessen, K., Sheeran, M., and Singh, S., 'Lava: hardware design in Haskell'. Proc. ACM Int. Conf. on Functional Programming (ACM Press, 1998)
131 Singh, S., and Lillieroth, C.J.: 'Formal verification of reconfigurable cores'. Proc. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 1999)
132 Guo, S., and Luk, W.: 'An integrated system for developing regular array design', *J. Syst. Archit.*, 2001, **47**, pp. 315–337
133 Luk, W., and McKeever, S.W.: 'Pebble: a language for parametrised and reconfigurable hardware design', *Lect. Notes Comput. Sci.*, 1998, **1482**
134 McKeever, S.W., Luk, W., and Derbyshire, A.: 'Compiling hardware descriptions with relative placement information for parametrised libraries', *Lect. Notes Comput. Sci.*, 2002, **2517**
135 Todman, T., Coutinho, J.G.F., and Luk, W.: 'Customisable hardware compilation'. Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (CSREA Press, 2004)
136 Ou, J., and Prasanna, V.: 'PyGen: a MATLAB/Simulink based tool for synthesizing parameterized and energy efficient designs using FPGAs'. Proc. Int. Symp. on Field-Programmable Custom Computing Machines (IEEE Computer Society Press, 2004)