Compiling for Reconfigurable Computing: A Survey

JOÃO M. P. CARDOSO

Universidade do Porto

PEDRO C. DINIZ

Instituto Superior Técnico and INESC-ID

and

MARKUS WEINHARDT

Fachhochschule Osnabrück

Reconfigurable computing platforms offer the promise of substantially accelerating computations through the concurrent nature of hardware structures and the ability of these architectures for hardware customization. Effectively programming such reconfigurable architectures, however, is an extremely cumbersome and error-prone process, as it requires programmers to assume the role of hardware designers while mastering hardware description languages, thus limiting the acceptance and dissemination of this promising technology. To address this problem, researchers have developed numerous approaches at both the programming languages as well as the compilation levels, to offer high-level programming abstractions that would allow programmers to easily map applications to reconfigurable architectures. This survey describes the major research efforts on compilation techniques for reconfigurable computing architectures. The survey focuses on efforts that map computations written in imperative programming languages to reconfigurable architectures and identifies the main compilation and synthesis techniques used in this mapping.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; B.1.4 [**Control Structures and Microprogramming**]: Microprogram Design Aids—*Languages and compilers; optimization*; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis; optimization*; B.6.3 [**Logic Design**]: Design Aids—*Automatic synthesis; optimization*; B.7.1 [**Integrated Circuits**]: Types and Design Styles; D.3.4 [**Programming Languages**]: Processors—*Compilers; optimization*

General Terms: Languages, Design, Performance

Additional Key Words and Phrases: Compilation, hardware compilers, high-level synthesis, mapping methods, reconfigurable computing, FPGA, custom-computing platforms

This research was partially supported by FCT, the Portuguese Science Foundation, under research grants PTDC/EEA-ELC/71556/2006, PTDC/EEA-ELC/70272/2006, and POSI/CHS/48018/2002.

This work was performed while J. Cardoso was with Universidade do Algarve and later with IST/UTL.

Authors' addresses: J. M. P. Cardoso, Dep. de Eng. Informática, Faculdade de Engenharia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal; email: jmpc@acm.org; P. C. Diniz, INESC-ID, Rua Alves Redol n. 9, 1000-029 Lisboa, Portugal; email: pedro.diniz@ist.utl.pt; M. Weinhardt Fak. IuI, Postfach 1940, 49009 Osnabrück, Germany; email: mweinhardt@computer.org.

DOI 10.1145/1749603.1749604 http://doi.acm.org/10.1145/1749603.1749604

ACM Reference Format:
Cardoso, J. M. P., Diniz, P. C., and Weinhardt, M. 2010. Compiling for reconfigurable computing: A survey. ACM Comput. Surv. 42, 4, Article 13 (June 2010), 65 pages.
DOI = 10.1145/1749603.1749604, http://doi.acm.org/10.1145/1749603.1749604

1. INTRODUCTION

The main characteristic of reconfigurable computing platforms [Lysaght and Rosenstiel 2005; Gokhale and Graham 2005] is the presence of reconfigurable hardware (*reconfigware*), that is, hardware that can be reconfigured on-the-fly (i.e., dynamically and on-demand) to implement specific hardware structures. For example, a given signal processing application might require only 12-bit fixed-point precision arithmetic and use custom rounding modes [Shirazi et al. 1995], whereas other application codes might make intensive use of a 14-bit butterfly routing network used for fast parallel computation of a fast Fourier transform (FFT). In either case, a traditional processor does not have direct support for these structures, forcing programmers to use hand-coded routines to implement the basic operations. In the case of an application that requires a specific interconnection network, programmers must encode in a procedure the sequential evaluation of the data flow through the network. In all these examples, it can be highly desirable to develop dedicated hardware structures that can implement specific computations for performance, but also for other metrics such as energy.

During the past decade a large number of reconfigurable computing systems have been developed by the research community, which demonstrated the capability of achieving high performance for a selected set of applications [DeHon 2000; Hartenstein 2001; Hauck 1998]. Such systems combine microprocessors and *reconfigware* in order to take advantage of the strengths of both. An example of such hybrid architectures is the Napa 1000 where a RISC (reduced instruction set computer) processor is coupled with a programmable array in a coprocessor computing scheme [Gokhale and Stone 1998]. Other researchers have developed reconfigurable architectures based solely on commercially available field-programmable-gate-arrays (FPGAs) [Diniz et al. 2001] in which the FPGAs act as processing nodes of a large multiprocessor machine. Approaches using FPGAs are also able to accommodate on-chip microprocessors as *softcores* or *hardcores*. In yet another effort, researchers have developed dedicated reconfigurable architectures using as internal building blocks multiple functional units (FUs) such as adders and multipliers interconnected via programmable routing resources (e.g., the RaPiD architecture [Ebeling et al. 1995], or the XPP reconfigurable array [Baumgarte et al. 2003; XPP]).

Overall, reconfigurable computing architectures provide the capability for spatial, parallel, and specialized computation, and hence can outperform common computing systems in many applications. This type of computing effectively holds extensive parallelism and multiple flows of control, and can leverage the existent parallelism at several levels (operation, basic block, loop, function, etc.). Most of the existing reconfigurable computing systems have multiple on- and off-chip memories, with different sizes and accesses schemes, and some of them with parameterized facilities (e.g., data width). *Reconfigware* compilers can leverage the synergies of the reconfigurable architectures by exploting multiple flows of control, fine- and coarse-grained parallelism, customization, and by addressing the memory bandwidth bottleneck by caching and distributing data.

Programming *reconfigware* is a very challenging task, as current approaches do not rely on the familiar sequential programming paradigm. Instead, programmers must assume the role of hardware designers, either by learning a new hardware-oriented programming language such as VHDL or Verilog or by coding in particular programming styles that are geared towards hardware synthesis. Many authors believe that automatic compilation from standard software programming languages is vital for the success of reconfigurable computing systems, as the complex design expertise required for developing a contemporary hardware system is too much for the typical user (e.g., embedded systems programmers) to handle.

A current, and pressing challenge in this area is the establishment of effective *reconfigware* compilers, which would help the programmer to develop efficient *reconfigware* implementations without the need to be involved in complex and low-level hardware programming. Although in the context of programmable logic devices,[1] mature design tools exist for logic synthesis and for placement and routing, there is a lack of robust integrated tools that take traditional sequential programs and automatically map them to reconfigurable computing architectures. In addition, high-level synthesis (HLS)[2] tools have mostly been developed for ASICs (application-specific integrated circuits), and neither wield the special characteristics of the reconfigurable architectures nor the required high-level abstraction. Those tools are typically based on resource-sharing schemes [Gajski et al. 1992] that target the layout flexibility of ASICs. As such, they are, typically, much less efficient when considering the predefined logic cell architecture and limited routing resources of fine-grained reconfigurable processing units (RPUs), for example, FPGAs, where resource sharing is inadequate for most operations. As a consequence, the inherent characteristics of *reconfigware* require specialized (architecture-driven) compilation efforts.

This survey describes representative research work on compilation for reconfigurable computing platforms. It does not, however, address the work on mapping methods using programming languages with hardware-oriented semantics and structures, as well as efforts on using software languages for describing circuits and circuit generator environments [Mencer et al. 2001]. Although the use of software languages (mainly Java and C++) adds new capabilities to hardware description language (HDL) approaches, its effective use resembles the definition of the architecture at register transfer level (RTL). While these approaches may not be considered as high-level programming flows, they can, however, be part of the back-end phase of most compilers described hereafter. Given the many perils of the imperative programming paradigm, a number of researchers have advocated the use of alternative programming models. Although efforts have been made for the hardware compilation of declarative programs (e.g., in Ruby [Luk and Wu 1994]), functional languages (e.g., Lava, Haskell [Bjesse et al. 1998] and SAFL [Sharp and Mycroft 2001]), and synchronous programming languages such as Esterel (e.g., Edwards [2002]), they have been limited in scope, and may gain wide acceptance only when such programming models permit more efficient solutions for a wide range of problems than the ones commonly accomplished using traditional languages (such as C). Due to the very limited acceptance of these programming models in both industry and academia, this survey does not address them. Alternatively, there has also been extensive work in the context of hardware/software partitioning in the context of compilation and synthesis for ASICs in the codesign community, which we do not explicitly address here. Readers interested in this topic may consult prior work by the codesign community [Micheli and Gupta 1997] and efforts considering *reconfigware* (e.g., Li et al. [2000]).

[1]Note that here we make no distinction between FPGAs and PLDs.
[2]There is no distinction among the terms *high-level synthesis*, *architectural synthesis*, and *behavioral synthesis*.

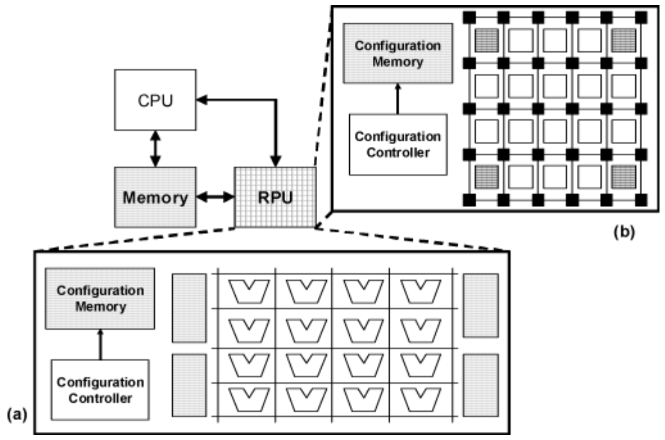


Fig. 1. Typical reconfigurable computing platforms: (a) coarse-grained RPU; (b) fine-grained RPU.

Complementary to this survey, the reader can find other scholarly publications focusing on specific features of reconfigurable computing platforms or on software tools for developing FPGA-based designs. A review of HLS for dynamically reconfigurable FPGAs can be found in Zhang and Ng [2000]. A summary of results of a decade of research in reconfigurable computing is presented by Hartenstein [2001]. A survey of reconfigurable computing for digital signal-processing applications is given in Tessier and Burleson [2001]. Compton and Hauck [2002] present a survey on systems and software tools for reconfigurable computing, and Todman et al. [2005] focus on architectures and design methods.

This survey is organized as follows. In Section 2 we briefly describe current architectures for reconfigurable computing platforms. In Section 3 we present an overview of typical compilation flows. In Section 4 we describe a set of representative high-level code transformations, and in Section 5 present the most representative mapping techniques. In Section 6 we describe a set of representative high-level compilaton efforts for reconfigurable architectures, highlighting the high-level techniques and code transformations. Lastly, in Section 7 we conclude with some final remarks.

2. RECONFIGURABLE ARCHITECTURES

Reconfigurable computing systems are typically based on reconfigurable processing units (RPUs) acting as coprocessor units and coupled to a host system, as depicted in Figure 1. The type of interconnection between the RPUs and the host system, as well as the granularity of the RPUs, lead to a wide variety of possible reconfigurable architectures. While some architectures naturally facilitate the mapping of specific aspects of computations to them, no single dominant RPU solution has emerged for all domains of applications. As a consequence, many research efforts have focused on the development of specific reconfigurable architectures. The Xputer architecture [Hartenstein et al. 1996] was one of the first efforts to address coarse-grained reconfigurable architectures. Internally, it consists of one reconfigurable data path array (rDPA) organized as a rectangular array of 32-bit arithmetic-logic units (ALUs). Each ALU can be reconfigured to execute some operators of the C language. The processing elements are mesh-connected via three levels of interconnection, namely nearest neighbors, row/column back-buses, and a global bus.

RaPiD [Ebeling et al. 1995] is a coarse-grained field-programmable architecture composed of multiple FUs (functional units), such as ALUs, multipliers, registers, and RAM blocks. These units are organized linearly over a bus, and communicate through registers in a pipeline fashion. RaPiD allows the user to construct custom application-specific architectures in a runtime reconfigurable way, that is, as the application is being executed. In RaPiD, data is streamed in directly from external memories or input devices (e.g., sensors). Programmed controllers generate a small instruction stream that is decoded as it flows in parallel with the datapath. In RaPiD, small sets of data and intermediate results are stored locally in registers and small RAMs close to their destination FUs. A programmable interconnect is configured to forward data between specific FUs on a per-application basis.

A radically distinct architecture is the PipeRench [Goldstein et al. 1999], an architecture geared solely to pipelined computations with a virtually unlimited number of pipeline stages (*hardware stripes*). Each *stripe* has an array of processing elements (PEs). Each PE consists of an ALU and a pass register file. The PipeRench is tailored to compute unrolled computations of streams and to the chaining of execution and configuration. In PipeRench, computations are scheduled on virtual stripes and the architecture swaps the configuration in and out for each of the stripes on demand.

The ADRES (architecture for dynamically reconfigurable embedded systems) [Mei et al. 2003] consists of a 2-D reconfigurable architecture of coarse-grained 32-bit PEs. The architecture tightly couples a VLIW (very long instruction word) processor and a reconfigurable array. The reconfigurable array is responsible for exploiting parallelism through highly-pipelined kernels, and the VLIW is responsible for exploiting the ILP (instruction-level parallelism) in nonkernel parts of the application. Besides the multiplexers for input operands, the output registers and the configuration buffers, each PE includes an FU and a register file. The FUs use predicates for enabling their execution (predicate signals are routed as operand interconnections between FUs).

The MIT RAW machine [Taylor et al. 2002] is a coarse-grained two-dimensional mesh architecture in which each computing core (*tile*) has a RISC processing element with registers and an ALU, local memory blocks (one for data, one for instructions, and another to program the switch network) and programmable communication channels. A controller is used to program a switch router. Initial versions of the RAW architecture have considered reconfigurable logic in each tile, closely coupled to the RISC [Waingold et al. 1997].

The Garp architecture [Callahan et al. 2000] integrates a MIPS core with an RPU to be used as an accelerator in a coprocessor integrated model. The RPU is organized as a 2-D array of CLBs (configurable logic blocks) interconnected by programmable interconnections. Each row has a section dedicated to memory interfacing and control and uses a fixed clock. The reconfigurable array has direct access to memory for fetching either data or configuration words, hence avoiding both data and reconfiguration bottlenecks.

The MorphoSys architecture [Singh et al. 2000] is another reconfigurable computing platform integrating a RISC processor core, a reconfigurable array of cells (8×8), and a memory interface unit. The MorphoSys RPU has coarse-grained granularity, as each cell has an ALU (28-bit fixed and capable of one of 25 functions), a multiplier (16×12 bits), and a register file (composed of four 16-bit registers).

Finally, the SCORE (stream computations organized for reconfigurable execution) system [Caspi et al. 2000; DeHon et al. 2006] uses distributed memory modules with attached reconfigurable logic, and explicitly supports a stream-oriented computational model between computation in resident pages and buffers managed by a runtime operating system manager [Markovskiy et al. 2002].

In general, the type of coupling of the RPUs to the existent computing system has a significant impact on the communication cost, and can be classified into the three groups listed below, in order of decreasing communication costs:

—RPUs coupled to the host bus: The connection between RPUs and the host processor is accomplished via a local or a system bus. Examples of reconfigurable computing platforms connecting to the host via a bus are HOT-I, II [Nisbet and Guccione 1997]; Xputer [Hartenstein et al. 1996]; SPLASH [Gokhale et al. 1990]; ArMem [Raimbault et al. 1993]; Teramac [Amerson et al. 1995]; DECPerLe-1 [Moll et al. 1995]; Transmogrifier [Lewis et al. 1998]; RAW [Waingold et al. 1997]; and Spyder [Iseli and Sanchez 1993].

—RPUs coupled like coprocessors: Here the RPU can be tightly coupled to the host processor, but has autonomous execution and access to the system memory. In most architectures, when the RPU is executing the host processor is in stall mode. Examples of such platforms are the NAPA [Gokhale and Stone 1998]; REMARC [Miyamori and Olukotun 1998]; Garp [Hauser and Wawrzynek 1997]; PipeRench [Goldstein et al. 2000]; RaPiD [Ebeling et al. 1995]; and MorphoSys [Singh et al. 2000].

—RPUs acting like an extended datapath of the processor and without autonomous execution. The execution of the RPU is controlled by special opcodes of the host processor instruction-set. These datapath extensions are called reconfigurable functional units. Examples of such platforms include the Chimaera [Ye et al. 2000b]; PRISC [Razdan 1994; Razdan and Smith 1994]; OneChip [Witting and Chow 1996]; and ConCISe [Kastrup et al. 1999].

Devices integrating one microprocessor with reconfigurable logic have been commercialized by companies such as Triscend [2000], Chameleon [Salefski and Caglar 2001]; and Altera [Altera Inc.],[3] Of the major players, only Xilinx [Xilinx Inc.] includes at the moment an FPGA with one or more *hardcore* microprocessors, the IBM PowerPC processor.

S*oftcores* based on von-Neumann architectures are widely used. The best known examples of such *softcore* microprocessors are the Nios (I and II) from Altera and the MicroBlaze from Xilinx. They assist in the development of applications, since a programming flow from known software languages (typically C) is ensured by using mature software compilers (typically the GNU gcc). There are also examples of companies delivering configurable processor *softcores*, specialized to domain-specific applications. One such example is the *softcore* approach addressed by Stretch [Stretch Inc], which is based on the Tensilica Xtensa RISC (a configurable processor) [Gonzalez 2000; Tensilica] and an extensible instruction set fabric.

Most research efforts use FPGAs and directly exploit its low-level processing elements. Typically, these reconfigurable elements or configurable logic blocks (CLBs) consist of a flip-flop and a function generator that implements a *boolean* function of up to a specific number of variables. By organizing these CLBs, a designer can implement virtually any digital circuit. Despite its flexibility, these fine-grained reconfigurable elements have been shown to be inefficient in time and area for certain classes of problems [Hartenstein 2001, 1997]. Other research efforts use ALUs as the basic *reconfigware* primitives. In some cases, this approach provides more efficient solutions, but limits system flexibility. Contemporary FPGA architectures (e.g., Virtex-II [Xilinx 2001] or Stratix [Altera 2002]) deliver distributed multiplier and memory blocks in an attempt to maintain the fine-grained flavor of its fabric while supporting other, coarser-grained, classes of architectures commonly used in data-intensive computations.

[3]Altera has discontinued the Excalibur FPGA devices, which integrated an on-chip *hardcore* ARM processor.

Table I. Examples of Architectures with Different Granularities

Cell Granularity	Canonical Operations	Examples of Devices	Shape	Cell Type	Bit-width
fine (wire)	Logic functions of 2-5 bits	FPGAs, Xilinx Virtex and Altera Stratix	2D array	LUT, multiplexer, register MULT, DSP blocks RAM Blocks	2-5 1 18 customizable
		DPGA [DeHon 1996]	2D array	LUT	4
coarse (operand)	ALU operations of 4-32 bits	PipeRench [Goldstein et al. 1999]	2D array	ALU + Pass Register File	2-32 (parametrizable before fabrication)
		rDPA/KressArray (Xputer) [Hartenstein et al. 1996; Kress 1996]	2D array	ALU + register	32 (parametrizable before fabrication)
		MATRIX [Mirsky and DeHon 1996]	2D array	ALU with multiplier + memory	8
		RaPiD [Ebeling et al. 1995]	1D array	ALU, multipliers, registers, RAM	16
		XPP [Baumgarte et al. 2003]	2D array	ALU with multiplier, RAM	4-32 (parametrizable before fabrication)
		MorphoSys [Singh et al. 2000]	2D array	ALU + multiplier + register file	28
		ADRES [Mei et al. 2003]	2D array	ALU + multiplier + register file	32
		ARRIX FPOAs [MathStar]	2D array	ALU + MAC + register file	16
mix-coarse	Sequence of assembly instructions	RAW [Taylor et al. 2002]	2D array	RISC + memory + switch network	32

When compared to native coarse-grained architectures, the approaches using an operation-level abstraction layer over the physical fine granularity of an FPGA have more flexibility, but require longer compilation times and need more configuration data (and longer reconfiguration times). This abstraction layer can be provided by a library of relatively placed macros (RPMs) with direct correspondence to the operators of the high-level software programming languages (arithmetic, logic, and memory access). This library can contain more than one implementation of the same operator, representing different tradeoffs among area, latency, and configuration time. Furthermore, the use of a library decreases the overall compilation time, enables more accurate estimations, and is already a proven concept in HLS systems. A second option for an abstraction layer is the use of hardware templates with specific architecture and parametrizable and/or programmable features. An extreme case of this, are the configurable *softcore* processors. Approaches using a layer of abstraction implemented by architecture templates are the dynamic processor cores implemented in fine-grained RPUs (e.g., FPGAs), such as the DISC (dynamic instruction set computer) [Wirthlin 1995]. This approach allows the dynamic reconfiguration of dedicated hardware units when a new instruction appears whose correspondent unit has not yet been configured.

Table I summarizes the various efforts on reconfigurable architectures, focusing on their granularity and main characteristics, classified into three broad categories, as follows:

—*Fine-grained*: The use of the RPUs with fine-grained elements to define specialized datapath and control units.

—*Coarse-grained*: The RPUs in these architectures are often designated field-programmable ALU arrays (FPAAs), as they have ALUs as reconfigurable blocks. These architecture are especially adapted to computations over typical data-widths (e.g., 16, 32 bits).

—*Mix-coarse-grained*: These include architectures consisting of tiles that usually integrate one processor and possibly one RPU on each tile. Programmable interconnect resources are responsible for defining the connections between tiles.

The PipeRench [Goldstein et al. 2000, 1999] (see Table I) can be classified as a fine- or as a coarse-grained architecture, as the parametrizable bit-width of each processing element can be set before fabrication within a range from a few bits to a larger number of bits typically used by coarse-grained architectures. Bit-widths from 2 to 32 bits have been used [Goldstein et al. 1999]. The XPP [Baumgarte et al. 2003] (see Table I) is a commercial coarse-grained device; Hartenstein [2001] presents additional coarse-grained architectures.

With RPUs that allow partial reconfiguration it is possible to reconfigure regions of the RPU while others are executing. Despite its flexibility, partial reconfiguration is not widely available in today's FPGAs. With acceptable reconfiguration time, dynamic (runtime) reconfiguration might be an important feature that compilers should address. Reconfiguration approaches based on layers of on-chip configuration planes selected by context switching have been addressed by, for example, DeHon [1996] and Fujii et al. [1999]. These techniques allow time-sharing of RPUs over computation. Overlapping reconfiguration with computation would ultimately allow us to mitigate, or even to eliminate, the reconfiguration time overhead.

3. OVERVIEW OF COMPILATION FLOWS

This section presents an overview of compilation flows that target reconfigurable computing architectures. A generic compilation flow is depicted in Figure 2. As with traditional compilation, it first uses a front-end to decouple the specific aspects of the input programming language (e.g., its syntax) from an intermediate representation (IR). Next, the middle-end of the compiler applies a variety of architecture-neutral transformations (e.g., constant folding, subexpression elimination, elimination of redundant memory accesses), and architecture-driven transformations (e.g., loop transformations, bit-width narrowing) to expose specialized data types and operations and fine-grained as well as coarse-grained parallelism opportunities. While some transformations exploit high-level architectural aspects such as the existence of multiple memories to increase the availability of data, other transformations exploit the ability to customize specific functional units in an RPU (reconfigurable processing unit), thereby directly implementing high-level instructions at the hardware level. The latter includes the customization of arithmetic units to directly support fixed-point operations using nonstandard bit-width formats that are representations or rounding modes. Finally, the flow includes a back-end. This compiler stage schedules macro-operations and instructions, generates the definition of a concrete architecture by the synthesis of its datapaths and FUs, and performs the low-level steps of mapping and placement and routing (P&R), if applicable. Some of these steps may be accomplished by integrating commercial synthesis tools in the flow, in many cases using target-specific component library modules.

As many of the reconfigurable computing platforms include a traditional processor, partitioning the computations between the processor and an RPU is a key aspect

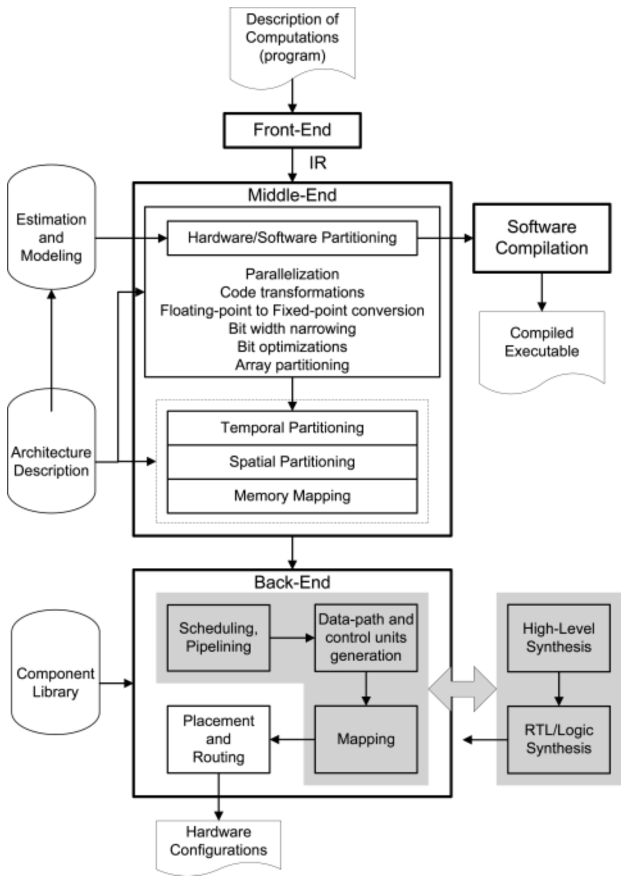


Fig. 2. Generic compilation flow for reconfigurable computing platforms.

of the compilation flow. This partitioning, and the corresponding data partitioning, is guided by specific performance metrics, such as the estimated execution time and power dissipation. As a result, the original computations are partitioned into a software specification component and a *reconfigware* component. The software component is then compiled onto a target processor using a traditional or native compiler for that specific processor. The *reconfigware* components are mapped to the RPU by, typically, translating the correspondent computations to representations accepted by hardware compilers or synthesis tools. As part of this partitioning, additional instructions to synchronize the communication of data between the processor and the RPU are required. This partitioning process is even more complex if the target architecture consists of multiple processors or the communication schemes between devices or cores can be defined in the compilation step, as is the case when targeting complex embedded systems.

After the *reconfigware*/software partitioning process, the middle-end engages in temporal and spatial partitioning for the portions of the computations that are mapped to the RPU. Temporal partitioning refers to the splitting of computations in sections to be executed by time-sharing the RPU, whereas spatial partitioning refers to the splitting of the computations in sections to be executed on multiple RPUs, that is, in a more traditional parallel computing fashion. The mapping of the data structures onto memory resources on the reconfigurable computing platform can also be done at this phase of the mapping.

Many compilation use commercially available HLS tools for the final mapping steps targeting the RPU. Other approaches have been developed to map the computations onto the RPU resources as a vehicle for exploiting specific techniques enabled by specific architectural features of its target RPUs. Loop pipelining, scheduling, and the generation of hardware structures for the datapath and for the control unit are included in this category. Note, however, that these efforts depend heavily on the granularity of the RPU, and therefore on its native model of computation. Finally, to generate the *reconfigware* configurations (that is, binaries to program the RPUs), compilers perform mapping, placement, and routing. Compilers targeting commercial FPGAs typically accomplish these tasks using vendor-specific tools.

Estimations (e.g. of the required hardware resources or execution latencies) are required by some of the compilation steps in this flow to select among transformations and optimizations. Some of these auxiliary techniques are very important for dealing with the many degrees of freedom that reconfigurable architectures expose. A compiler should engage in sophisticated design space exploration (DSE) approaches to derive high-quality *reconfigware* implementations.

3.1. Programming Languages and Execution Models

It is widely believed that the major barrier to adoption of promising reconfigurable technology is the lack of adequate programming flows that offer a higher level of abstraction than currently provided by available HDLs [Babb et al. 1999]. Tools supporting high-level programming specifications would accelerate tremendously the development cycle of reconfigurable systems and facilitate the migration of already developed algorithms to these systems—a key aspect for their widespread acceptance.

The main obstacle to offering a high-level programming abstraction, such as the imperative programming model of widely popular languages like C or Java, lies in the semantic gap between this imperative model and the explicitly concurrent models used to program hardware devices. Common hardware description languages such as VHDL or Verilog use an execution model based on communicating sequential processes (CSP) [Hoare 1978], and thus are far from the imperative models. Hence compilation of programs from the imperative paradigm to hardware has to bridge this semantic gap by automatically extracting as much concurrency as possible, or simply relying on library implementations, where the notions of concurrent execution have been crystallized by library developers and openly publicized in application programming interfaces (APIs).

This semantic gap prompted the development of a wide range of approaches from the programming model perspective covering a wide spectrum of solutions, ranging from the easier approach, where the input language already offers a concurrent execution model close to the CSP model, to the harder approach of uncovering concurrency automatically from traditional imperative languages.

The extraction of concurrency has been a long-standing, and notoriously hard, research problem for the compiler and parallel computing communities. Constructs such as pointer manipulation in languages such as C or C++ hinder static analyses techniques from achieving a significant number of program transformations in the synthesis process [Séméria et al. 2001]. Compilation support for object-oriented mechanisms and dynamic data structures (e.g., memory allocation of linked lists) also requires advanced compilation analyses in the context of hardware synthesis [Jong et al. 1995; Radetzki 2000]. Alternative imperative execution models, such as languages with explicit support for data streaming (e.g., Streams-C [Gokhale et al. 2000b]), alleviate some of the data disambiguation problems and substantially improve the effectiveness of the mapping to reconfigurable systems, as they are especially suited for data streaming computations and for concurrent execution. Intraprocess concurrency, however, is still limited by the ability of the compiler to uncover concurrency from sequential statements.

At the other end of the spectrum, there are the approaches that use languages with explicit mechanisms for specifying concurrency. In this class there is a wealth of efforts, ranging from languages that expose concurrency at the operation level (e.g., Handel-C [Page 1996]) to tasks (e.g., Mitrion-C [Mitrionics]) or threads (e.g., Java threads [Tripp et al. 2002]).

Orthogonal to general-purpose language efforts, others have focused on the definition of languages with constructs for specific types of applications. SA-C [Böhm et al. 2001] is a language with single-assignment semantics geared toward image processing applications which has a number of features facilitate translation to hardware, namely, custom bit-width numerical representations and window operators. The language semantic effectively relaxes the order in which operations can be carried out and allows the compiler to use a set of predefined library components to implement them very efficiently. Other authors have developed their own target-specific languages. The RaPiD-C [Cronquist et al. 1998] and the DIL [Budiu and Goldstein 1999] languages were developed specifically for pipeline-centered execution models supported by specific target architectures. While these languages allow programmers to close the semantic gap between high-level programming abstractions and the low-level implementation details, they will probably ultimately serve as intermediate languages which a compiler tool can use when mapping higher level abstraction languages to these reconfigurable architectures.

The indisputable popularity of MATLAB [Mathworks], as a domain-specific language for image/signal processing and control applications, made it a language of choice when mapping to hardware computations in these domains. The matrix-based data model makes MATLAB very amenable to compiler analyses, in particular array-based data-dependence techniques. The lack of strong types, a very flexible language feature, requires that effective compilation must rely heavily on type and shape inference (see, e.g., Haldar et al. [2001a]), potentially limiting the usefulness of more traditional analyses.

Finally, there have also been efforts at using graphical programming environments such as the Cantata environment, used in the CHAMPION project [Ong et al. 2001], and the proprietary language Viva [Starbridge-Systems]. Essentially, these graphical systems allow the concurrency to be exposed at the task level, and are thus similar in spirit to the task-based concurrent descriptions offered by the CSP-like languages.

3.2. Intermediate Representations

Given the inherently parallel nature of reconfigurable architectures, where multiple threads of control can operate concurrently over distinct data items, the intermediate representation (IR) of a compiler for these architectures should explicitly represent this notion of concurrency. The IR should enable, rather than hamper, transformations that take advantage of specific features of the target reconfigurable architecture.

A natural IR explicitly represents control and data dependences as in the traditional control/dataflow graph (CDFG) [Gajski et al. 1992]. This representation uses the control-flow structure of the input algorithm and embeds operations in basic blocks. For each basic block, a dataflow graph (DFG) which is later mapped to the RPU is constructed. Note that aliases (arising from array or pointer accesses or from procedure calls) hinder the construction of DFGs. Concurrent execution of basic blocks is limited by the ability of a compiler to uncover data dependences across blocks. This limitation is even more noticeable when the available parallelism through multiple flows of control can be supported by the target architecture. The hyperblock representation [Mahlke et al. 1992] prioritizes regions of the control flow graph (CFG) to be considered as a whole, thus increasing the amount of available concurrency in the representation [Callahan and Wawrzynek 1998], which can be used by a fine-grained scheduler to enlarge the number of operations considered at the same time [Li et al. 2000].

A hierarchical representation, the hierarchical task graph (HTG) [Girkar and Polychronopoulos 1992], was explored in the context of HLS when targeting ASICs (e.g., Gupta et al. [2001]). This representation can efficiently represent the program structure in a hierarchical fashion and explicitly represent functional parallelism, from the imperative programming language. The HTG combined with a global DFG provided with program decision logic [August et al. 1999] seems to be an efficient intermediate model to represent parallelism at various levels. For showing multiple flows of control it is possible to combine information from the data-dependence graph (DDG) and the control-dependence graph (CDG) [Cytron et al. 1991], thus allowing a compiler to adjust the granularity of its data and computation partition with the target architecture characteristics.

In the co-synthesis and HLS community the use of task graphs is very common. An example is the unified specification model (USM) representation [Ouaiss et al. 1998b]. The USM represents task- and operation-level control and dataflow in a hierarchical fashion. At the task level, the USM captures accesses of each task to each data set by using edges and nodes representing dataflow and data sets, respectively, and is thus an interesting representation when mapping data sets to memories.

Regardless of the intermediate representation that exposes the available concurrency of the input program specification, there is still no clear migration path from more traditional imperative programming languages to such hardware-oriented representations. This mapping fundamentally relies on the ability of a compilation tool to uncover the data dependences in the input computation descriptions. In this context, languages based on the CSP abstractions are more amenable to this representation mapping than languages with state-full imperative paradigms such as C or C++.

3.3. Summary

There have been many different attempts, reflecting very distinct perspectives, in addressing the problem of mapping computations expressed in high-level programming languages to reconfigurable architectures. To date, no programming model, high-level language or intermediate representation has emerged as a widely accepted programming and compilation infrastructure paradigm that would allow compilers to automatically map computations described in high-level programming languages to reconfigurable systems. Important aspects such as the ability to map an existing software code base to such a language and to facilitate the porting of library codes are likely to play key roles in making reconfigurable systems attractive for the average programmer.

4. CODE TRANSFORMATIONS

The ability to customize the hardware implementation to a specific computation increases the applicability of many classical code transformations, which is well documented in the compiler literature (e.g., Muchnick [1997]). While in many cases these transformations can simply lead to implementations with lower execution times, they also enable implementations that consume less hardware resources, given the ability for fine-grained *reconfigware* to implement specialized circuits.

We now describe in detail key code transformations suitable for reconfigurable computing platforms. We distinguish between transformations that are very low-level, (at the bit-level) and can therefore exploit the configurability of very fine-grained reconfigurable devices such as FPGAs, and more instruction-level transformations (including loop-level transformations) that are more suitable for mapping computations to coarser-grained reconfigurable architectures.

4.1. Bit-Level Transformations

This section describes three common operation specializations used when compiling to fine-grained reconfigurable architectures, namely bit-width narrowing, bit-optimizations, and conversion from floating- to fixed-point data types.

4.1.1. Bit-width Narrowing. In many programs, the precision and range of the numeric data types are over-defined, that is, the necessary bit-widths to store the actual values are clearly smaller than the ones supported natively by traditional architectures [Caspi 2000; Stefanovic and Martonosi 2000]. While this fact might not affect performance when targeting a microprocessor or a coarse-grained RPU (reconfigurable processing unit), it becomes an aspect to pay particular attention to when targeting specialized FUs in fine-grained reconfigurable architectures. In such cases, the implementation size and/or the delay of an FU with the required bit-width can be substantially reduced. Bit-width inference and analysis is thus an important technique when compiling for these fine-grained architectures, especially when compilation from programming languages with limited support for discrete primitive types, such as the typical software programming languages, is addressed. There are examples of languages with the bit-width declaration capability, such as Valen-C [Inoue et al. 1998] a language based on C, which was designed having in mind the targeting of embedded systems based on core processors with parameterized data-width; other examples of languages with support for bit-width declarations are Napa-C, Stream-C, and DIL.

Bit-width narrowing can be static, profiling-based, dynamic or both. Static analyses are faster but need to be more conservative. Profiling-based analyses can have long runtimes, can reduce the bit-widths even more, but are heavily dependent on the input data. One of the first approaches using static bit-width analysis for mapping computations to *reconfigware* described in the literature is the approach presented by Razdan [1994] and Razdan and Smith [1994] in the context of the compilation for the PRISC architecture. This approach only considers static bit-width analysis for acyclic computations.

Budiu et al. [2000] describe *BitValue*, a more advanced static bit-width dataflow analysis that is able to deal with loops. The analysis aims to find the bit value for each operand/result. Each bit can have a definite value, either 0 or 1, an unknown value, or can be represented as a "don't care". They formulate *BitValue* as a dataflow analysis problem with forward, backward, and mixing steps. The dataflow analysis to propagate those possible bit values based on propagation functions dependent on the operation being analyzed. The backward analysis to propagate "don't care" values, and the forward analysis to propagate both defined and undefined values. Forward and backward analyses are done iteratively through bit-vector representations until a fixed-point is reached.

Another static analysis technique focuses on the propagation of value ranges that a variable may have during execution, and has been used in the context of compilation to *reconfigware* by Stephenson et al. [2000]. This analysis can determine narrower bit-widths requirements than previous analyses, and uses auxiliary information, such as the array dimension, to make a bound in the variables indexing array elements. However, using value ranges does not permit extraction of information about bits except at the extremities (i.e., least significant bits and most significant bits) of the data.

Some authors claim that static analysis is sometimes excessively conservative and thus use offline profiling-based analyses to determine the bit-width of variables. However, these analyses often incur long runtimes, as the program must be executed once or several times, and strongly depend on the dataset used. Thus, imprecision may occur in some implementations when data sets produce overflows or underflows not exposed in the profiling. To circumvent this problem, researchers have proposed offline, program execution-based, analyses that do not need data sets, but may incur long execution times [Ogawa et al. 1999].

A precision analysis specific to dynamic fine-grained reconfigurable architectures is proposed by Bondalapati and Prasanna [1999]. They use static or profiling-based inference of the sufficient bit-widths for each iteration of a loop. Then, specialized datapaths can be used for executing particular iteration ranges of a loop by exploiting dynamic reconfiguration.

Implementations capable of dynamically acquiring sufficient bit-widths to perform an operation correctly have been used in the context of the reduction of power dissipation in processor architectures [Brooks and Martonosi 1999]. Such techniques can be used in fine-grained architectures to reduce the delay of FUs when data arriving at an FU does not need all the bit-width statically allocated. An advanced use of dynamic bit-with inference (although to the best of our knowledge has not been researched) could be used to specialize, during runtime, the datapath according to the precisions needed.

4.1.2. Bit-Optimizations.

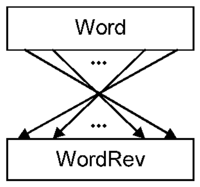
Applying bit-width narrowing and bit-optimization to programs with intensive bit-manipulations (mask, shift by constants, concatenation, etc.) is likely to lead to significant improvements by saving resources and decreasing execution times. Bit-optimizations are frequently done in logic synthesis using binary decision diagram (BDD) representations [Micheli 1994]. This representation is potentially time- and memory-consuming, and thus is not very effective when used in large examples (note that a small DFG of a program can have a large logic graph). Because of this, some compilers use other techniques aimed at reducing the logic needed. Such techniques operate mainly on the DFG with propagation of constants at the bit level using, for example, the BitValue propagation technique referred to above.

The DIL is an example of a compiler that considers some optimizations at the bit level [Budiu and Goldstein 1999]. The DIL language has some constructs that facilitate bit-optimizations also. For instance, the statement `c=(a.b)[7,0]`, where `b` is an 8-bit variable, the operator "." represents concatenation, and square brackets represent the bit-range of the expression assigned to `c`, simplifies to `c=b[7,0]` and permits us to have wires only between the 8 least significant bits of `b` directly connected to `c`. The previous statement in DIL is equivalent to the C statement: `c=(0xff)&((a<<8)|((0xff)&b))`, which requires more aggressive analysis to show the optimized circuit.

Aggressive bit optimizations should determine that the implementation of some statements in hardware do not need hardware resources, as is the case for `if((ans & 0x8000)==0x8000)`. Some bit-level analysis and optimizations can determine, after loop unrolling, that a fine-grained hardware implementation of the body of the function shown in Figure 3(a) only requires wire interconnections as depicted in Figure 3(b).

```
int Reverse(int Word) {
    int WordRev = 0;
        for(int i=0; i<32; i++) {
            WordRev |= (Word & 1);
            WordRev << 1;
            Word >> 1;
        }
    return WordRev;
}
```

(a)



(b)

Fig. 3. Bit reversing (32 bit word): (a) coded in a software programming language; (b) implemented in hardware.

4.1.3. Conversion from Floating- to Fixed-Point Representations. Floating-point data representations are a commonly used format to represent real numbers in almost all computing domains. However, the costs associated with arithmetic operations in floating-point format are much higher than using an integer representation in terms of logic gates and number of clock cycles. An alternative representation, commonly used in DSP (digital signal processing) algorithm implementations, is the fixed-point data type, which allows the arithmetic operations to be done by integer FUs with the requirement of scaling factors (the binary point is implicitly encoded as shifts), implemented by shifting the operands and/or the result of the operations. It is common to see algorithms that have been developed mostly on computers supporting floating-point data types. In some cases the tradeoff between the precision and efficiency of a computation in fixed-point format is acceptable given the substantial reduction in cost. However, each time these algorithms are ported to architectures where floating-point is not natively supported, the designer faces an error-prone, tedious, and time-consuming conversion from floating- to fixed-point representations.

This floating- to fixed-point conversion has been the focus of several research projects that compile C programs onto DSPs [Aamodt and Chow 2000; Kum et al. 2000; Willems et al. 1997]. Most approaches reported in the literature use profiling and/or user annotations to assist in the automatic phases of the conversion. It is common that whenever the tools are unable to find a translation for a specific variable, they ask for the help of the user. In the context of ASIC synthesis, some companies have already presented tools to convert from floating- to fixed-point, usually assisted by profiling steps; for example, the CoCentric^{TM} tool [Synopsys 2000] uses a design flow starting from the SystemC language. The conversion is based on an interpolation mechanism that helps the designer in the conversion of most variables in a program.

Floating- to fixed-point conversions have been the focus of some research groups when compiling to fine-grained reconfigurable architectures (e.g., FPGAs). Leong et al. [1999] show a conversion method based on profiling and the minimization of a cost function. This method assigns a unique fixed-point representation to each variable in the source code. Although the approach does not deal with loops in the C code, it highlights the importance of the conversion. Nayak et al. [2001b] describe an automatic approach, which divides the conversion into two main steps. The first step aims at finding the minimum number of bits of the integer part of the representation using value range propagation analysis (forward and backward), with similarities to the bit-width reduction techniques covered in the previous section. In the second step, the approach seeks the minimum number of bits in the fractional part of the fixed-point representation. It starts by using the same number of fractional bits for all the variables and then refines the representation via an error analysis technique.

Concerning the conversion of numeric formats and their efficient implementation, the compilation of numerical algorithms to reconfigurable architectures may have some particularities, according to the granulartity of the target architecture. In coarse-grained architectures (with or without barrel-shift hardware support), the reduction of the number of scaling operations might be an important goal. This reduction can have great importance in the overall performance of the implementation, as a trivial translation can produce a large number of such shift operations. Elimination of shift operations is possible when the operands that need to be aligned (e.g., addition) are represented with the same number of fractional bits. In fine-grained architectures, the shift operations by a known compile-time constant can be implemented by redirection of the bits of the fractional part of the representation, and thus is costless. In this case the reduction of the number of scaling operations is not considered.

4.1.4. Nonstandard Floating-Point Formats. It is possible to improve the execution time of arithmetic operations and/or reduce the amount of hardware resources used for specific applications by the adoption of nonstandard floating-point formats, which use specific numbers of bits for exponent and mantissa representations according to accuracy requirements. One floating-point format is referred to as block floating-point number representations [Ralev and Bauer 1999]. It combines the advantages of fixed- and floating-point representations, and hence can be an important optimization when compiling digital signal-processing applications [Kobayashi et al. 2004] to *reconfigware* (e.g., especially to fine-grained architectures such as FPGAs).

4.2. Instruction-Level Transformations

At the next level in granularity in code transformations, there are many instruction-level transformations that attempt to simplify or reduce the hardware resources allocated for a given computation by using a combination of algebraic simplification or circuit specialization.

Simple algebraic transformations can be used for *tree height reduction* (THR), *common subexpression elimination* (CSE), *constant folding*, *constant propagation*, and *strength reduction* [Muchnick 1997]. Many of the algebraic transformations have a positive impact, both in terms of execution time as well as use of hardware resources, and are independent of the target reconfigurable architecture. Examples of such transformations include the classical algebraic strength reduction and simplification cases such as replacing $-1 \times a$ by $-a$, $-(-j)$ by j, or $0+i$ by i. Other transformations are useful when the target architecture does not directly support the original operation, for example, replace i^2 by $i \times i$.

There are algebraic transformations specific to specialized hardware implementations that are only possible due to the flexibility of *reconfigware*. These transformations simplify the hardware needed to perform a given operation. For example, when a constant of the form 2^N is added to an operand, a single incrementer and a bit-level concatenation is required instead of a generic adder (an example of *bit-level operator specialization*). Another example is *operator strength reduction* (OSR); for example, we can replace $2 \times i$ by $i<<1$ or replace $3 \times i$ by $i+(i<<1)$.

When compiling arithmetic expressions, some properties should be evaluated in order to find more efficient hardware implementations. The arithmetic properties (commutative, associative, distributive, etc.), when efficiently combined, can substantially reduce the number of operations in an expression or/and the critical path length of the resulting hardware implementation.

4.2.1. Tree-Height Reduction (THR). *Tree-height reduction* or *tree-height minimization* [Micheli 1994] is a well-known technique to reduce the height of a tree of operations by reordering them in a way that the initial functionality is preserved (see Figure 4).

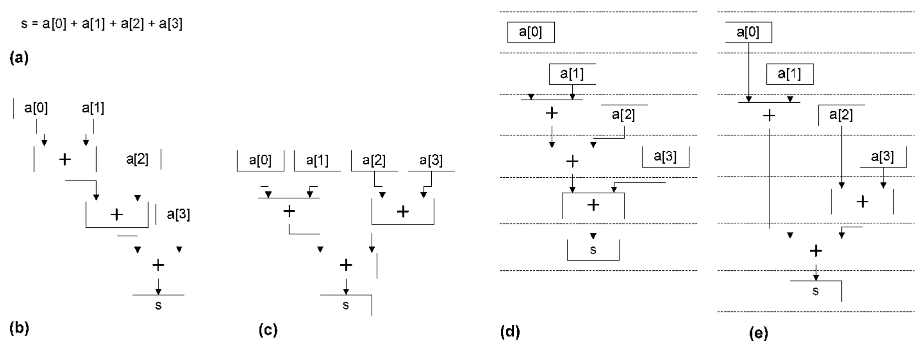


Fig. 4. *Tree-height reduction*: (a) a simple expression; (b) DFG with cascaded operations; (c) DFG after *tree-height reduction* is applied; (d-e) Impact on the scheduling length considering the serialization of memory accesses when applying to the DFG in (b) and (c), respectively.

The technique attempts to expose more parallelism by, in the best case, reducing the height of an expression tree from O(n) to O(log n), with n representing the number of nodes (operations) in the critical path, leading to a reduction of the critical path length of the circuitry that implements the expression tree.

THR can be easily performed when the expression consists of operations of the same type. When considering expressions with different operations, THR can be applied by finding the best combination of the arithmetic properties of those operations and factorization techniques. Applying THR may, in some cases, lead to worse implementations. This might happen when some operations in the expression tree share hardware resources and hence their execution must be serialized. The example in Figure 4 depicts this particular case. Figure 4(e) shows a schedule length, obtained from the DFG after THR, which is longer than the schedule length directly obtained from the original DFG and shown in Figure 4(d).

4.2.2. Operation Strength Reduction (OSR). The objective of *operation strength reduction* is to replace an operation by a computationally less expensive one or a sequence of operations. Strength reduction is usually applied in the context of software compilers to induction variables [Muchnick 1997]. Also, integer divisions and multiplications by compile-time constants can be transformed into a number of shifts and additions/subtractions [Magenheimer et al. 1988]. Due to the hardware characteristics—specialization (shifts by constants are implemented as simple interconnections) and a dataflow implementation (without the need of auxiliary registers to store the intermediate results)—strength reduction can reduce both the area and the delay of the operation, and is therefore well suited for the compilation to fine-grained reconfigurable architectures (e.g., FPGAs without built-in multipliers or dividers). Trivial cases occur when there is a multiplication or a division of an integer operand by a power of two constant. For these cases, a multiplication is accomplished by a simple shift of the operand, which only requires interconnections on fine-grained architectures.

The nontrivial multiplication cases require other implementation schemes. Some authors use arithmetic operations (e.g., factorization) to deal with the multiplications by constants. Bernstein [1986] proposes an algorithm that aims to find an optimal solution, but with exponential time complexity. An efficient scheme for hardware is using the canonical signed digit (CSD) representation [Hartley 1991], which is constant with a minimum number of nonzero bits and uses the symbols -1, $+1$ and 0. This representation permits to use of a small number of adder/subtracter components [Hartley 1991]. The minimum number of these components can be attained by applying

Table II. Example of Operation Strength Reduction Techniques on Integer Multiplications by Constants

Operation	231×A		
Representation	Binary	CSD	CSD+CSE
	011100111	100-10100-1	Pattern: 100-1
Resources (not considering shifts)	5 adders	2 subtracters and 1 adder	1 adder and 1 subtracter

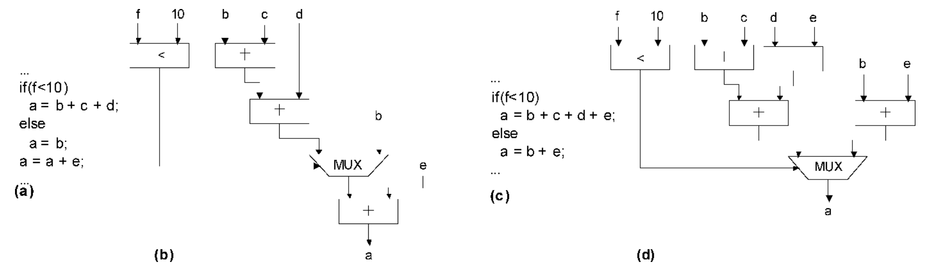

```
...
if(f<10)
  a = b + c + d;
else
  a = b;
a = a + e;
...
```
```
...
if(f<10)
  a = b + c + d + e;
else
  a = b + e;
...
```

Fig. 5. Code hoisting: (a) original code; (b) DFG implementation without code hoisting; (c) transformed code after code hoisting; (d) DFG implemntation with code hoisting and THR.

common subexpression elimination (CSE) on the CSD representation. In this case, CSE resumes to the identification of common CSD subpatterns. Table II shows three cases of applying OSR to the simple arithmetic operation: 231×A. The binary case refers to the direct use of the binary representation and gives a solution with the maximum number of operations (resources) required. As we can see, the use of CSD and CSE significantly reduces the number of additions and subtractions.

The OSR transformation can also be applied when dealing with floating-point operations, and has been used, for example, in the DeepC compiler [Babb et al. 1999], transforming expressions such as x/2.0 by subtracting one unit from the exponent of x.

4.2.3. Code Motion: Hoisting and Sinking. Code motion [Muchnick 1997], either locally or globally, is a technique that changes the order of operations in the program. Two distinct movements can be applied: code hoisting (moving up, i.e., against the flow of control) and code sinking (moving down, i.e., along the flow of control). Code motion is beneficial in the reduction of the size of the code by moving repeated computations of the same expression or subexpression, to a common path. In the context of hardware compilation, code motion may have the potential benefit of making the two computations independent (in terms of control flow), thus enabling concurrent execution. Such movement may increase the potential for other optimization techniques, as in the example in Figure 5. In this example, code hoisting and THR result in one more adder unit, but decrease the critical path length by the delay of one adder unit. Depending on the example, code hoisting and code sinking can be applied in the context of conditional paths to decrease the schedule length or to reduce the number of resources needed.

Depending on the scope of usage in a program, various types of code motion can be classified as follows:

—Loop invariant code motion: Code hoisting of loop invariant computations eliminates code that does not need to be executed on every loop iteration, and thus reduces the length of the schedule of the execution of the body of the loop.

—Interprocedural code motion: This transformation moves code (simple expressions, loops, etc.) across function boundaries such as constants and symbolic invariant expressions.

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
    sum = 0;
    for(y=0; y < W; y++) {
        sum += (a[x][y] * b[y][x]);
    }
    c[x] = sum;
}
```
(a)

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
    sum  = (a[x][0] * b[0][x]);
    sum += (a[x][1] * b[1][x]);
    sum += (a[x][2] * b[2][x]);
    c[x] = sum;
}
```
(b)

Fig. 6. Loop unrolling and opportunities for parallel execution: (a) original C source code; (b) full unrolling of inner loop.

—Code motion across conditionals: This is the most used type of code motion. Recent techniques have been applied in the CDFG and HTG for HLS of ASICs [Gupta et al. 2001; Santos et al. 2000], considering constraints on the number of each FU type.

—Code motion within a basic block: This is implicitly performed when a DFG is used, but it should be considered when ASTs are the intermediate representation used. In this case, code motion is used when performing scheduling with, for example, resource-sharing schemes.

One example of the first type is scalarization, which substitutes an array access with an address invariant in the loop by a variable and puts the array access before the loop body. In this context, this transformation reduces the number of memory accesses.

4.3. Loop-Level Transformations

An important goal of some of the more complex, and thus far-reaching, transformations is the matching of the instruction-level parallelism in a given computation to the available hardware resources at hand. In this category are the classical loop unrolling, loop tiling, and unroll-and-jam transformations. For example, loop unrolling potentially creates more opportunities for concurrently executing multiple instances of arithmetic operators that are only subject to data or hardware resource dependences.

Many loop transformations [Wolfe 1995] such as coalescing, collapsing, distribution (fission), jamming (fusion), interchanging, peeling, reordering, reversal, stripmining, tiling, splitting, and unrolling, when used in the context of reconfigurable hardware implementation, exhibit various unique characteristics. Overall, these transformations aim at increasing the amount of instruction-level parallelism (ILP), for example, by unrolling, and/or increasing the availability of the data items (e.g, by tiling). In addition, they may also enable other transformations mentioned in the previous sections.

Loop unrolling is the most commonly used loop transformation, as it decreases the loop overhead while increasing the opportunities for ILP. The loop body is replicated, and the index for each iteration is propagated to the statements in each instance of the loop body as illustrated in Figure 6. In some cases, however, due to I/O constraints and lack of further potential for optimizations, the utilization of loop unrolling does not compensate the extra hardware resources needed.

Loop tiling [Wolfe 1995] can be applied to inner loops to create two nested loops, and is able to split the original iteration space. Such a transformation enables the parallelization of instances of the new inner loop when each instance computes on data sets located on different memories or on the same memory with more than one port.

```
...
int img[N][N];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i++) {
        ... = img[j][i];
    }
    ...
}
```
(a)

```
...
int imgOdd[N][N/2], imgEven[N][N/2];
...
for(j=0; j < N; j++) {
    ...
    for(i=0; i < N; i+=2) {
        ... = imgOdd[j][i/2];
        ... = imgEven[j][i/2];
    }
    ...
}
```
(b)

Fig. 7. Loop unrolling and array data distribution example: (a) original C source code; (b) loop unrolled by 2 and distribution of img.

Another example of a loop transformation, especially devoted to decreasing execution time, is loop merging (fusion), because it can increase the efficiency of loop pipelining.

4.4. Data-Oriented Transformations

The flexibility of reconfigurable arhitectures, in terms of the configuration and organization of storage structures, makes data-oriented transformations such as data distribution, data replication, and scalar replacement particularly suited for these architectures. Many other transformations, in particular loop transformations, exhibit a special synergy with these data-oriented transformations, particularly for computations that manipulate array variables using affine index access functions, as in many signal and image processing algorithms.

4.4.1. Data Distribution. This transformation partitions a given program array variable (other data structures can also be considered) into many distinct arrays, each of which holds a disjoint subset of the original array's data values. Each of these newly created arrays can then be mapped to many distinct internal memory units or modules, but the overall storage capacity used by the arrays is preserved. This transformation, sometimes in combination with loop unrolling, enables implementations that can access the various sections of the distributed data concurrently (whenever mapped to distinct memories, thus increasing the data bandwidth). Figure 7 illustrates the application of loop unrolling and data distribution for the array img of the example code depicting the declarations of the two arrays imgOdd and imgEven.

4.4.2. Data Replication. Another data-oriented transformation that increases the available data bandwidth, this time at the expense of an increase in the storage used, is data replication. In this case, shown in Figure 8, data is copied as many times as desired into distinct array variables, which can then be mapped to distinct memory modules to enable concurrent data access.

Clearly, and compared to the previous transformation, the storage cost of replication may be prohibitive. As a result, its applicability is limited to small arrays that are immutable throughout the computation, that is, to variables that hold coefficient or parameter data that needs to be accessed frequently and freely. In addition to the storage issues, there is also the issue of consistency should the data be modified during the computation. In this case, the execution must ensure all replicas are updated with the correct values before the multiple copies can be accessed [Ziegler et al. 2005].

```
...
int img[N][N];

...
for(j=0; j < N; j++) {

    ...
    for(i=0; i < N; i++) {
        ...= img[j][i];
    }
    ...
}
```
(a)

```
...
int imgA[N][N], imgB[N][N];

...
for(j=0; j < N; j++) {

    ...
    for(i=0; i < N; i+=2) {
        ... = imgA[j][i];
        ... = imgB[j][i+1];
    }
    ...
}
```
(b)

Fig. 8. Loop unrolling and array data replication example: (a) original source code; (b) loop unrolled by 2 and replication of img.

```
...
int vec[N][4], k[4];

...
for(j=0; j < N; j++) {

    ...
    for(i=0; i < 4; i++) {
        ... = vec[j][i] * k[i];
    }
    ...
}
```
(a)

```
...
int vec[N][4], k[4], k0, k1, k2, k3;

...
k0 = k[0]; k1 = k[1]; k2 = k[2]; k3 = k[3];
for(j=0; j < N; j++) {
    ...
    ... = vec[j][0] * k0;
    ... = vec[j][1] * k1;
    ... = vec[j][2] * k2;
    ... = vec[j][3] * k3;
    ...
}
```
(b)

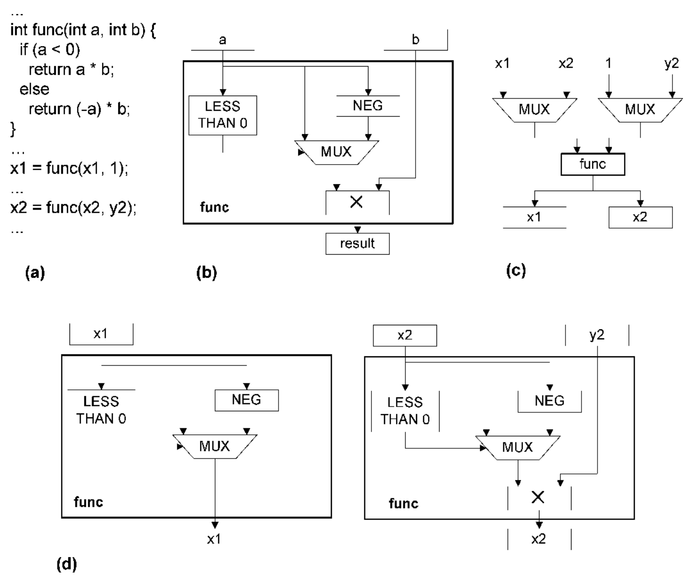
Fig. 9. Scalar replacement using loop urolling: (a) original C source code; (b) scalar replacement of *k* using loop unrolling.

4.4.3. Scalar Replacement. In this transformation [So et al. 2004] the programmer selectively chooses which data items will be reused throughout a given computation and cached in scalar variables. In the context of traditional processors, the compiler then attempts to map these scalar variables to internal registers for increased performance. When mapping computations to reconfigurable architectures, designers attempt to cache these variables either in discrete registers or in internal memories [So and Hall 2004]. Figure 9 illustrates the application of scalar replacement to the k array variable. In this case, multiple accesses to each element of the array are eliminated by a prior loading of each element to a distinct scalar variable. The use of scalar replacement using loop peeling was shown in Diniz [2005]. In this case, the first iteration where the reused data is accessed is peeled away from the remaining loop iterations.

This transformation, although simple to understand and apply when the reused data is read-only in the context of the execution of a given loop, is much more complicated to implement when in a given loop the data that is reused is both read and written. In this case, care must be taken that the cached data in the scalar replaced register is copied back into the original array data, so that future instantiations of the transformations can use the correct value.

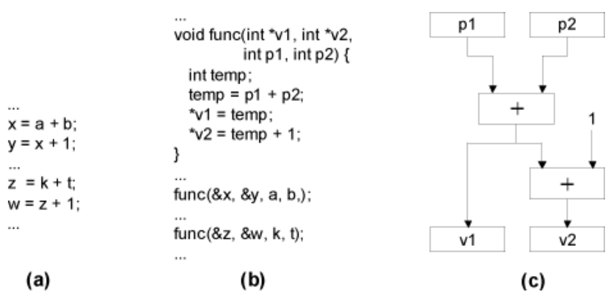
4.5. Function Inlining and Outlining

Function inlining and *outlining* are dual source transformations with complementary effects. *Function inlining* (also known as *function unfolding* and *inline expansion*) replaces a call instruction or function invocation statement with the statements in the body of the invoked function. With respect to hardware compilation, one of the first



```
...
int func(int a, int b) {
  if (a < 0)
    return a * b;
  else
    return (-a) * b;
}
...
x1 = func(x1, 1);
...
x2 = func(x2, y2);
...
```

Fig. 10. Example illustrating *function inlining*: (a) source code; (b) datapath for the function; (c) hardware structure used to time share the hardware resources of the function; (d) datapaths after *function inlining*.



```
...
x = a + b;
y = x + 1;
...
z = k + t;
w = z + 1;
...
```

```
...
void func(int *v1, int *v2,
          int p1, int p2) {
  int temp;
  temp = p1 + p2;
  *v1 = temp;
  *v2 = temp + 1;
}
...
func(&x, &y, a, b,);
...
func(&z, &w, k, t);
...
```

Fig. 11. Example with *function outlining*: (a) source code; (b) transformed code; (c) datapath for the function.

uses of *function inlining* was referred to by Rajan and Thomas [1985]. *Function outlining* (also known as *inverse function inlining*, *function folding*, and *function exlining* [Vahid 1995]) abstracts two or more similar sequences of instructions replacing them with a call instruction or function invocation of the abstracted function.

Figures 10 and 11 illustrate these transformations. While *function inlining* increases the amount of potential instruction-level parallelism by exposing more instructions in the function call-site, function outlining reduces it. *Function inlining* eliminates the explicit sharing of the resources implementing the function body as each call site is now replaced by its own set of hardware resources, as illustrated in Figure 10. However, the resources for each call-site are now exposed at the call site level to resource sharing algorithms, and may augment the potential for resource sharing. Inlining also allows the specialization of the inlined code, and consequently the correspondent hardware resources, for each specific call site. This may be possible when there are constant parameters, leading to constant propagation, or different types inferred or bit-widths for one or more input/output variables of the function according to the call site, in turn allowing many other optimizations as well as operator specialization. In addition, each of the call site invocations has its own set of hardware resources, exposing what is called functional or task parallelism in the program.

Conversely, *function outlining*, as illustrated in Figure 11, improves the amount of resource sharing, as all invocations use the same hardware for implementing the original functionality. This is performed at the expense of constraining the potential task-level parallelism in the hardware implementation, as the various function invocations now need to have their execution serialized by the common hardware resources.

5. MAPPING TECHNIQUES

We now present some of the most important aspects related to the mapping of computations to reconfigurable architectures—a very challenging task given the heterogeneity of modern reconfigurable architectures. As many of the platforms use a board with more than one reconfigurable device and multiple memories, a key step consists in the spatial partitioning of computations among the various reconfigurable devices in the board and/or its temporal partitioning when the computations do not fit in a single device or in all devices, respectively. Furthermore, array variables must be mapped to the available memories (onchip and/or offchip memories). At lower levels, operations and registers must be mapped to the resources of each reconfigurable device. The compiler must supply one or more computing engines for each reconfigurable device used. In addition, it is very important to exploit pipelining execution schemes at either a fine- or coarse-grained level.

The following sections describe several mapping techniques, beginning with the basic execution techniques and then addressing spatial and temporal partitioning and pipelining.

5.1. Control Flow Optimizations

When compiling sequential programming languages, the exploitation of multiple flows of control provided by specialized architectures can be an important performance-enhancing technique. The concurrent nature of hardware provides support for parallel execution of multiple branches of the computation as highlighted by the transformations in the next sections.

5.1.1. Speculative Execution. The generation of specific architectures and the mapping of computational constructs on reconfigurable architectures in such a way that speculative execution of operations without side-effects is accomplished might be important to attain performance gains. This speculative execution does not usually require additional hardware resources, especially if the functional units (FUs) are not being shared among operations. This is the case for most of the integer operations as, for example, the additional hardware resources needed to share an adder of two integer operands makes the sharing inefficient. Speculation in the case of the operations with side-effects (e.g., memory writes) requires the compiler to quantify if the restoring step that must be done neither degrades the performance nor needs an unacceptable number of hardware resources.

5.1.2. Multiple Flows of Control. Since the control units for the datapath are also generated by the compilation process, it is possible to address multiple flows of control. This can increase the number of operations executing on each cycle by increasing the ILP. These multiple flows of control can be used in the presence of conditional

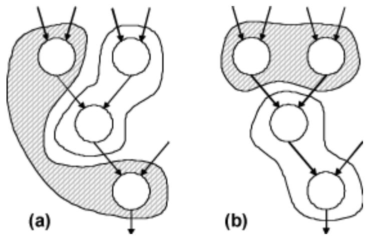


Fig. 12. Examples of partitions (the graphs can represent a DFG or a task graph): (a) possible spatial partitions but impossible temporal partitions; (b) possible spatial and temporal partitions.

constructs (e.g., *if-then-else*), loops without data-dependencies among them, or functional parallelism. The generation of implementations with concurrent execution of, for example, data-independent loops may lead to noticeable performance improvements, especially when such loops do not share resources.

5.1.3. From Control to Dataflow. Given the suitability in fine-grained reconfigurable architectures for implementing concurrent harware structures, program decision logic, that is, logic that enables/disables the execution of a certain instruction [August et al. 1999], and predicated execution [Beck et al. 1993] are effective techniques for handling control constructs. This implementation scheme can be used to convert the mutually exclusive paths in *if-then-else* or *switch* statements to concurrent paths (known as *if-conversion* [Allen et al. 1983]) where operations are controlled by the corresponding predicates. When considering speculative execution of operations without side-effects, only selection points and memory stores have to be dependent on the program decision logic. Selection points (e.g., implemented with multiplexers) do not need intervention of the control unit, as selection entries corresponding to signals are directly produced by the program decision logic which can be integrated in the datapath.

The creation of the program decision logic may consider minimization of the logic expressions. The impact of minimization, when dealing with fine-grained reconfigurable architectures, may not be important. However, when targeting coarse-grained architectures, where each logic operation needs a processing element, the minimization may lead to important reductions on the number of resources needed.

The selection points are easily determined by transforming the program to the SSA (static single assignment) form [Cytron et al. 1991]. The selection points are explicitly represented in the SSA-form. Furthermore, since this form is single-assignment, it simplifies the construction of the DFG.

Besides *if-conversion* there are other techniques that can transform control constructs into simple dataflow. For instance, the statement `if((a&1)==1) a++;` can be implemented by adding 0 to `a` and using the least significant bit of the variable `a` as carry-in. The potential performance improvement for such transformations is low, but when they are applicable, the resource savings are substantial.

5.2. Partitioning

Both temporal and spatial partitioning may be performed at the structural or behavioral (also known as functional partitioning) levels. This description of partitioning techniques focuses mainly on the partitioning at behavioral levels, since this article is about compilation of high-level descriptions to reconfigurable computing platforms. Figure 12 presents an example depicting spatial and temporal partitioning. Note that the dependences must imply a strict ordering of the temporal partitions. This is not the case for spatial partitions. When applied simultaneously, the partitioning problem is much more complex, and current solutions deal with it by splitting the partitioning in two steps. For example, after temporal partitioning, spatial partitioning is performed for each temporal partition. An example of a system dealing with spatial and temporal partitioning is the SPARCS [Ouaiss et al. 1998a]. The next sections describe the most relevant work on temporal and spatial partitioning.

5.2.1. Temporal Partitioning. Temporal partitioning allows us to map computations to reconfigurable architectures without sufficient hardware resources. It reuses the available device resources by different circuits (configurations) by time-multiplexing the device, and is thus a form of virtualization of hardware resources. Resource virtualization on RPUs, achievable due to its dynamic reconfiguration capabilities, also provides an attractive solution to save silicon area. The resultant temporal partitions need a controller responsible for the control flow of configurations (done by the host processor or by a specific controller). Schemes to communicate data between configurations are also needed to support this type of partitioning.

Most of the research effort on temporal partitioning have been applied to rapid prototyping of hardware circuits (structural temporal partitioning at gate-level) that do not fit in the available hardware resources (e.g., Liu and Wong [1999] and Trimberger [1998]). One of the first compilers addressing temporal partitioning was presented by Gokhale and Marks [1995]. Other examples of compilers integrating temporal partitioning are the NENYA compiler [Cardoso and Neto 1999, 2003] and the XPP-VC compiler [Cardoso and Weinhardt 2002].

The similarities of both scheduling in HLS [Gajski et al. 1992] and temporal partitioning allow the use of known scheduling schemes for temporal partitioning. Some authors, for example, Ouaiss et al. [1998a] and Vasilko and Ait-Boudaoud [1996], have considered temporal partitioning at the behavioral level, bearing in mind the integration of synthesis. Vasilko and Ait-Boudaoud [1996] present a heuristic based on a static list-scheduling algorithm, enhanced to consider temporal partitioning and partial reconfiguration. The approach exploits the partial dynamic reconfiguration capability of the devices, while creating temporal partitions.

The simplest approaches only consider temporal partitioning, without exploiting the sharing of FUs. Purna and Bhatia [1999] use both a temporal partitioning algorithm based on leveling the operations by an ASAP scheme and another one based on clustering of nodes. The approach neither considers communication costs nor resource sharing. Takayama et al. [2000] present an alternative algorithm that selects the nodes to be mapped in a temporal partition with two different approaches, one for satisfying parallelism and the other for decreasing communication costs.

Cardoso and Neto [2000] present an extension to static list scheduling, where the algorithm is aware of the communication costs while trying to minimize the overall execution time. The results achieved compare well to near-optimal solutions obtained with a simulated annealing algorithm tuned to do temporal partitioning while minimizing an objective function which integrates the execution time of the temporal partitions and the communication costs.

Ouaiss et al. [1998a] and Kaul and Vemuri [1998] define the temporal partitioning problem as a 0-1 nonlinear programming (NLP) model. This model is then transformed to integer linear programming and solved. Due to the long execution times, this approach is not practical for large examples, and thus some heuristic methods have been developed to permit its usability on larger input examples [Kaul and Vemuri 1999]. Kaul et al. [1999] exploit loop fission while doing temporal partitioning. They aim at minimizing the overall latency by using the active temporal partition as long

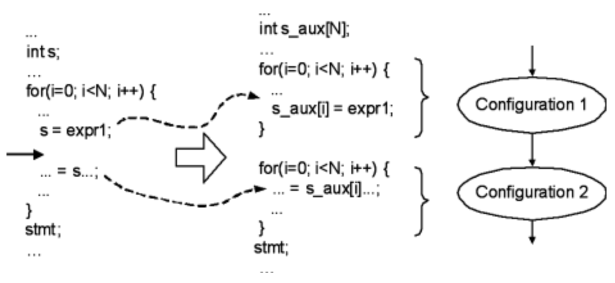


Fig. 13. Loop distribution as a form of splitting a loop across different temporal partitions.

as possible in the presence of a sequence of tasks executed repeatedly. Sharing FUs is considered inside tasks, and temporal partitioning is performed at the task level. Design space exploration is performed by inputting different design implementations for each task (with different resource sharing tradeoffs) to the temporal partitioning algorithm. Each implementation is generated by a HLS tool, constraining the number of FUs of each type. Note, however, that this approach lacks a global view and is time-consuming.

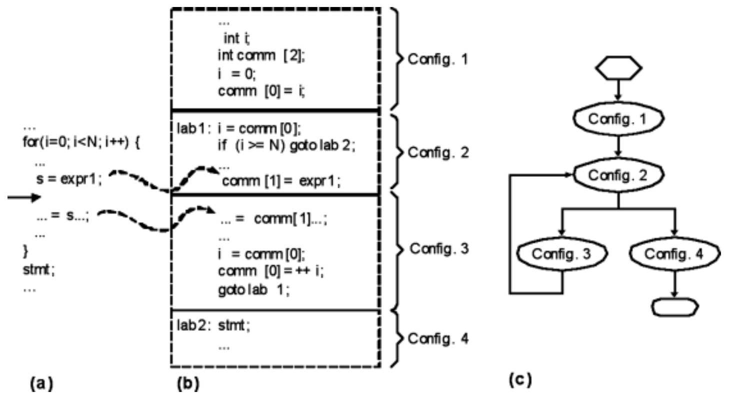
Compilers for dynamically reconfigurable FPGA-based systems need to consider the case of reducing the number of temporal partitions (configurations), by enabling sharing of some FUs in the same temporal partition. Pandey and Vemuri [1999] show a scheme based on the force-directed list-scheduling algorithm that considers resource sharing and temporal partitioning. The algorithm tries to minimize the overall execution times, performing a tradeoff between the number of temporal partitions and the sharing of FUs. However, the approach adapts a scheduling algorithm not originally tailored to do temporal partitioning and lacks a global view. Cardoso [2003] describes an algorithm matched to the combination of temporal partitioning and sharing of FUs that, in addition, maintains a global view. Unlike other approaches, the algorithm merges those tasks in a combined and global method. The results confirm the efficiency and effectiveness of the approach.

Compiling loops that require more resources for each temporal partition than available can be attained by the application of two loop transformations, namely:

—*Loop distribution*, a technique used in software compilation [Muchnick 1997], partitions a loop into two or more loops, each of which now fits in the available hardware resources. However, loop distribution often uses more memory resources to store the values of certain scalar variables (scalar variable expansion) computed in each iteration of the first loop and used by the second loop (see example in Figure 13).[4] Although using loop distribution can be efficient, it might not be applied due to loop-carried dependences. Its advantage is that reconfiguration is only needed between the loops.

—*Loop dissevering* [Cardoso and Weinhardt 2002] (see Figure 14) can be applied to guarantee the compilation of any kind of loop, in the presence of any type of dependences, and with any nested structures. This transformation converts a loop into a set of configurations, which can be managed by the host CPU or by a configuration controller when this controller is included in the target architecture [Baumgarte et al. 2003]. This transformation does not depend on loop dependenecs and does not require expansion of certain scalar variables into arrays, as is the case

[4]When the loops are unbound or with iteration space determined at compile time, this also requires the communication of data sizes unknown at compile time.



```
...
for(i=0; i<N; i++) {
  ...
  s = expr1;
  ... = s...;
  ...
}
stmt;
...
```

```
        ...
         int i;
        int comm  [2];
        i  = 0;
        comm  [0] = i;
lab1 :  i = comm [0];
        if  (i >= N) goto lab 2;
        ...
         comm [1] =  expr1;
         ... =  comm[ 1]...;
        ...
        i  = comm [0];
        comm  [0] = ++ i;
        goto lab  1;
lab2 :  stmt;
        ...
```

Fig. 14. Loop dissevering as a form of splitting a loop across different temporal partitions: (a) original source code (the arrow shows where the loop is partitioned); (b) transformed code with the statements needed to communicate the value of scalar variables between configurations; (c) the reconfiguration controlflow graph to be orchestrated by the configuration management or the host microprocessor.

for the loop distribution technique. Since loop dissevering requires reconfigurations in each loop iteration, it is more suitable for short loops and outer loops in loop nests.

Research effort have presented devices with multiple on-chip contexts with almost negligible reconfiguration times (one clock cycle) of configurations already on-chip, and industrial efforts have already presented context switching between configurations in a few nanoseconds [Fujii et al. 1999]. With those non-negligible reconfiguration times, temporal partitioning methods should consider the overlapping of reconfiguration and execution in order to reduce the overall execution time. Ganesan and Vemuri [2000] present a method to do temporal partitioning considering pipelining of the reconfiguration and execution stages. Their approach divides the FPGA into two parts, bearing in mind the overlapping of the execution of a temporal partition in one part (previously reconfigured) with the reconfiguration of the other part.

5.2.2. Spatial Partitioning. Spatial partitioning refers to the task of automatically splitting a computation into a number of parts such that each of them can be mapped to each reconfigurable device (RPU). This task is performed when computations cannot be fully mapped into a single RPU and the reconfigurable computing platform has two or more RPUs with interconnection resources among them. Spatial partitioning does not need to preserve dependences when splitting the initial computations. Spatial partitioning, however, needs to take care of the pins available for interconnection, and to operations that access the same data sets. The computations accessing those data sets need to be mapped on RPUs that have access to the memory storing them.

As spatial partitioning has been extensively studied by the hardware design community, mainly in mapping circuitry onto multi-FPGA boards, it is not surveyed here. A detailed presentation of this topic can be found in Brasen and Saucier [1998]; Krupnova and Saucier [1999]; and Vahid et al. [1998]. In the context of compiling to *reconfigware*, behavioral spatial partitioning was, to the best of our knowledge, originally reported in Schmit et al. [1994]. Behavioral partitioning has already been shown to lead to better results than structural partitioning [Schmit et al. 1994; Vahid et al. 1998]. Peterson et al. [1996] describe an approach targeting multiple FPGAs using simulated annealing. Lakshmikanthan et al. [2000] present a multi-FPGA partitioning algorithm based on the Fiduccia-Mattheyses algorithm. Results achieved with a simulated annealing approach are also shown.

Spatial partitioning is also performed in the CHAMPION environment [Ong et al. 2001]. There, different algorithms are applied at the module level, and only partitioning points at the interconnections of the modules are considered. In this environment, the partitioning preserves the functional information flow, which is similar to structural partitioning. The algorithms consider the constraints on the pin number of each FPGA, on the accesses to external memories (a partition can contain a certain number of RAM access modules, which is limited by the number of local memories connected to each FPGA), and on the temporal partitioning. The partitioning phase aims at minimizing the number of FPGAs used. When targeting machines with multiple FPGAs, where each FPGA has access to a single memory, it is important to split computations across the FPGAs whenever such individual computations access disjoint sets of data. Thus, the computations and data are both split and mapped onto different FPGAs and memories and can execute in parallel, thereby enabling coarse-grained parallelism.

5.3. Resource Mapping

This section briefly describes the issues on mapping the computing constructs to the resources available in the target architecture.

5.3.1. Register Assignment. In principle, scalar variables are directly stored in the RPU registers, and compound data structures (e.g., arrays) mapped to memories. Hence scalar variables need not be loaded from and stored to memories and are always directly available on the RPU.

Typically, not all the variables used in a given program need registers, as some of them are promoted to wires. Register assignment algorithms [Muchnick 1997] can be used whenever the number of variables in the program (in each temporal partition, to be more precise) exceeds the number of registers in the reconfigurable architecture.

When using static-single-assignment (SSA) intermediate representation forms, the list of Φ–functions presented in each loop header (here, only natural loops [Muchnick 1997] are considered) defines the scalar variables which need to be registered in order to preserve the functionality. An SSA-form can be directly mapped to a DFG, since each SSA scalar variable is associated to an edge in the DFG (each edge represents a data transfer). These edges can be typically implemented by wires in hardware. This approach is well suited for the dataflow characteristics of the reconfigurable architectures.

Another possibility is to use one register for each variable in the source program. However, this option may use an unnecessary number of registers, may constrain the chaining of operations, and may increase the amount of hardware needed to enable the writing to a register by all the source definitions.

As flip-flops are abundantly distributed in FPGA-based reconfigurable architectures (one per cell), register sharing is not fruitful. There is no need for lifetime analysis of scalar variables in the code to use a minimum number of registers to store them. Such an analysis is used in software compilation and in HLS because of the usual use of centralized register files, the limited number of register resources, and, in the case of ASICs, the profitable sharing of registers.

Note, however, that more registers are needed to store results of FUs assigned to more than one operation or as pipeline stages, as explained in the following sections.

Registers can also be used to store data computed or loaded in a loop iteration and used in subsequent iterations:

—Temporal common subexpression elimination is a technique used in CAMERON to optimize the generated datapath [Böhm et al. 2001]. The technique identifies expressions that recompute values that were computed in previous iterations of the loop and replaces them with registers. The authors show the application of the technique to window-based image processing applications.

—Data reuse is applied when the same array elements are loaded in different loop iterations, or when an array element is stored in one loop iteration and loaded in subsequent loop iterations. Whenever this happens, the number of loads can be reduced by using registers to store data in between iterations, as explained later on.

5.3.2. Mapping of Operations to Resources. Successive operations can sometimes be grouped in a single operation corresponding to an FU in the hardware library of the target reconfigurable architecture, for example, A×B+C is grouped to MUL-ADD(A,B,C). Such a grouping is known in software compilation by instruction-combining [Muchnick 1997]. It is used, for example, in the XPP [Baumgarte et al. 2003] where each PE of the architecture directly supports special functions such as MUL-ADD. When targeting fine-grained reconfigurable hardware (e.g., FPGAs), instruction-combining is only worth the effort if the hardware to implement the combined instructions performs better than the individual operations. Note also that instruction-combining may reduce the potential to share FUs among operations.

A specific example of mapping operations to resources is related to the use of constant coefficient multipliers. This type of multiplier can be efficiently implemented using FPGA hardware resources [Wirthlin et al. 2001]. Such multipliers might be an option to consider over *operator strength reduction* when such reduction may need too many adders/subtracters.

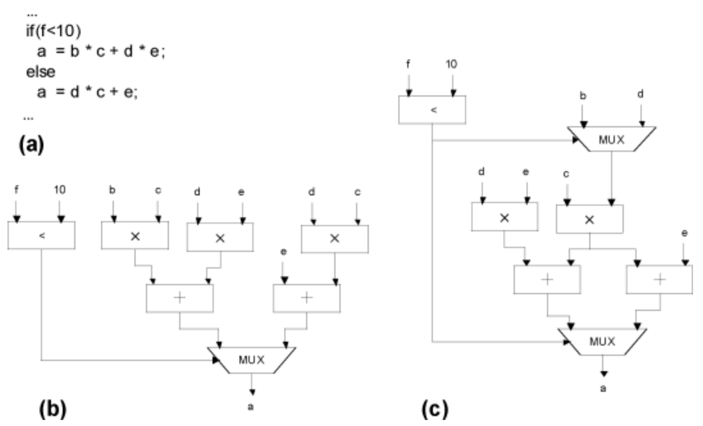
Whenever each operation can be computed by more than one FU (e.g., using different implementations of a multiplier), as is the case when dealing with fine-grained architectures, the mapping phase can be more complex, and heuristics are usually employed. A simple strategy may use the fastest FUs for the operations in the critical path of the datapath, and may try to use FUs with fewer resources for operations in the other paths.

Some FUs can be shared/reused by a number of operations of the same type. A set of operations can be used in an FU, for example, ALU, and operations of different types can also reuse that FU.[5] When a resource is reused, registers, multiplexers, and control circuitry are needed. One of the compilers that include resource sharing by operations in a basic block is the NAPA-C compiler [Gokhale and Gomersall 1997]. The compiler also uses the commutative property which, whenever possible, tries to swap the input operands of a shared FU in order to reduce the number of multiplexers needed.

Sometimes, FUs can be shared among operations without the intervention of the control unit and insertion of registers (see example in Figure 15) [Cardoso and Neto 1999]. Such an opportunity may arise among compatible operations present in branches of *if-then-else* and *switch* instructions (which define mutually exclusive execution paths). However, this sharing can increase the critical path delay, as in the example shown in Figure 15.

There are sometimes specialized opportunities to optimize the mapping. For instance, in fine-grained *reconfigware*, the generated circuit can have access to the reset signal of the flip-flops, and thus can simplify the mapping of computations with scalar variables initialized to zero, which is the case for many loop control variables. Without

[5]Note that even in fine-grained reconfigurable architectures (e.g., FPGAs) the reuse of some of the FUs is not efficient due to the hardware resources needed by the scheme to implement the reuse.



```
...
if(f<10)
  a = b * c + d * e;
else
  a = d * c + e;
...
```

Fig. 15. Hardware implementations of a simple example (a): (b) without functional unit sharing; (c) sharing one multiplier in mutually exclusive execution paths.

using the reset signal, the initial value to be assigned (zero) is selected with a multiplexer structure, among the other assignments. Using the reset signal in examples of type: *for (int i = 0; i < N; i++) {};* where there is no assignment to variable i in the loop body, implies the elimination of the multiplexer.

Selection structures (e.g., Φ functions in the SSA-form) can be implemented as multiplexers or by sharing lines/buses accessed with a set of tri-state buffers for each data source. Selection points of the form N:1 (N inputs to one output) can be implemented by trees of 2:1 multiplexers or by a single N:1 multiplexer. The implementation depends on the granularity and multiplexer support of the target architecture (e.g., N:1 multiplexers are decomposed in 2:1 multiplexers in the *garpcc* compiler [Callahan 2002]). The use of multiplexers is an efficient solution when the number of data sources to be multiplexed is low; otherwise the use of tri-state connections may be preferable. Although many reconfigurable devices directly support 2:1 multiplexers, tri-state logic is not supported by most architectures.

Some simple conditional statements, however, can be implemented by fairly simple hardware schemes. For example, expressions such as (a<0) and (a>=0) can be performed by simply inspecting the logic value of the most significant bit of the variable, a (assuming the sign of the representation identified by that bit).

Mapping operations to hardware resources in the presence of bit-widths that exceed the ones directly supported by coarse-grained reconfigurable architectures presents its own set of issues. In these architectures, and when operations exceed the available bit-width directly supported by the FUs, the operations must be decomposed, that is, transformed to canonical/primitive operations. This decomposition or expansion of instructions may then exacerbate instruction scheduling issues.

5.4. Pipelining

Pipeline execution overlaps different stages of a computation. For instance, a given complex operation is divided into a sequence of steps, each of which is executed in a specific clock cycle and by a given FU. The pipeline can execute steps of different computations or tasks simultaneously, resulting in a substantial throughput increase and thus leading to better aggregate performance. Reconfigurable architectures, both fine- and coarse-grained, present many opportunities for custom pipelining via the configuration of memories, FUs and pipeline interstage connectivity. The next sections describe various forms of pipelining enabled by these architectures.

5.4.1. Pipelined Resources. In this form of pipelining, operations are structured internally as multiple pipeline stages. Such stages may lead to performance benefits when those operations are executed more than once.[6] Some approaches use decomposition of arithmetic operations in order to allow the addition of pipeline stages. For instance, the approach by Maruyama and Hoshino [2000] decomposes arithmetic operations to sets of 4-bits. Such decomposition is used to increase the throughput of the pipelining structures and to chain sequences of operations more efficiently. For instance, the least significant 4 bits of the result of a 32-bit ripple-carry adder can be used by the next operation without waiting for the completion of the 32-bits. Other authors perform decomposition due to the lack of direct support in coarse-grained architectures. The DIL compiler for the PipeRench architecture is an example of the latter case [Cadambi and Goldstein 2000].

5.4.2. Pipelining Memory Accesses. Pipelining of memory operations is an extremely important source of performance gains in any computer architecture, given the growing gap between processor speeds and memory latencies. Reconfigurable computing architectures are no exceptions to this rule, and several commercially available architectures offer support for such memory access modes [Annapolis 1999b; Jones et al. 1999]. In this context researchers have developed frameworks to exploit these features, particularly in the context of data-streaming applications [Frigo et al. 2001].

5.4.3. Loop Pipelining. Loop pipelining is a technique that aims at reducing the execution time of loops by explicitly overlapping computations of consecutive loop iterations. Only iterations (or parts of them) that do not depend on each other can be overlapped. Two distinct cases are loop pipelining of inner loops and pipelining across nested loops. While the former has been researched for many years, the latter has not been the focus of too much attention. We first consider inner-loop datapath pipelining and discuss important aspects on compiling applications with well ”structured” loops.[7]

The *pipeline vectorization* technique of the SPC compiler proposed by Weinhardt and Luk [2001b] in the context of FPGAs was also applied to the XPP-VC compiler [Cardoso and Weinhardt 2002], which targets the coarse-grained PACT XPP reconfigurable computing architecture [Baumgarte et al. 2003]. *Pipeline vectorization* considers loops without true loop-carried dependences and loops with regular loop-carried dependences (i.e., those with constant dependence distances) as pipelining candidates. As only innermost loops are eligible for pipelining with this technique, some loop transformations (unrolling, tiling, etc.) are considered as making the most promising loops innermost loops. For each loop being pipelined, *pipeline vectorization* constructs a DFG for the loop body by replacing conditional statements by predicated statements and selections between variable definitions by multiplexers. For loops without dependences, the body's DFG is pipelined by inserting registers as in standard hardware operator pipelines. Loops with regular dependences are treated similarly, but feedback registers for the dependent values must be inserted. To reduce the number of memory accesses per loop iteration, memory values which are reused in subsequent iterations are stored in shift registers. Finally, the loop counter and memory address generators are

[6]Resources compute more than once when used in loop bodies or when used by more than one operation presented in the behavioral description.
[7]*L*oops that are perfectly or quasi-perfectly nested and have symbolically constant or even compile-time constant loop bounds and that manipulate arrays using data references with affine index access functions.

```
...
for(int i=0; i<N; i++) {
     C[i] = A[i]*B[i];
}
...
```

(a)

```
...
int tmp1 = A[0]; // prologue
int tmp2 = B[0]; // prologue
for(int i=1; i<N; i++) {
     C[i-1] = tmp1*tmp2;
     tmp1 = A[i];
     tmp2 = B[i];
}
C[N-1] = tmp1*tmp2; // epilogue
...
```

(b)

Fig. 16. Example of software pipelining: (a) original code; (b) transformed code.

adjusted to control the filling of the aforementioned shift registers, pipeline registers, and the normal pipeline operation.

Another approach to inner loop pipelining is based on *software pipelining* (also known in the context of HLS as *loop folding* [Gajski et al. 1992]) techniques [Muchnick 1997]. Figure 16 shows an example of the *software pipelining* transformation (using a prologue and an epilogue), which can lead to better performance if the arrays, A and B, can be accessed concurrently. In such a case, the new values of tmp1 and tmp2 are loaded in parallel with the multiplication tmp1*tmp2.

Many authors use the modulo scheduling scheme to perform *software pipelining*. Rau's iterative modulo scheduling (IMS) algorithm [Rau 1994] is the one most used. Examples of compilers using variations of Rau's algorithm are the *garpcc* [Callahan and Wawrzynek 2000]; the MATLAB compiler of the MATCH project [Haldar et al. 2001a]; the NAPA-C compiler [Gokhale et al. 2000a]; and the compiler presented by Snider [2002]. The loop pipelining approach used in MATCH [Haldar et al. 2001a] also uses a resource constrained list-scheduling algorithm in order to limit the number of operations active per state, and thus the number of resources needed. Snider's approach [2002] uses an IMS version that considers retiming [Leiserson and Saxe 1991] to optimize the throughput and exploits the insertion of pipeline stages between operations. The *garpcc* [Callahan and Wawrzynek 2000] targets a reconfigurable architecture with a fixed clock period and pipeline intrinsic stages, and does not require exploitation of the number of stages and retiming optimizations. Table III presents the main differences of the loop pipelining schemes used by the state-of-the-art compilers as far as loop pipelining is concerned.

Another pipelining scheme consists in the overlapping iterations of an outer loop maintaining the pipelining execution of the inner loop, as illustrated by Bondalapati [2001]. The idea is to overlap the execution of iterations of the outermost loop maintaining the pipelined execution of the inner loop. This scheme requires memories instead of simple registers as pipeline stages. These stages inserted in the body of the inner loop must decouple the execution of all the iterations of such loops. Interesting speedups have been shown for this technique when applied to an IIR (infinite impulse response) filter and targeting the Chameleon architecture [Salefski and Caglar 2001].

Some loop transformations, such as loop unrolling, may increase the potential for performance improvements by exposing higher ILP degrees, permitting other code optimizations, and/or by exposing a larger scope for loop pipelining. Automatic analysis schemes to determine pipelining opportunities in the presence of loop transformations lead to very complex design spaces, as these loop transformations interact and substantially impact the amount of hardware resources required to implement pipelined implementations after being subject to these transformations.

Recently, a pipeline scheme at coarse-grained levels has been exploited by Ziegler et al. [2002]. The scheme allows the overlap of the execution steps of subsequent loops or functions, that is, functions or loops waiting for data to start computing as soon

Table III. Main Differences of Inner Loop Pipelining Schemes

Compiler	Loop Pipelining Scheme	Based on the Algorithm:	Target Architecture Characteristics	Goal	Applicability
SPC [Weinhardt and Luk 2001b]	Pipeline Vectorization	Pipeline Vectorization, retiming	Fixed clock period, some FUs with fixed pipeline stages	Throughput as high as possible	well-structured, inner FOR loops with affine index memory accesses
garpcc [Callahan and Wawrzynek 2000]	Software pipelining	IMS	Fixed clock period, intrinsic pipeline stages	Throughput as high as possible	A broad class of inner loops with affine index memory accesses
Snider's compiler [Snider 2002]	Software pipelining	IMS with retiming	Some FUs with fixed pipeline stages	Exploit clock period versus area (adding stages)	A broad class of inner loops with affine index memory accesses

as the required data items are produced in a previous function or by a specific iteration of a previous loop. Compilers can identify the opportunities for this coarse-grained pipelining by analyzing the source program and recognizing producer/consumer relationships between iterations of loops. In the simplest case, the compiler identifies the set of computations for each of the loops in the outermost loop body and organizes them into tasks. Then, it forms a DAG that maps to the underlying computing units. Each unit can execute more than one function, and the compiler is responsible for generating structures to copying the data to and from the storage associated with each pipeline stage (if a direct link between the computing units is not supported). It is also responsible for generating synchronization code that ensures that each of the computing units accesses the correct data at the appropriate intervals.

5.4.4. Pipelining Configuring-Computation Sequences. Since the configuration of the reconfigurable resources takes several clock cycles,[8] it is important to hide such latency overheads by overlapping the steps to store the next configuration on chip while the current active configuration is running. Even in architectures without on-chip cache for configuration data or context planes, but with partial reconfiguration support, such pipelining can be used to hide the reconfiguration time overhead by using, for example half of the device to run the current configuration and the other half to configure the next one [Ganesan and Vemuri 2000]. On some reconfigurable architectures (e.g., the XPP [Baumgarte et al. 2003]) three stages are needed to execute each configuration: fetching (loading the configuration data from an external memory to the on-chip cache); configuring (loading the configuration data from the on-chip cache directly to the array resources[9]); and computing or running the active configuration.

Given the latencies associated with fetching and loading configurations, it is critical for these systems to rely on advanced compiler analyses and optimization algorithms

[8]Even when using multicontext devices, loading configuration data onto the inactive planes takes several clock cycles.
[9]This step can directly activate array resources to start running or can refer to the storing of each array resource configuration in "shadow planes" (context planes) each time a resource is not free to accept another configuration.

that can minimize the number of configurations needed and their reconfiguration such that the overall execution time is minimized [Fekete et al. 2001]. Prefetching, is a common technique for hiding some of the latency associated with the loading and fetching of configurations, and is used whenever the configurations can be scheduled in a deterministic fashion. Despite the obvious relationship to compilation techniques, the aspects of scheduling and managing configurations are omitted in this survey, as they are commonly seen as operating system or runtime-system issues.

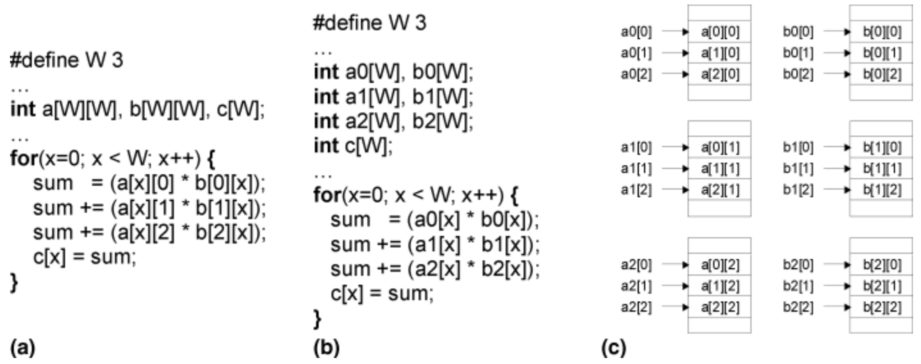
5.5. Memory Mapping Optimizations

Today's reconfigurable computing platforms include a hierarchy of memories with variable sizes and access speeds, and with variable numbers of ports and different depth/width configurations. The next sections describe important memory optimizations.

5.5.1. Partitioning and Mapping of Arrays to Memory Resources. Manual mapping of the program data-structures onto the large number of available physical memory banks is too complex, and exhaustive search methods are too time-consuming. Although the generation of a good memory mapping is critical for the performance of the system, very few studies have been done on the automatic assignment of data structures to complex reconfigurable computing systems, especially research that considers certain optimizations such as allocating arrays across memory banks. Compiler techniques can be used to partition the data structures and to assign each partition to a memory in the target architecture. Data partitioning has long been developed for distributed memory multicomputers where each of its units accesses its local memory (see, e.g., Ramanujam and Sadayappan [1991]). All the work done in this area is therefore applicable to reconfigurable architectures.

Arrays can be easily partitioned when the memory is disambiguated at runtime. However, this form of partitioning has limited use, does not guarantee performance improvements, and requires hardware structures to select from different possible memories at runtime. Hence, schemes to guarantee data allocation into different memories such that each set is disambiguated at compile time can lead to significant performance improvements. One of these techniques is known as bank disambiguation, and is a form of intelligent data partitioning. Concerning reconfigurable computing, bank disambiguation has been performed in the context of compilation to the RAW machine [Barua et al. 2001] and to a version of the RAW machine with custom logic in place of the RISC microprocessor [Babb et al. 1999]. Such a scheme permits us to discover, at compile time, subsets of arrays or data accessed by pointers that can be partitioned into multiple memory banks. The idea is to locate the data as close as possible to the processing elements using it. These techniques can perform a very important role when compiling to coarse-grained (e.g., XPP [Baumgarte et al. 2003]) and fine-grained (e.g., Virtex-II [Xilinx 2001] and Stratix [Altera 2002]) reconfigurable computing architectures, as they include various on-chip memory banks.

Figure 17 gives an illustrative example: Unrolling the inner loop results in the source code shown in Figure 17(a). From the unrolled version, it is possible to perform data partitioning into smaller arrays (Figure 17(b)), having in mind the distribution of data into different memories (bank disambiguation, see Figure 17(c)). The transformation with subsequent memory mapping of the six arrays into six individually accessed memories permits access in each iteration of the loop to all six data elements at the same time, and thus increases the number of operations performed per clock cycle. Barua et al. [2001] present an automatic technique called modulo unrolling to unroll by the factor needed to obtain disambiguation for a specific number of memory



```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
    sum  = (a[x][0] * b[0][x]);
    sum += (a[x][1] * b[1][x]);
    sum += (a[x][2] * b[2][x]);
    c[x] = sum;
}
```

(a)

```
#define W 3
...
int a0[W], b0[W];
int a1[W], b1[W];
int a2[W], b2[W];
int c[W];
...
for(x=0; x < W; x++) {
   sum  = (a0[x] * b0[x]);
   sum += (a1[x] * b1[x]);
   sum += (a2[x] * b2[x]);
   c[x] = sum;
}
```

(b)

Fig. 17. Bank disambiguation as a form to increase the memory bandwidth and the ILP degree when targeting architectures that support various memory banks accessed at the same time: (a) original C code; (b) C code after unrolling and bank disambiguation; (c) arrays mapped to six memories and the correspondence of each array element to the original arrays.

banks. The technique can be used not only to improve performance but also show that array variables of a size surpassing each on-chip memory capacity can be efficiently partitioned so that each partition fits a single memory.

Although reconfigurable computing architectures usually add other optimization opportunities based on the customization of memory size, width, and address generator unities, some relations exist with techniques previously proposed by the high-performance computing community, such as the compilation for distributed memory parallel systems [Gokhale and Carlson 1992].

HLS for ASICs assumes that the hardware is custom-built for the application and that the minimization of the number of memory banks and the size of each memory are important goals. In *reconfigware*, an architecture is implemented in the available resources, the number and type of memory banks is known *a priori*, and the memory locations could be used as long as memory space is available. Some related issues remain however, such as the techniques used in the context of HLS for ASICs to map each array element in memory positions in order to simplify the address generators (less area and delay) [Schmit and Thomas 1998]. Also related are the methods for mapping arrays to memories which consider the presence of multiple accesses at the same time in the code due to the exposition of both data parallelism [Ramachandran et al.; Schmit and Thomas 1997] and functional parallelism [Khouri et al. 1999].

A number of approaches have addressed some of these features. The work of Gokhale and Stone [1999] presents a method to map arrays to memories based on implicit enumeration, but considers only one level of memories of equal type and size and external to the FPGA. Weinhardt and Luk [2001a] describe an integer linear programming approach for the inference of on-chip memories with the goal of reducing the memory accesses, but in the context of loop pipelining and vectorization. The work by Ouaiss and Vemuri [2001] is one of the first attempts to target a reconfigurable memory hierarchy. They use an integer linear programming approach to find the optimal mapping of a set of data segments to the memories. They neither consider all the possible characteristics of the memory resources nor the impact of a specific mapping on the scheduler, and the approach requires long runtimes. Lastly, the work by Baradaran and Diniz [2006] uses array data access pattern compiler knowledge along low-level critical path and scheduling information in order to make better decisions of what to cache in RAMs or registers in the context of scalar replacement and loop-based transformations.

5.5.2. Parallelization of Memory Accesses. In many reconfigurable systems, optimizations related to memory accesses are important contributions to decreasing the execution time. While some systems have dedicated units to fetch data from memory, others must implement a specialized memory access controller. This controller typically deals with the vagaries of the physical interface (e.g., data pins and timing or the hardware protocols) and is customizable for the width and number of memory access paths. Several approaches used to increase the performance of the memory interfaces in the context of reconfigurable computing are now surveyed, paying particular attention to the unique opportunities offered by customization.

As with traditional systems, parallel memory accesses increase the memory bandwidth. Reconfigurable computing systems can exploit this opportunity by defining different memory ports to interface with the available memory banks in a parallel fashion.

5.5.3. Customization of Memory Controllers. Reonfigurable architectures offer a unique opportunity to address the memory access and interfacing issues via customization. Improvements can be achieved by creating specialized hardware components for generating addresses and packing and unpacking data items (see next section). Compilers can analyze memory access patterns for pipelining of array references across multiple iterations of loops. Furthermore, the compiler can derive information about the relative rates among various array references and embed that knowledge into the scheduling of memory operations.

Baradaran et al. [2004] focus on the design, implementation and validation of external memory interfacing modules that are generated by a compiler and a HLS tool that translates high-level computations directly to FPGA hardware. For maximum generality, they have separated the memory interface into two components. The first component is target-dependent, and captures the specific target architecture timing requirements for accessing memory. The second component is architecture-independent, and provides a set of channels and memory access mode abstractions for the application to store and retrieve data from memory. In particular, the interface schemes they have implemented allow compiler-generated designs to exploit pipelined access modes and to mix both pipelined and nonpipelined memory access modes. This approach also allows the development of application-specific scheduling optimizations [Park and Diniz 2001].

5.5.4. Packing/Unpacking Data Items. An important feature of reconfigurable architectures is the ability to define operations over nonstandard data type formats. In terms of memory operations, further gains can be exploited by packing and unpacking data items that are smaller than the basic memory transfer unit into a single memory transfer format. For example, if a computation operates on consecutive elements of an array of 5 bit elements and each memory operation transfers 32-bit words, the compiler can organize the data layout of the array so that 6 data items are packed into 32 bits in memory. Figure 18 illustrates the physical mapping of an array into memory using padding of the least two significant bits. Several compiler analysis techniques have been developed both in the context of distributed memory multi-computers as well as reconfigurable architectures to exploit these opportunities [Rivera and Tseng 1998].

The typical implementation uses "conversion FIFOs" to pack and unpack the data into the 32-bit external data format and allows the datapath to consume the data as if it were composed of 6 consecutive 5-bit elements.

```
int:5 a[N]; // 5 bit integer
if((N % 6) == 0) {
    for(i=0; i < N; i += 6) {
        ... = a[i+0];
        ... = a[i+1];
        ... = a[i+2];
        ... = a[i+3];
        ... = a[i+4];
        ... = a[i+5];
    }
}
(a)
```

```
int a32[N];
if((N % 6) == 0) {
    for(i=0; i < N; i += 6) {
        w = a32[i];
        ... = (w & 0x0000001F):5;
        ... = ((w & 0x000003E0) >> 5):5;
        ... = ((w & 0x00007C00) >> 10):5;
        ... = ((w & 0x000F1000) >> 15):5;
        ... = ((w & 0x01F00000) >> 20):5;
        ... = ((w & 0x3E000000) >> 25):5;
    }
}
(b)
```

Fig. 18. Packing and unpacking example of 5-bit items into 32-bit words: (a) original code; (b) code using unpacking.

```
...
for(int i=3; i<N; i++) {
    y[i] = y[i-1]*w1+
           y[i-2]*w2+
           y[i-3]*w3;
}
...
(a)
```

```
int D3, D2=0, D1=0;
...
for(int i=1; i<N; i++) {
    D3 = D2;
    D2 = D1;
    D1 = y[i-1];
    If(i>2)
        y[i] = D1*w1+D2*w2+D3*w3;
}
...
(b)
```

Fig. 19. Example of the use of a tapped-delay line for reducing the number of memory accesses: (a) original code; (b) transformed code.

5.5.5. Elimination of Memory Accesses using Register Promotion. This mapping technique exploits the reuse of array values across iterations of a given loop and is typically used in combination (either explicit or implicit) with the scalar replacement transformations described in Section 4. The overall objective is to eliminate redundant memory accesses across loop iterations by reusing data accessed in earlier iterations, and saved in scalar variables which are then promoted to registers. Figure 19 shows an example of the application of this technique, enabling the reduction of the number of memory loads from $3\times$N-9 to N-1. The transformed code (Figure 19(b)) can be synthesized to a tapped-delay line where the various taps correspond to the scalar variables D1, D2, and D3.

While this example exploits the reuse in scalar variables with a tapped-delay line, it is also possible to reuse the data using a RAM module. In the latter case the delay line is conceptually implemented using a read and write pointer RAM address. The implementation typically uses many fewer resources, but all the data elements of the tapped-delay line are not immediately available, which can be a substantial disadvantage. The work by Baradaran et al. [2004] explores the space and time tradeoffs for these alternative implementations.

5.6. Scheduling

Scheduling is the process of assigning operations to a discrete time step, usually a clock cycle, to a specific, and possibly shared, hardware resource or FU. Typically, the scheduler generates state transition graphs (STGs) that model a control unit to coordinate the memory accesses, the execution of the loops, and the sharing of hardware resources. From the STGs, a scheduler can easily generate a finite state machine based on one-hot encoding to implement the controller for the state transition graph.

There has been work that exposes some scheduling concepts to the programmer. An example is the approach presented by Page [1996] which considers compilation for Handel-C descriptions. The mapping assigns a register to each variable in the source code, thereby chaining all operations in an expression in one clock cycle. To control pipelined execution, the user must use auxiliary variables at operation level.

Other work has focused on the generation of specific hardware implementations able to execute concurrent loops without dependences in the context of HLS for ASICs. [Lakshminarayana et al. 1997] describe a static scheduling algorithm that generates control units to coordinate the parallel execution of such concurrent loops. Ouaiss and Vemuri [2000] present an arbitration scheme to deal with concurrent accesses to the same unit at runtime.

Traditionally, HLS has focused on the exploitation of resource sharing with the goal of achieving optimum performance results within a predefined silicon area. The resources constraints are typically specified by inputting the number of FUs for each operation type. Some approaches use conditional code motion beyond basic blocks [Gupta et al. 2001; Santos et al. 2000] that might reduce the schedule length without an increase in area. As simple primitive operations seldom justify the effort of sharing a component, compilers attempt to perform code motion and aggregate several instructions in a single basic block to improve the profitability of resource sharing.

To enhance the scope of the scheduling phase, some authors use regions of basic blocks. Examples of such regions are the use of the *hyperblock* and the approach presented in Cardoso and Neto [2001]. In this later approach, basic blocks are merged across loop boundaries. The scheduler works at the top level of a hierarchical task graph (HTG) and merges blocks at loop boundaries based on an ASAP (as soon as possible) scheme. For each HTG, the scheduler uses a static-list scheduling algorithm [Gajski et al. 1992] at the operation level.

5.7. Back-End Support

The back-end support is related to the generation of the datapath and control unit structures (based on the granularity of the target reconfigurable architecture) and the configuration data (also known as bitstream in the FPGA domain) to be processed by the reconfigurable computing platform.

For generating the hardware structure when a fine-grained architecture is targeted, three distinct approaches can be used: HLS tools, logic synthesis tools, or module generators. Note, however, that the use of HLS tools needs the support of logic synthesis and/or module generators.

Some compilers use back-end commercial logic synthesis and placement and routing tools. Such an approach has been adapted from the traditional design flow for ASICs, and has the long runtimes and the inevitability of independent phases as its major disadvantages. The compiler and the logic synthesis tool are connected by generating the description of the architecture in a textual model accepted by those tools. Most of them accept popular HDLs (such as VHDL or Verilog) at RTL (register-transfer level) in structural and/or behavioral form. Other approaches use a high-level back-end, such as an HLS tool (e.g., Bondalapati et al. [1999]). In those approaches, the compiler has the goal of translating among semantic differences between, for example, software programming languages and hardware description languages.

Module generators can be used efficiently to generate the hardware structure of each operation. Their use in some cases outperforms logic synthesis in terms of hardware resources and delays and reduces the compilation time considerably. Examples of compilers using module generators are the Nenya [Cardoso and Neto 1999]; the *garpcc* [Callahan et al. 2000]; the NAPA-C [Gokhale and Stone 1998]; and the DIL compiler [Budiu and Goldstein 1999]. The DIL compiler expands the module structures to perform placement and routing. It performs some optimizations on each specific instance of a module according to the information about input/output variables (bit-width, bit constants, etc.). One of the tools that can be used as a module generator is the JHDL framework [Bellows and Hutchings 1998], which allows us to describe the datapath at RTL in a subset of Java extended by special APIs [Wirthlin et al. 2001]. Such a description can be independent of the target FPGA architecture, and thus requires the use of mapping, placement, and routing to generate the configuration data.

Another approach is the use of module generators for a particular FPGA. Such generators use a structural description based on the operations of each FPGA cell, and thus do not need the mapping phase. Such structures can contain preplacement information that can reduce the overall placement and routing time. Jbits [Guccione et al. 2000] is a tool to automatically generate bitstreams without heavy commercial P&R tools. Jbits has already been used as a back-end in a compiler environment [Snider et al. 2001]. This compiler performs some low-level optimizations, which are done at the LUT level (the Xilinx Virtex FPGA was used), and include LUT/register merging and LUT combining. The approach shows a compilation time of seconds to compile an example of medium complexity.

Most of the compilers to *reconfigware* do not use complex logic synthesis steps for generating the control structure (the main reason is the use of relatively uncomplicated control units). This is different from HLS techniques when targeting ASICs, which use heavy sharing of FUs, and thus need complex control units [Gajski et al. 1992]. Some compilers to *reconfigware* use templates associated with each language constructor and a combination of trigger signals to sequence the hardware execution (e.g., Wirth [1998]).

For coarse-grained architectures, where most of the operations in the source language are directly supported by the FUs existing in each cell of the target architecture, neither a circuit generator nor logic synthesis are needed to generate the datapath. In such architectures, the mapping phase is also much easier. However, in architectures supporting control structures, the control generation is mostly conducted by the compiler itself. For the generation of the bitstreams, proprietary placement and routing tools are used. Such architectures reduce the compilation time tremendously because they require less complex back-end phases (see the XPP-VC compiler [Cardoso and Weinhardt 2002]).

6. COMPILERS FOR RECONFIGURABLE ARCHITECTURES

The next section lists the most prominent research efforts on compiling from high-level languages to reconfigurable architectures. The maturity of some of these techniques and (some) stability in the underlying reconfigurable technology enabled the emergence of commercial companies with their own technical compilation solutions. They help to port applications written in high-level programming languages to reconfigurable devices. Although this survey does not describe their technical solutions in detail, two commercial efforts are noteworthy, namely SRC [SRC Computers Inc.] and Nallatech [Nallatech Inc.]. Both support the mapping of programs written in a subset of the C programming language to FPGAs.

In this section we distinguish efforts that target fine-grained RPUs (reconfigurable processing units), for example, FPGAs, from the ones targeting coarse-grained RPUs. Despite the, sometimes radical difference in terms of granularity, there are many commonalities between the many techniques used to map computations to these architectures. We begin with a brief description of the various, more representative, compilation efforts. Many of these efforts, while similar in terms of the input language they handle, differ substantially in the techniques or granularity used. Many aspects of these compilers are highlighted in Table IV through Table XIII at the end of this section.

6.1. Compilers for FPGA-Based Systems

This section focuses on compilation efforts that exclusively target FPGA-based systems through a wide variety of approaches and for a wide range of input programming languages and computational models. First, some earlier and, for that reasons, less mature compilation efforts are described.

6.1.1. Early Compilation to FPGA Efforts. One of the first efforts on the compilation of computations to FPGAs considered simple translation schemes of OCCAM programs [Page and Luk 1991] directly into hardware. Other simple approaches have been reported such as Transmogrifier C [Galloway 1995] and the compiler by Wo and Forward [1994], both considering a C-subset and a syntax-oriented approach [Wirth 1998]. Some authors considered the use of explicitly parallel languages expressing concurrency at the operation level. One such approach is presented by Page [1996], which considers compilation from Handel-C descriptions.

The PRISM I-II [Agarwal et al. 1994; Athanas and Silverman 1993] approach was one of the first to aim at compiling from a subset of the C programming language for a target architecture with software and *reconfigware* components [Athanas 1992]. It focused on low-level optimizations (at the gate or logic level). Schmit et al. [1994] describe one of the first approaches to target FPGAs using HLS tools. Their approach focused on spatial partitioning at both behavioral and structural domains. Other authors have used commercial HLS tools to define a specific architecture using the FPGA substrate, as in the example in Doncev et al. [1998], and then map the computation at hand to it.

6.1.2. Compilation to FPGAs Comes of Age. After the earlier and arguably more timid efforts, a growing interest in academia for compilation efforts from high-level languages to FPGAs has been seen. Rather than forcing the programmers to code in hardware-oriented programming languages, many research efforts took the pragmatic approach of directly compiling high-level languages, such as C and Java, to hardware. Most of these efforts support mechanisms to specify hardware and produce behavioral RTL-HDL descriptions to be used by a HLS or a logic synthesis tool. Initially, most efforts focused on ASICs but due to the increasing importance of FPGAs, many compilation efforts explicitly targeting FPGA devices have been developed. While initially many of these efforts originated in academia (e.g., Weinhardt and Luk [2001b] and Nayak et al. [2001a]), there was also a great deal of interest shown by industry. As a result, tools (e.g., the FORGE compiler) especially designed for targeting FPGAs became a reality.

A representative selection of the most noteworthy efforts for compiling from high-level languages to reconfigurable architectures is now presented. The selection starts with efforts to compile C programs and continues by showing efforts that consider other languages (e.g., MATLAB or Java). At the end, two different approaches are described: one considers a component-based approach and the other is more related to HLS, since it starts with VHDL specifications.

6.1.3. The SPC Compiler. The SUIF pipeline compiler (SPC) focuses on automatic vectorization techniques [Weinhardt and Luk 2001b] for the synthesis of custom hardware pipelines for selected loops in C programs. The SPC analyzes all program loops, in particular innermost loops without irregular loop-carried dependences. The pipelines aggressively exploit ILP opportunities when executing the loops in hardware. Address generators are synthesized for accesses to program arrays (multidimensional), and shift-registers are used extensively for data reuse. Pipeline vectorization takes advantage of several loop transformations to meet hardware resource constraints while maximizing available parallelism. Memory allocation and access optimizations are also included in the SPC [Weinhardt and Luk 2001a].

6.1.4. The Compiler from Maruyama and Hoshino. Maruyama and Hoshino [2000] developed a prototype compiler that maps loops written in the C programming language to fine-grained pipeline hardware execution. Their compiler splits the arithmetic operations of the input program into cascading 8-bit-wide operations (a transformation known as decomposition) for higher pipelining throughput. It uses speculative execution techniques to allow it to initiate the execution of a loop iteration while the previous iteration is still executing. In the presence of memory bank dependences the pipeline is stalled and the accesses are serialized. Feedback dependences, either through scalar or array variables, cause speculative operations in the pipeline to be cancelled, but restarted after the updates to the arrays complete. The compiler also supports the mapping of some recursive calls by converting them internally to iterative constructs. In addition, the compiler front-end supports parallelization annotations that allow the compiler to perform task and data partitioning, but few technical details are available.

6.1.5. The DeepC Silicon Compiler. The DeepC silicon compiler presented in Babb [2000] and Babb et al. [1999] maps C or Fortran programs directly into custom silicon or reconfigurable architectures. The compiler uses the SUIF framework and relies on state-of-the-art pointer analysis and high-level analyses techniques such as bit-width analysis [Stephenson et al. 2000]. Many of its passes, including parallelization and pointer-analysis, have been taken directly from MIT's RAW compiler (Rawcc) [Barua et al. 2001; Lee et al. 1998]. The compiler targets a mesh of tiles where each tile has custom hardware connected to a memory and an inter-tile communication channel. This target architecture is inspired by the RAW machine concept, but where the processing element used in each tile is custom-logic (reconfigurable or fixed) instead of a microprocessor. The compiler performs bank disambiguation to partition data across the distributed memories attempting to place data in the tile that manipulates it. A specific architecture consisting of an FSM and a data-path is then generated for each tile. Besides the generation of the architecture for each tile, the compiler also generates the communication channels and the control unit, which is embedded in each FSM, to communicate data between tiles. The compiler supports floating-point data types and operations. Each floating-point operation in the code is replaced by a set of integer and bit-level micro-operations correspondent to the realization of its floating-point unit. This permits optimizing some operations dealing with floating-point data. Finally, the DeepC compiler generates technology-independent Verilog behavioral RTL models (one for each tile) and the circuitry is generated with back-end commercial RTL synthesis and placement and routing tools.

6.1.6. The COBRA-ABS Tool. The COBRA-ABS tool [Duncan et al. 1998, 2001] synthesizes custom architectures for data-intensive algorithms using a subset of the C programming language. This tool performs spatial partitioning, scheduling, mapping, and allocation using simulated annealing techniques. The target architecture for the COBRA-ABS tool is a multi-FPGA system with various arithmetic ASICs, each of which with its own local memory. The system is programmed in a VLIW style using a large register file and a centralized control unit. The system uses commercial synthesis tools for placement and routing to generate the various FPGA configurations. This tool uses a HLS approach that targets ASICs. Since the HLS is based on a simulated-annealing approach, the reported synthesis time is prohibitively long. For example, a simple design with 13 operations and 12 memory accesses took approximately 12 hours on a Sun Ultra 1/140 to synthesize.

6.1.7. The DEFACTO Project. The DEFACTO (design environment for adaptive computing technology) project [Bondalapati et al. 1999] is a system that maps computations, expressed in high-level imperative languages such as C and FORTRAN, directly to FPGA-based computing platforms such as the Annapolis WildStar^{TM} FPGA-based board [Annapolis 1999]. It partitions the input source program into a component that executes on a traditional processor and another component that is translated to behavioral, algorithmic-level VHDL to execute on one or more FPGAs. DEFACTO combines parallelizing compiler technology with commercially available high-level VHDL synthesis tools. DEFACTO uses the SUIF system and performs several traditional parallelizing compiler analyses and transformations such as loop unrolling, loop tiling, data and computation partitioning. In addition, it generates customized memory interfaces and maps data to internal storage structures. In particular, it uses data reuse analysis coupled with balance metrics to guide the application of high-level loop transformations. When doing hardware/software partitioning, the compiler is also responsible for automatically generating the data management (copying to and from the memory associated with each of the FPGAs), and synchronizing the execution of every FPGA in a distributed memory parallel computing style of execution.

6.1.8. The Streams-C Compiler. Streams-C [Gokhale et al. 2000b] relies on a communication sequential processes (CSP) parallel programming model [Hoare 1978]. The programmer uses the annotation mechanism to declare processes, streams, and signals. In Streams-C, a process is an independently executing object with a process body specified by a C routine, and signals are used to synchronize their execution. The program also defines the data streams and associates a set of input/output ports to each process, explicitly introducing read and write operations via library primitives.

The Streams-C compiler builds a process graph decorated with the corresponding data stream information. It then maps the computation on each process to an FPGA and uses the MARGE compiler [Gokhale and Gomersall 1997] to extract and define the specific datapath VHDL code (structural RTL model) from the AST (abstract syntax tree) of the C code that defines the body of each process. The compiler inserts synchronization and input/output operations by converting the annotations in the source C, first to SUIF internal annotations and then to library function calls. The MARGE compiler is also responsible for scheduling the execution of loops in each process body in a pipelined fashion. Performance results are reported by Frigo et al. [2001].

6.1.9. The CAMERON Project. The Cameron compiler [Böhm et al. 2001, 2002] translates programs written in a C-like single-assignment language, called SA-C, into dataflow graphs (DFGs) and then onto synthesizable VHDL designs [Rinker et al. 2001]. These DFGs typically represent the computations of the body of the loops and have nodes that represent the input/output of data, as well as nodes corresponding to macro-node operations. These macro-operations represent more sophisticated operations such as the generation of the window inside a given array, defining which elements of the array participate in the computation. The compiler then translates each DFG to synthesizable VHDL, mostly relying on predefined and parameterizable library macro templates. A back-end pass uses commercial synthesis tools, such as [Synplicity]'s Synplify, to generate the FPGA's bitstreams.

The resulting compiled code has both a component that executes on a traditional processor as well as one executing on computing architectures with FPGA devices. SA-C is a single-assignment (functional) language intended to facilitate the development of image processing applications and its efficient translation to hardware. Its features include the specification of variables' bit-widths and the use of syntax mechanisms to specify certain operations over one- or two-dimensional arrays, enabling the compiler to create specific hardware structures for such operations. The specification of reduction operators also allows the programmer to directly to hint to the compiler about the appropriate efficient implementations for these operators in VHDL.

The SA-C compiler applies a series of traditional program optimizations (code motion, constant folding, array and constant value propagation, and common-subexpression elimination). It also includes other transformations with a wider scope that can be controlled by pragmas at the source-code level. These include array and loop bound propagation, so that the maximum depth of loop constructs could be easily inferred from the array these constructs manipulate. Bit-width selection for space and tiling to expose more opportunities for ILP are also considered by the compiler.

This work addresses an important aspect of mapping applications to reconfigurable architectures by focusing on a particular domain of applications and developing language constructs specific for that domain. The core of the compilation is devoted to using the information in pragmas and matching them with the predefined library templates using the semantics of the looping and window constructs.

6.1.10. The Match Compiler. The MATCH[10] [Banerjee et al. 2000; Nayak et al. 2001a] compiler accepts MATLAB descriptions [Mathworks] and translates them to behavioral RTL-VHDL [Haldar et al. 2001b], subsequently processed by commercial logic synthesis and placement and routing tools. MATCH includes an interface for dealing with IP (intellectual property) cores. An IP core database is used which includes the EDIF/HDL representation of each IP, the performance/area description, and the interface to the cores. The instantiated cores are integrated during the compilation. As MATLAB is not a strongly typed and shaped language, the translation process includes type and shape[11] analysis (directives can be used by the user). The result of these analyses is annotated in the AST. Matrix operations are then expanded into loops (scalarization). Then, parallelization is conducted (loops can be split onto different FPGAs and run concurrently), which generates a set of partitioned ASTs with embedded communication and synchronization. The next step is to perform precision analysis by transforming floating-point and integer data types to fixed-point and inferring the number of bits needed. The compiler then integrates the IP cores (they can be FUs for each operation in the AST or optimized hardware modules for specific functions called in the AST) and creates a VHDL-AST representation with the optimized cores.

[10]A transfer of technology has been carried out to the startup company, AccelChip, Inc. ACCELCHIP http://www.accelchip.com/.
[11]Shape analysis refers to the inference about the number of dimensions of a Matlab variable (matrix).

Another step is related to pipelining and scheduling. The compiler checks if a loop[12] can be pipelined. If so, a DFG with predicated nodes is generated and the memory accesses are scheduled based on the number of ports in each memory. The compiler tries to pipeline the memory accesses, which is frequently the performance bottleneck. During this task, the compiler uses the delays of each IP core. Modulo scheduling is used to exploit loop pipelining parallelism. Finally, the compiler translates the VHDL-AST onto a VHDL-RTL description used by commercial logic synthesis tools to generate the netlist later placed and routed by the specific FPGA tools.

6.1.11. The Galadriel and Nenya Compiler. The GALADRIEL compiler translates Java bytecodes to an architectural synthesis tool that is specific for reconfigurable computing platforms (NENYA) [Cardoso and Neto 1999, 2003], comprising an FPGA connected to one or more memories. The compiler accepts a subset of Java bytecodes that enables the compilation of representative algorithms, specified in any language that can be compiled to the Java virtual machine (JVM). NENYA extracts hardware images from the sequential description of a given method to be executed by the FPGA. For designs larger than the size of the FPGA, the approach addresses temporal partitioning techniques in the compilation process.

The compiler exposes several degrees of parallelism (operation, basic block, and functional) [Cardoso and Neto 2001]. The method being compiled is fully restructured according to the control and data dependences exposed (loop transformations and loop pipelining were not included). The compiler generates, for each temporal partition, a control unit and a datapath module, including memory interfaces. The control unit is output in behavioral RTL-VHDL and the datapath in structural RTL-VHDL.

The scheduler uses fine-grained timing slots to schedule operations, considers different macro-cell alternatives to implement a given functionality, and takes into account the bit-width of the operands for the area/delay estimation (each component in the hardware library is modeled by functions obtained by line/curve fitting). Earlier versions of the compiler targeted the Xilinx XC6200 FPGA using circuit generators. When such generators are not available for a specific FPGA, commercial logic synthesis is used for the preback-end phase [Cardoso and Neto 2003]. Finally, the FPGA bitstreams are generated using vendor-specific placement and routing tools.

6.1.12. The Sea Cucumber Compiler. Sea Cucumber (SC) [Tripp et al. 2002] is a Java to FPGA hardware compiler that takes a pragmatic approach to the problem of concurrency extraction. It uses the standard Java thread model to recognize task-level or coarse-grained parallelism as shown by programmers. In the body of each thread, the SC compiler then extracts the fine-grained parallelism using conventional controlflow and dataflow analysis at the statement level and across multiple statements. Once the CFG is extracted, a series of low-level instruction-oriented transformations and operations are performed, such as predication to allow it to generate hardware circuit specifications in EDIF using JHDL [Bellows and Hutchings 1998], which are then translated to Xilinx bitstreams. As described in the literature, this compiler does not aim at the automatic discovery of loop-level or cross-iteration parallelism, as is the goal of other compilers that map imperative programs to hardware.

6.1.13. The HP-Machines Compiler. Researchers at the Hewlett Packard Corp. developed a compiler to deal with a subset of C++ with special semantics [Snider et al. 2001].

[12]Loop constructs present in the Matlab code are converted to finite state machines (FSM).

This approach uses an abstract state machine concurrent programming model, called "Machines". An application can be specified by a set of state machines, and parallelism can be represented explicitly. The user specifies the medium-grained parallelism and the compiler extracts the fine-grained parallelism (ILP). The compiler is capable of high- and low-level optimizations such as bit-width narrowing and bit optimizations to reduce to wire concatenation operations. The compiler maps different Xilinx FPGAs by using the JBits [Guccione et al. 2000] tool as a back-end.

6.1.14. The SPARCS Framework. Other authors have adapted traditional HLS frameworks to target reconfigurable computing systems. In the SPARCS framework [Ouaiss et al. 1998a], temporal and spatial partitioning are performed before HLS. The system starts from a task-memory graph and a behavioral, algorithmic level VHDL specification for each task. The unified specification model (USM) [Ouaiss et al. 1998b] is used as an intermediate representation. Although the system resolves important issues related to temporal/spatial partitioning and resource sharing, it suffers from long run times because it is dominated by design space exploration. The target of the tool is a platform with multiple FPGAs and memories, for which they use commercial logic synthesis and placement and routing tools. The interconnections among FPGAs and memories can be static or reconfigurable at runtime.

6.1.15. The ROCCC Compiler. The Riverside optimizing compiler for reconfigurable computing (ROCCC) is a C to hardware compiler that focuses on FPGA-based acceleration of frequently executed code segments, most notably loop nests [Guo et al. 2004]. It uses a variation of the SUIF2 and MachSUIF compilation infrastructure, called compiler intermediate representation for reconfigurable computing (CIRRF) [Guo and Najjar], where streaming-oriented operations are explicitly represented via internal primitives for buffer allocation and external memory operation scheduling. This representation is particularly useful in the context of window-based image and signal processing computations [Guo et al. 2004].

6.2. Compilers for Coarse-Grained Reconfigurable Architectures

Many other research and commercial projects took the alternative route of considering proprietary *reconfigware* and used their own tools from the language specification down to the placement and routing phases. Examples of such efforts are the compilers for the PipeRench, RaPiD, Xputer, and XPP.

6.2.1. The DIL Compiler for PipeRench. The PipeRench compiler (called the DIL compiler) [Budiu and Goldstein 1999] maps computations described in a proprietary intermediate single-assignment language, called the dataflow intermediate language (DIL), into the PipeRench. The PipeRench consists of columns of pipeline stages (stripes), and the model of computation permits pipelining the configuration of the next strip to be used with the execution of the current one.

DIL can be viewed as a language to model an intermediate representation of high-level language descriptions such as C, and is used to describe pipelined combinatorial circuits (unidirectional dataflow). DIL has C-like syntax and operators and uses fixed-point type variables with bit widths that can be specified by the programmer as specific bit-width values, or alternatively, derived by the compiler. The approach mainly delegates to the compiler the inference of bit-widths for the data used [Budiu et al. 2000]. The only bit-widths that must be specified are those related to input and output variables.

The compiler performs over 30 analyses steps before generating the configuration model to be executed in the PipeRench. Each arithmetic operator used in the DIL program is specified in a library of module generators. Those modules are described in DIL as well. The compiler expands all the modules (operator decomposition) and functions and fully unrolls all the loops presented in the application. Hence, a straight-line single assignment program is generated. Then, a hierarchical acyclic DFG is constructed. The DFG has nodes representing operations, I/O ports, and delay-registers. Each operator node can be a DFG itself. After the generation of the global DFG, the compiler performs some optimizations. Such optimizations include traditional compiler optimizations (e.g., common subexpression elimination, algebraic simplifications, and dead code elimination), a form of retiming known by register reduction and interconnection simplification. One of the main strengths of the compiler is to perform such optimizations through the overall representation (also considering the representation of each module used by each operator). Hence, it specializes and optimizes the proper instance of each module structure based on the bit-width of the input/output variables and on the constant values of input variables.

The placement and routing (P&R) phase is done through the DFG using a deterministic linear-time greedy algorithm, based on list scheduling [Cadambi and Goldstein 2000]. This approach leads to implementations that are two or three orders of magnitude faster than the P&R in commercial tools.

6.2.2. The RaPiD-C Compiler. To facilitate the specification and mapping of computations to the RaPiD reconfigurable architecture [Ebeling et al. 1995], the RaPiD project also developed a new concurrent programming language, RaPiD-C [Cronquist et al. 1998], with a C-like syntax flavor. RaPiD-C allows the programmer to specify parallelism, data movement, and data partitioning across the multiple elements of the array. In particular, RaPiD-C introduces the notion of space-loop, or *sloop*. The compiler unrolls all of the iterations of the *sloop* and maps them onto the architecture. The programmer, however, is responsible for permuting and tiling the loop to fit onto the architecture, and is also responsible for introducing and managing the communication and synchronization in the unrolled versions of the loops. In particular, the programmer must assign variables to memories and ensure that the correct data is written to the RAM modules by using the language-provided synchronization primitives, *wait* and *signal*.

In terms of compilation, the RaPiD-C compiler extracts from each of the loop constructs a control tree for each concurrent task [Cronquist et al. 1998]. A concurrent task is defined by a *par* statement in the language. Each task control tree has, as interior nodes, sequential statements (defined by the *seq* construct) and *for* statements. At the leaves, the control tree has *wait* and *signal* nodes. The compiler inserts synchronization dependences between *wait* and *signal* statements.

During compilation, the compiler inserts registers for variables and ALUs for arithmetic operations, in effect creating a DFG for the entire computation to be mapped to a stage in the architecture. The compiler then extracts address and instruction generators that are used to control the transfer of values between the stages of the architecture during execution. In terms of dynamic control extraction, the RaPiD-C compiler uses two techniques, called multiplexer-merging and FU merging, respectively. In the multiplexer merging, the compiler aggregates several multiplexers into a single larger multiplexer and modifies the control predicates for each of the inputs, taking into account the netlist of multiplexers created as a result of the implementation of conditional statements. The FU merging takes advantage of the fact that a set of several ALUs can be merged if their inputs are mutually exclusive in time and the union of their inputs does not exceed a fixed number.

The compiler also represents in its control tree occasions in which the control is data dependent, that is, dynamic. In this scenario, the compiler represents this dependency by an event and a condition. These abstractions are translated into hardware resources present at every stage. For example, the first iteration of a loop can be explicitly represented by the *i.first* condition being true only at the first iteration of a *for* loop. Also, *alu.carry* is explicitly represented at the hardware level. The compiler then aggregates these predicates into instructions, generates the corresponding decoding that drives each of the available control lines of the architecture so that the control signals can be present at the corresponding stage during execution.

6.2.3. The CoDe-X Framework. CoDe-X [Becker et al. 1998] is a compilation framework to automatically map C applications onto a target system consisting of a host computer and Xputers [Hartenstein et al. 1996]. Segments of C computations are mapped to the KressArray/rDPA, while the remaining code is compiled to the host system with a generic C-compiler. The CoDe-X compiler uses a number of loop transformations (such as loop distribution and stripmining) to increase the performance of the segment of code mapped to the rDPA. The compiler also performs loop folding, loop unrolling, vectorization, and parallelization of loops when they compute on different stripes of data. The datapath synthesis system (DPSS) framework [Hartenstein and Kress 1995] is responsible for generating the configurations. The DPSS accepts ALE-X, a language to describe arithmetic and logic expressions for Xputers, descriptions which can be generated from C code using the CoDe-X compiler. The rDPA address generator and control units (both coupled to the array) orchestrate loop execution on the Xputer. As the array itself only implements dataflow computations, the compiler converts controlflow statements of the form *if-then-else* to dataflow. The DPSS tool is responsible for scheduling, using a dynamic list-scheduling algorithm, the operations in the statements mapped to the array, with the only constraint being a single I/O operation at each time (the Xputer uses only one bus to stream I/O data). The final step uses a mapper, based on simulated annealing, to perform the placement and routing of the generated datapath onto the array and to generate the configurations for the array and for the controller.

6.2.4. The DRESC Compiler. DRESC (dynamically reconfigurable embedded system compiler) [Mei et al. 2002] is a C compiler that was developed to target ADRES [Mei et al. 2005; Sutter et al. 2006]. The compiler uses as a front-end the [IMPACT] compiler infrastructure. A partitioning step is, with the help of profiling, responsible for identifying the computationally intensive loops in the application. A novel modulo scheduling algorithm, which combines placement, routing, and scheduling, is used to map pipelined loop kernels onto the reconfigurable array. One interesting point of the compilation flow is the possibility of targeting a family of ADRES architectures by changing an architectural description file. Examples were shown for exploiting different routing topologies as well as heterogeneous FUs [Mei et al. 2005].

6.2.5. The XPP-VC Compiler. The XPP vectorizing C compiler (XPP-VC) [Cardoso and Weinhardt 2002] maps C code into the XPP architecture [Baumgarte et al. 2003]. It is based on the SUIF compiler framework and uses new mapping techniques combined with the pipeline vectorization method previously included in the SPC [Weinhardt and Luk 2001b]. Computations are mapped to the ALUs and data-driven operators of the XPP architecture and temporal partitioning is employed whenever input programs require more resources than are available for each configuration.

6.3. Compilers for Tightly Coupled Reconfigware and Software Components

Many research efforts have been tailored to architectures that couple a microprocessor to RPUs (reconfigurable processing units) in the same chip. Most of them address compilation techniques targeting in-house architectures. Some of the most remarkable compilers are the CHIMAERA-C [Ye et al. 2000a], the *garpcc* [Callahan et al. 2000], and the NAPA-C [Gokhale and Stone 1998] just to name a few. The next sections give a brief introduction to these compilers.

6.3.1. The CHIMAERA-C Compiler. The CHIMAERA-C compiler [Ye et al. 2000a] extracts sequences of instructions from the C code, each one forming a region of up to nine input operands and one output. The goal of this approach is to transform small acyclic sequences of instructions, without side-effect operations, in the source code into special instructions implemented in the RPU (which in this case is a simple reconfigurable functional unit) [Hauck et al. 2004]. This approach aims to achieve speed-ups of the overall execution. As the compiler only considers *reconfigware* compilation for the previously described segments of code, it is inappropriate for used as a stand-alone *reconfigware* compiler.

6.3.2. The Compiler for Garp. An ANSI C compiler for the Garp architecture, called *garpcc* [Callahan et al. 2000], was developed at the University of California at Berkeley. It uses the hyperblock[13] [Mahlke et al. 1992] to extract regions in the code (loop kernels) suitable for *reconfigware* implementation, and transforms "if-then-else" structures into predicated forms. The basic blocks included in a hyperblock are merged and the associated instructions are represented in a single DFG, which explicitly exposes the operation-level fine-grained parallelism [Callahan and Wawrzynek 1998]. The compiler integrates *software pipelining* techniques [Callahan and Wawrzynek 2000], adapted from previous work on compilation, to VLIW processors. The compiler uses predefined module generators to produce the proprietary structural description. The *garpcc* compiler relies on the *Gama* tool [Callahan et al. 1998] for the final mapping phases. *Gama* uses mapping methods based on dynamic programming and generates from the DFG a specific array configuration for the reconfigurable matrix by integrating a specific placement and routing phase.

The ideas of the *garpcc* compiler were used by Synopsys in the Nimble compiler [Li et al. 2000]. As *garpcc*, Nimble also uses as a front-end the SUIF framework [Wilson et al. 1994]. The target of the Nimble compiler are devices with an embedded CPU tightly coupled to an RPU. The compiler uses profiling information to identify inner loops (kernels) for implementation on a specific datapath in the RPU, aiming at the overall acceleration of the input program. *Reconfigware*/software partitioning is automatic. The *reconfigware* segment is mapped to the RPU by the ADAPT datapath compiler. This compiler uses module descriptions for generating the datapath, and generates a sequencer for scheduling possible conflicts, for example, memory accesses, and orchestration of loops. The compiler also considers floor-planning and placement during this phase. The datapath structure with placement information is then fed to the placement and routing FPGA vendor tool to generate the bitstreams.

6.3.3. The NAPA-C Compiler. The NAPA-C compiler uses a subset of C with extensions to specify concurrency and bit-widths of integers [Gokhale and Stone 1998]. It targets

[13]A hyperblock represents a set of contiguous basic blocks. It has a single point of entry and may have multiple exits. The basic blocks are selected from paths commonly executed and suitable for implementation in *reconfigware*.

Table IV. Characterization of a Representative Number of Compilation Efforts

			Target Architectures			Domain Specialization	
			FPGAs Commercial	Microprocessor coupled with reconfigurable processing units (RPUs)	Coarse-grained Reconfigurable Architectures	Architecture-Specific	Application-Specific
Programming Model	Sequential	Low-Level	Transmogrifier-C				
		High-Level	PRISM DEFACTO ROCCC Trident-C MATCH SA-C DeepC Galadriel & Nenya	DRESC Code-X Garpcc CHIMAERA-C	DRESC Code-X DIL XPP-VC	DIL	
	Concurrent (e.g., CSP)	Low-Level	SPARCS				
		High-Level	Handel-C Stream-C Sea Cucumber HP-Machines		RaPiD-C	RaPiD-C	Stream-C

a reconfigurable architecture consisting of a microprocessor coupled to *reconfigware* (FPGA resources), both sharing the same memory address space. The user declares the code segment to be compiled to *reconfigware*. The compiler uses the MARGE (malleable architecture generator) tool to generate a datapath [Gokhale and Gomersall 1997]. MARGE uses a library of parameterized macros/modules, including ALUs, counters, encoders, filters, SRAMs, and register banks. Some of the library elements are preplaced and prerouted, thus reducing the total compilation time from hours to minutes [Gokhale and Gomersall 1997]. Macros are generated using the Modgen tool. The logic structure of each macro can be described with a C-like language (structure is defined as a function of the bit-width of the operands, geometric shape is also defined, etc.). The output of MARGE is an RTL model containing Modgen components. With MARGE, FUs are shared by operations whenever possible, that is, every time the FUs are of the same bit-width size, the same functionality, and are not being used by other operations on the schedule cycle. Control generators are used to synthesize control units to orchestrate the *reconfigware* execution. MARGE assigns a register for each scalar variable in the source code. Both the registers to store variables and the auxiliary registers to store temporary values are implemented by banks of registers.

Results are shown using the NAPA-C compiler to target the CLAy^{TM} FPGA from National Semiconductor in Gokhale and Gomersall [1997] and the NAPA (RISC+FPGA) device [Rupp et al. 1998] in Gokhale and Stone [1998]. Gokhale and Stone [1999] show a scheme for automatic mapping of array variables to memory banks. The NAPA-C compiler was ultimately improved to compile stream-based computations, described in the Stream-C language [Gokhale et al. 2000b], to commercially available FPGAs.

6.4. Overview of Compiler Properties

The following tables summarize important properties of the previously discussed compilers. Table IV presents a general characterization of the compilers, which are

Table V. Main Characteristics of Some Compilers: General Information (I)

Compiler	Year (1st pub.)	Affiliation
Transmogrifier-C	1995	Univ. of Toronto, Canada
PRISM-I, II	1992	Brown Univ., USA
Handel-C	1996	Oxford Univ., Celoxica, UK
Galadriel & Nenya	1998	INESC-ID, Univ. of Algarve, Portugal
SPARCS	1998	Univ. of Cincinnati, USA
DEFACTO	1999	Univ. of South California/Information Sciences Institute, USA
SPC	1996	Univ. Karlsruhe, Germany, London Imperial College, UK
DeepC	1999	MIT, USA
Maruyama	2000	Univ. of Tsukuba, Japan
MATCH	1999	Northwestern Univ., USA
CAMERON	1998	Colorado State Univ., USA
NAPA-C	1997	Sarnoff Corporation, USA
Stream-C	2000	Los Alamos National Laboratory, Sarnoff Corporation, Adaptive Silicon, Inc., USA
Garpcc	1998	Univ. of California at Berkely, USA
CHIMAERA-C	2000	Northwestern University, USA
HP-Machine	2001	Hewlett-Packard Laboratories, USA
ROCCC	2003	University of California at Riverside, USA
DIL	1998	Carnegie Mellon Univ., USA
RaPiD-C	1997	Univ. of Washington, USA
CoDe-X	1995	Univ. of Kaiserlautern, Germany
XPP-VC	2002	PACT XPP Technologies AG., Munich, Germany

grouped according to the programming model used (sequential or concurrent) with distinctions on the abstraction level (high or low). Also represented in the table are the types of reconfigurable computing architectures they target (commercial FPGAs, a processor coupled with RPUs or coarse-grained RPUs) and the domain of specialization (architecture-specific or application-specific), which is important in some of the cases. Not surprisingly, most compilers use a high-level sequential programming model and target commercial FPGAs. This is explained pragmatically by the wide acceptance of imperative software programming languages when describing at high-levels of abstraction and by the widespread usage of commercial FPGAs.

More detailed characteristics of some of these compilers are illustrated in Tables V through XIII. A separation mark in each table splits the compilers targeting fine-grained from those targeting coarse-grained architectures. Each table shows, for each compiler, the most relevant information and compilation techniques (✓'s identify supporting and ×'s identify not supporting or not applicable). Table V shows the year of the first known publication and the place where the compiler was implemented; Table VI shows the input language accepted, the granularity of the description, and the programming model. Tables VII and VIII depict the front-end, the data types supported, the intermediate representations, and the level of parallelism exploited. Table IX shows the support of loops, array variables, the approach used when mapping the existent arrays onto the memories of the platform, the support of loop pipelining, the use of shift-registers and packing to reduce memory accesses. Table X exhibits the support of bit-width narrowing, bit optimizations, and sharing of FUs. Table XI identifies the support for temporal, spatial, and *reconfigware*/software partitioning.

Table XII shows the representation model for the output, the tool to generate the hardware structure, and the tool to generate the configuration data. Finally, in Table XIII we can see, for each compiler, the target platform whose results were published.

Although a theme that requires further research effort to be effective and competitive to manual design, a number of research compilers have been adopted by companies

Table VI. Main Characteristics of Some Compilers: General Information (II)

Compiler	Input Programming Language	Granularity of description	Model Used
Transmogrifier-C	C-subset	Operation	Software, imperative
PRISM-I, II	C-subset	Operation	Software, imperative
Handel-C	Concurrency + channels + memories (C-based)	Operation	Delay, CSP model, each assignment in one cycle
Galadriel & Nenya	Any language compiled to Java bytecodes (subset)	Operation	Software, imperative
SPARCS	VHDL tasks	Operation	VHDL and tasks
DEFACTO	C-subset	Operation	Software, imperative
SPC	C, Fortran: (subsets)	Operation	Software, imperative
DeepC	C, Fortran: (subsets)	Operation	Software, imperative
Maruyama	C-subset	Operation	Software, imperative
MATCH	MATLAB	Operation and/or functional blocks	Software, imperative
CAMERON	SA-C	Operation	Software, functional
NAPA-C	C-subset extended	Operation	Software, imperative added with concurrency
Stream-C	C-subset extended	Operation	Software, stream-based, processes
Garpcc	C	Operation	Software, imperative
CHIMAERA-C	C	Operation	Software, imperative
HP-Machine	C++ (subset) extended to specify Machines	Operation	Machines (process/thread) Notion of update per cycle
ROCCC	C-subset	Operation	Software, imperative
DIL	DIL	Operation	Delay notion, ?
RaPiD-C	RaPiD-C	Operation	Specific to RaPiD, par, wait, signal, and pipeline statements
CoDe-X	C-subset, ALE-X	Operation	Software, imperative
XPP-VC	C-subset (extended)	Operation	Software, imperative

as one of their main products. As an example, research techniques used in the MATCH compiler [Banerjee et al. 2000; Nayak et al. 2001a], were transferred to AccelChip. Other examples of academic research on compilers, the results of which have also been transferred to companies, include the research work on hardware compilation from Handel-C performed at the Oxford University [Page 1996] in the second half of the 1990's, ultimately leading to the creation of Celoxica. Research on the Streams-C hardware compiler [Gokhale et al. 2000b] was licensed to Impulse Accelerated Technologies, Inc. [Impulse-Accelerated-Technologies; Pellerin and Thibault 2005]; and the work on the *garpcc* compiler [Callahan et al. 2000] was used by Synopsys in the Nimble compiler [Li et al. 2000].

There has been ongoing research effort. One of the most relevant recent efforts is the Trident C-to-FPGA compiler [Tripp et al. 2005], which was especially developed for mapping scientific computing applications described in C to FPGAs. The compiler addresses floating-point computations and uses analysis techniques to expose high levels of ILP and to generate pipelined hardware circuits. It was developed with user-defined floating-point hardware units in mind as well.

Table VII. Main Characteristics of Some Compilers (cont.): Front-End Analysis (I)

Compiler	High-Level Mapping to Hardware	Front-End
Transmogrifier-C	Integrated	Custom
PRISM-I, II	Integrated	Lcc
Handel-C	Integrated	Custom
Galadriel & Nenya	Integrated	Custom: GALADRIEL
SPARCS	Integrated HLS	Custom
DEFACTO	Commercial HLS	SUIF
SPC	Integrated	SUIF
DeepC	Commercial RTL synthesis	SUIF
Maruyama	Integrated	Custom
MATCH	Commercial RTL synthesis	Custom
CAMERON	Commercial LS	Custom
NAPA-C	Integrated	SUIF
Stream-C	Integrated	SUIF
Garpcc	Integrated	SUIF
CHIMAERA-C	Integrated	GCC
HP-Machine	Integrated	Custom
ROCCC	Integrated	SUIF2
DIL	Integrated	Custom
RaPiD-C	Integrated	Custom
CoDe-X	Integrated	Custom
XPP-VC	Integrated	SUIF

Table VIII. Main Characteristics of Some Compilers (cont.): Front-End Analysis (II)

Compiler	Data Types	Intermediate Representations	Parallelism
Transmogrifier-C	Bit-level	AST	Operation
PRISM-I, II	Primitive	Operator Network (DFG)	Operation
Handel-C	Bit-Level	AST ?	Operation
Galadriel & Nenya	Primitive	HPDG + global DFG	Operation, inter basic block, inter-loop
SPARCS	Bit-level	USM	Operation, task-level
DEFACTO	Primitive	AST	Operation
SPC	Primitive	DFG	Operation
DeepC	Primitive	SSA	Operation
Maruyama	Primitive	DDGs (data dependence graphs)	Operation
MATCH	Primitive	AST, DFG for pipelining	Operation
CAMERON	Bit-Level	Hierarchical DDCF (Data Dependence and Control Flow) + DFG + AHA graph	Operation
NAPA-C	Pragmas (bit-level)	AST	Operation
Stream-C	Pragmas (bit-level)	AST	Operation
Garpcc	Primitive	Hyperblock + DFG	Operation, inter basic blocks
CHIMAERA-C	Primitive	DFG	Operation
HP-Machine	Bit-level	Hypergraph + DFG	Operation, Machines
ROCCC	Primitive	CIRRF	Operation
DIL	Bit-level (fixed-point)	Hierarchical and acyclic DFG	Operation
RaPiD-C	Primitive + pipe + ram	Control Trees	Functional, Operation
CoDe-X	Primitive	DAG	Operation, loops
XPP-VC	Primitive	HTG+, CDFG	Operation, inter basic block, inter-loop

Table IX. Main Characteristics of Some Compilers (cont.): Supported Features

Compiler	Loops	Array Variables	Floating-Point Operations	Pointers	Recursive Functions	Sharing of FUs
Transmogrifier-C	✓	×	×	×	×	×
PRISM-I, II	✓	×	×	×	×	×
Handel-C	✓	×	×	×	×	×
Galadriel & Nenya	✓	✓	×	×	×	(only on distinct conditional paths)
SPARCS	✓	✓	×	×	×	✓
DEFACTO	✓	✓	×	×	×	✓
SPC	✓ (inner)	✓	×	×	×	×
DeepC	✓	✓	✓	✓	×	✓
Maruyama	✓	✓	×	×	limited	×
MATCH	✓	✓	Converted to fixed-point	×	×	×
CAMERON	✓	✓	×	×	×	×
NAPA-C	✓	✓	×	×	×	✓
Stream-C	✓	✓	×	×	×	✓
garpcc	✓ (inner)	 ✓ ✓	 Software Software	 ✓ ✓	Software	 ×
CHIMAERA-C	×	×	Software	Software	software	×
HP-Machine	× (unrollable)	✓	×	×	×	✓
DIL	× (unrollable)	× (arrays are used to specify interconnections)	×	×	×	×
RaPiD-C	✓	✓ (used to access I/O data)	×	×	×	✓
CoDe-X	✓	✓	×	×	×	×
XPP-VC	✓	✓	×	×	×	×

7. FINAL REMARKS AND CONCLUSION

Reconfigurable computing platforms present numerous challenges to the average software programmer, as they expose a hardware-oriented computation model where programmers must also assume the role of hardware designers. To address this gap, researchers have developed compilation techniques and systems supporting the notion of *reconfigware* to automatically map computations described in high-level programming languages to reconfigurable architectures. Several of the approaches have been influenced by research in the area of hardware/software codesign and hardware synthesis, whereas others have attempted to bridge the gap directly using techniques from traditional compilation systems for which a dedicated hardware-oriented, and often architecture-specific, back-end was developed.

This survey presents some of the most prominent compilation techniques and efforts for reconfigurable platforms, whose main characteristics are identified in Table IV through Table XIII. Invariably, the applicability of some compilation techniques depends on the granularity of the target platform. As an example, bit-level operator specialization is clearly important for fine-grained architectures, but has limited usage for coarse-grained architectures. Other techniques, such as tree-height reduction are important in both kinds of architectures. Techniques developed for more traditional single processor and multiprocessor architectures, such as loop-based and data-oriented transformations, can be adapted when mapping computations to reconfigurable architectures. Furthermore, reconfigurable architectures have enabled newer

Table X. Main Characteristics of Some Compilers (cont.): Optimizations

Compiler	Bit-width Narrowing	Bit-Optimizations	Arrays-to-multiple-Memories Mapping	Loop Pipelining	Memory Accesses Reduction by Shift-Register	Memory Accesses Reduction by Packing
Transmogrifier-C	×	✓	×	×	×	×
PRISM-I, II	×	✓	×	×	×	×
Handel-C	×	×	Explicit use of memories	Manual	manual	manual
Galadriel & Nenya	✓	✓	(exhaustive or manual)	×	×	×
SPARCS	×	✓	✓	?	?	×
DEFACTO	×	×	×	✓	✓	✓
SPC	×	×	✓ (ILP)	✓	✓	×
DeepC	✓	×	✓	×	×	×
Maruyama	×	×	×	✓	×	×
MATCH	✓	✓	×	✓	×	✓
CAMERON	×	×	×	✓	✓	×
NAPA-C	×	×	✓ (implicit enumeration)	✓	×	×
Stream-C	×	×	✓ (implicit enumeration)	✓	×	×
Garpcc	×	×	×	✓	×	× (queues are used to grap a cache line at a time)
CHIMAERA-C	×	×	×	×	×	×
HP-Machine	✓	✓	×	✓	×	×
ROCCC	×	×	?	?	✓	?
DIL	✓	✓	×	✓	×	×
RaPiD-C	×	×	Explicit use of memories	✓	×	×
CoDe-X	×	×	×	✓	×	✓
XPP-VC	×	×	✓ (one array per internal memory)	✓	✓	×

techniques or variations of existing ones that are of limited or no applicability for traditional architectures.

Given the traditional lengthy and error-prone process of hardware synthesis, many systems have used pre-existing compilation frameworks, coupled with either commercial high-level synthesis tools or custom solutions that include module generators, to reduce the overall hardware/software solution development time. Furthermore, the recent use of placement schemes based on the list-scheduling algorithm shows short compilation times with acceptable results. A pressing concern when targeting reconfigurable platforms, in particular for fine-grained architectures, is the ability of the compiler to estimate the size and the clock rate and/or the delay of the final mapping.

Overally, the automatic mapping of computations to reconfigurable computing architectures represents a relatively new and very promising area of research, where researchers have built on a wealth of research on software compilation, parallelizing compilers, and system-level and architectural synthesis. This survey describes various up-to-date techniques on compilation to reconfigurable computing platforms. Many, but possibly not all, of the techniques described here are currently used in state-of-the-art compilers for reconfigurable computing platforms, thus revealing the diversity and also the maturity of the field of reconfigurable computing.

Table XI. Main Characteristics of Some Compilers (cont.): Forms of Partitioning

Compiler	Array Partitioning	Temporal Partitioning	Spatial Partitioning	RW/SW Partitioning
Transmogrifier-C	×	×	×	×
PRISM-I, II	×	×	×	×
Handel-C	×	×	×	×
Galadriel & Nenya	×	✓	×	×
SPARCS	×	✓	✓	×
DEFACTO	×	×	×	×
SPC	×	×	×	×
DeepC	✓	×	×	×
Maruyama	×	×	×	×
MATCH	×	×	✓	✓
CAMERON	×	×	×	×
NAPA-C	×	×	×	× (annotations)
Stream-C	×	×	×	×
garpcc	×	×	×	✓
CHIMAERA-C	×	×	×	✓
HP-Machine	×	×	×	×
ROCCC	×	×	×	×
DIL	×	× (the architecture is automatically virtualized)	×	×
RaPiD-C	×	×	×	×
CoDe-X	×	×	×	✓
XPP-VC	×	✓ (includes loop dissevering)	×	×

Table XII. Main Characteristics of Some Compilers (cont.): Back-End Support

Compiler	Output of the compiler: (1): RTL-HDL (2): Algorithmic HDL (3): bitstreams	Generation of the Hardware Structure: VLS: Vendor Logic Synthesis CG: Circuit Generators OO: one-to-one Mapping	Back-end (bitstream generation): CPR: Commercial Placement and Routing Tools
Transmogrifier-C	?	CG	CPR
PRISM-I, II	?	CG	CPR
Handel-C	(1)	CG	CPR
Galadriel & Nenya	(1)	Data-path: CG Control unit: VLS	CPR
SPARCS	(1)	VLS	CPR
DEFACTO	(2)	VLS	CPR
SPC	(1)	CG	CPR
DeepC	(1)	VLS	CPR
Maruyama	(1)	VLS	CPR
MATCH	(2)	VLS	CPR
CAMERON	(1)	VLS	CPR
NAPA-C	(1)	CG	CPR
Stream-C	(1)	CG	CPR
garpcc	(3)	CG	Proprietary: Gama
CHIMAERA-C	?	Manually ?	—
HP-MAchine	(3)	CG	Proprietary + Jbits
ROCCC	Data-path: (1) Control unit: (2)	VLS	CPR
DIL	(3)	CG	Proprietary
RaPiD-C	(3)	OO	Proprietary
CoDe-X	(3)	OO	Proprietary
XPP-VC	(3)	OO	Proprietary: xmap

Table XIII. Main Characteristics of Some Compilers (cont.): Target Platform

Compiler	Target Platform
Transmogrifier-C	1 FPGA
PRISM-I, II	1 FPGA
Handel-C	1 FPGA
Galadriel & Nenya	1 FPGA, multiple memories
SPARCS	multiple FPGAs, multiple memories
DEFACTO	multiple FPGAs, multiple memories
SPC	1 FPGA, multiple memories?
DeepC	A mesh of tiles each one with 1 FPGA connected to 1 memory
Maruyama	1 FPGA, multiple memories
MATCH	multiple FPGAs, each one connected to one memory
CAMERON	1 FPGA, multiple memories
NAPA-C	1 FPGA, multiple memories
Stream-C	1 FPGA, multiple memories
garpcc	μP (Garp) connected to a proprietary RPU
CHIMAERA-C	μP (SimpleScalar) connected to an RFU
HP-MAchine	1 FPGA, multiple memories
ROCCC	1 FPGA
DIL	PipeRench RPU
RaPiD-C	RaPiD RPU
CoDe-X	Multi-KressArrays connected to a host system
XPP-VC	1 XPP, multiple memories

REFERENCES

AAMODT, T. AND CHOW, P. 2000. Embedded ISA support for enhanced floating-point to fixed-point ANSI-C compilation. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*. ACM, New York, 128–137.

ACCELCHIP. http://www.accelchip.com/.

AGARWAL, L., WAZLOWSKI, M., AND GHOSH, S. 1994. An asynchronous approach to efficient execution of programs on adaptive architectures utilizing FPGAs. In *Proceedings of the 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*. IEEE, Los Alamitos, CA, 101–110.

ALLEN, J. R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*. ACM, New York, 177–189.

ALTERA INC. http://www.altera.com/.

ALTERA INC. 2002. Stratix programmable logic device family data sheet 1.0, H.W.A.C. Altera Corp.

AMERSON, R., CARTER, R. J., CULBERTSON, W. B., KUEKES, P., AND SNIDER, G. 1995. Teramac-configurable custom computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*. IEEE, Los Alamitos, CA, 32–38.

ANNAPOLIS MICROSYSTEMS INC. 1999. WildStarTM reconfigurable computing engines, User's manual R3.3.

ATHANAS, P. 1992. An *Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration*. Brown University.

ATHANAS, P. AND SILVERMAN, H. 1993. Processor reconfiguration through instruction-set metamorphosis: Architecture and compiler. *Computer 26*, 3, 11–18.

AUGUST, D. I., SIAS, J. W., PUIATTI, J.-M., MAHLKE, S. A., CONNORS, D. A., CROZIER, K. M., AND HWU, W.-M. W. 1999. The program decision logic approach to predicated execution. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE, Los Alamitos, CA, 208–219.

BABB, J. 2000. High-level compilation for reconfigurable architectures, Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA.

BABB, J., RINARD, M., MORITZ, C. A., LEE, W., FRANK, M., BARUA, R., AND AMARASINGHE, S. 1999. Parallelizing applications into silicon. In *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 70–81.

BANERJEE, P., SHENOY, N., CHOUDHARY, A., HAUCK, S., BACHMANN, C., HALDAR, M., JOISHA, P., JONES, A., KANHARE, A., NAYAK, A., PERIYACHERI, S., WALKDEN, M., AND ZARETSKY, D. 2000. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. IEEE, Los Alamitos, CA, 39–48.

BARADARAN, N. AND DINIZ, P. 2006. Memory parallelism using custom array mapping to heterogeneous storage structures. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'06)*. IEEE, Los Alamitos, CA, 383–388.

BARADARAN, N., PARK, J., AND DINIZ, P. 2004. Compiler reuse analysis for the mapping of data in FPGAs with RAM blocks. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'04)*. IEEE, Los Alamitos, CA, 45–152.

BARUA, R., LEE, W., AMARASINGHE, S., AND AGARWAL, A. 2001. Compiler support for scalable and efficient memory systems. *IEEE Trans. Computers 50*, 11, 1234–1247.

BAUMGARTE, V., EHLERS, G., MAY, F., CKEL, A, N., VORBACH, M., AND WEINHARDT, M. 2003. PACT XPP: A self-reconfigurable data processing architecture. *J. Supercomput. 26*, 2, 167–184.

BECK, G., YEN, D., AND ANDERSON, T. 1993. The Cydra 5 minisupercomputer: Architecture and implementation. *J. Supercomput. 7*, 1–2, 143–180.

BECKER, J., HARTENSTEIN, R., HERZ, M., AND NAGELDINGER, U. 1998. Parallelization in co-compilation for configurable accelerators. In *Proceedings of the Asia South Pacific Design Automation Conference (ASP-DAC'98)*, 23–33.

BELLOWS, P. AND HUTCHINGS, B. 1998. JHDL-An HDL for reconfigurable systems. In *Proceedings of the IEEE 6th Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 175–184.

BERNSTEIN, R. 1986. Multiplication by Integer constants. *Softw.Pract. Exper. 16*, 7, 641–652.

BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. 1998. Lava: Hardware design in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM, New York, 174–184.

BÖHM, W., HAMMES, J., DRAPER, B., CHAWATHE, M., ROSS, C., RINKER, R., AND NAJJAR, W. 2002. Mapping a single assignment programming language to reconfigurable systems. *J. Supercomput. 21*, 2, 117–130.

BÖHM, A. P. W., DRAPER, B., NAJJAR, W., HAMMES, J., RINKER, R., CHAWATHE, M., AND ROSS, C. 2001. One-step compilation of image processing algorithms to FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA, 209–218.

BONDALAPATI, K. 2001. Parallelizing of DSP nested loops on reconfigurable architectures using data context switching. In *Proceedings of the IEEE/ACM 38th Design Automation Conference (DAC'01)*. ACM, New York, 273–276.

BONDALAPATI, K., DINIZ, P., DUNCAN, P., GRANACKI, J., HALL, M., JAIN, R., AND ZIEGLER, H. 1999. DEFACTO: A design environment for adaptive computing technology. In *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99)*. Lecture Notes in Computer Science, vol. 1586, Springer, Berlin, 570–578.

BONDALAPATI, K. AND PRASANNA, V. K. 1999. Dynamic precision management for loop computations on reconfigurable architectures. In *Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 249–258.

BRASEN, D. R. AND SAUCIER, G. 1998. Using cone structures for circuit partitioning into FPGA packages. *IEEE Trans. Comput.-Aid. Des. Integrt. Circuits Syst. 17*, 7, 592–600.

BROOKS, D. AND MARTONOSI, M. 1999. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA'99)*. IEEE, Los Alamitos, CA, 13–22.

BUDIU, M., GOLDSTEIN, S., SAKR, M., AND WALKER, K. 2000. BitValue inference: Detecting and exploiting narrow bit-width computations. In *Proceedings of the 6th International European Conference on Parallel Computing (EuroPar'00)*. Lecture Notes in Computer Science, vol. 1900, Springer, Berlin, 969–979.

BUDIU, M. AND GOLDSTEIN, S. C. 1999. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA'99)*. ACM, New York, 195–205.

CADAMBI, S. AND GOLDSTEIN, S. 2000. Efficient place and route for pipeline reconfigurable architectures. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD'00)*. IEEE, Los Alamitos, CA, 423–429.

CALLAHAN, T. J. 2002. Automatic compilation of C for hybrid reconfigurable architectures. Ph.D. thesis, University of California, Berkeley.

CALLAHAN, T. J., HAUSER, J. R., AND WAWRZYNEK, J. 2000. The Garp architecture and C compiler. *Computer 33*, 4, 62–69.

CALLAHAN, T. J. AND WAWRZYNEK, J. 2000. Adapting software pipelining for reconfigurable computing. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*. ACM, New York, 57–64.

CALLAHAN, T. J. AND WAWRZYNEK, J. 1998. Instruction-level parallelism for reconfigurable computing. In *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL'98)*. Lecture Notes in Computer Science, vol. 1482, Springer, Berlin, 248–257.

CALLAHAN, T. J., CHONG, P., DEHON, A., AND WAWRZYNEK, J. 1998. Fast module mapping and placement for data-paths in FPGAs. In *Proceedings of the ACM 6th International Symposium on Field Programmable Gate Arrays (FPGA'98)*. ACM, New York, 123–132.

CARDOSO, J. M. P. 2003. On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures. *IEEE Trans. Comput. 52*, 10, 1362–1375.

CARDOSO, J. M. P. AND NETO, H. C. 2003. Compilation for FPGA-based reconfigurable hardware. *IEEE Des. Test Comput. Mag. 20*, 2, 65–75.

CARDOSO, J. M. P. AND WEINHARDT, M. 2002. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL'02)*. Lecture Notes in Computer Science, Springer, Berlin, 864–874.

CARDOSO, J. M. P. AND NETO, H. C. 2001. Compilation increasing the scheduling scope for multi-memory-FPGA-based custom computing machines. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*. Lecture Notes in Computer Science, vol. 2147, Springer, Berlin, 523–533.

CARDOSO, J. M. P. AND NETO, H. C. 2000. An enhanced static-list scheduling algorithm for temporal partitioning onto RPUs. In *Proceedings of the IFIP TC10/WG10.5 10th International Conference on Very Large Scale Integration (VLSI'99)*.

CARDOSO, J. M. P. AND NETO, H. C. 1999. Macro-based hardware compilation of Java bytecodes into a dynamic reconfigurable computing system. In *Proceedings of the IEEE 7th Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 2–11.

CASPI, E. 2000. Empirical study of opportunities for bit-level specialization in word-based programs, Tech. rep., University of California Berkeley.

CASPI, E., CHU, M., RANDY, H., YEH, J., WAWRZYNEK, J., AND DEHON, A. 2000. Stream computations organized for reconfigurable execution (SCORE). In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*. Lecture Notes in Computer Science, vol. 1896, Springer, Berlin, 605–614.

CELOXICA LTD. http://www.celoxica.com/.

COMPTON, K. AND HAUCK, S. 2002. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv. 34*, 2, 171–210.

CRONQUIST, D. C., FRANKLIN, P., BERG, S. G., AND EBELING, C. 1998. Specifying and compiling applications for RaPiD. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 116–125.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

DEHON, A., MARKOVSKYA, Y., CASPIA, E., CHUA, M., HUANGA, R., PERISSAKISA, S., POZZI, L., YEHA, J., AND WAWRZYNEKA, J. 2006. Stream computations organized for reconfigurable execution. *Microprocess. Microsyst. 30*, 6, 334–354.

DEHON, A. 2000. The density advantage of configurable computing. *Computer* 33, 4, 41–49.

DEHON, A. 1996. Reconfigurable architectures for general-purpose computing. Tech. rep. MIT, Cambridge, MA.

DINIZ, P. C. 2005. Evaluation of code generation strategies for scalar replaced codes in fine-grain configurable architectures. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. IEEE, Los Alamitos, CA, 73–82.

DINIZ, P. C., HALL, M. W., PARK, J., SO, B., AND ZIEGLER, H. E. 2001. Bridging the gap between compilation and synthesis in the DEFACTO system. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*. Lecture Notes in Computer Science, vol. 2624, Springer, Berlin, 2003, 52–70.

DONCEV, G., LEESER, M., AND TARAFDAR, S. 1998. High level synthesis for designing custom computing hardware. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 326–327.

DUNCAN, A. A., HENDRY, D. C., AND CRAY, P. 2001. The COBRA-ABS high level synthesis system for multi-FPGA custom computing machines. *IEEE Trans. VLSI Syst. 9*, 1, 218–223.

DUNCAN, A. A., HENDRY, D.C., AND CRAY, P. 1998. An overview of the COBRA-ABS high level synthesis system for multi-FPGA systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 106–115.

EBELING, C., CRONQUIST, D. C., AND FRANKLIN, P. 1995. RaPiD—Reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'95)*. Lecture Notes in Computer Science, vol. 975, Springer, Berlin, 126–135.

EDWARDS, S. 2002. High-level synthesis from the synchronous language Esterel. In *Proceedings of the 11th IEEE/ACM International Workshop on Logic and Synthesis (IWLS'02)*. 401–406.

FEKETE, S., KÖHLER, E., AND TEICH, J. 2001. Optimal FPGA module placement with temporal precedence constraints. In *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference and Exhibition (DATE'01)*. 658–665.

XILINX, INC. Forge. Forge compiler. http://www.lavalogic.com/.

FRIGO, J., GOKHALE, M., AND LAVENIER, D. 2001. Evaluation of the Streams-C C-to-FPGA compiler: An applications perspective. In *Proceedings of the ACM 9th International Symposium on Field-Programmable Gate Arrays (FPGA'01)*. ACM, New York, 134–140.

FUJII, T., FURUTA, K., MOTOMURA, M., NOMURA, M., MIZUNO, M., ANJO, K., WAKABAYASHI, K., HIROTA, Y., NAKAZAWA, Y., ITO, H., AND YAMASHINA, M. 1999. A dynamically reconfigurable logic engine with a multi-context/multi-mode unified-cell architecture. In *Proceedings of the IEEE International Solid State Circuits Conference (ISSCC'99)*. IEEE, Los Alamitos, CA, 364–365.

GAJSKI, D. D., DUTT, N. D., WU, A. C. H., AND LIN, S. Y. L. 1992. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer, Amsterdam.

GALLOWAY, D. 1995. The transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings of the 3rd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'95)*. IEEE, Los Alamitos, CA, 136–144.

GANESAN, S. AND VEMURI, R. 2000. An integrated temporal partitioning and partial reconfiguration technique for design latency improvement. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'00)*. IEEE, Los Alamitos, CA, 320–325.

GIRKAR, M. AND POLYCHRONOPOULOS, C. D. 1992. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parall. Distrib. Syst. 3*, 2, 166–178.

GOKHALE, M. AND GRAHAM, P. S. 2005. *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays.* Springer, Berlin.

GOKHALE, M., STONE, J. M., AND GOMERSALL, E. 2000a. Co-synthesis to a hybrid RISC/FPGA architecture. *J. VLSI Signal Process. Syst. Signal, Image Video Technol. 24*, 2, 165–180.

GOKHALE, M., STONE, J. M., ARNOLD, J., AND KALINOWSKI, M. 2000b. Stream-oriented FPGA computing in the Streams-C high level language. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. IEEE, Los Alamitos, CA, 49–56.

GOKHALE, M. AND STONE, J. 1999. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In *Proceedings of the 7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 63–69.

GOKHALE, M. AND STONE, J. M. 1998. NAPA C: Compiling for a hybrid RISC/FPGA architecture. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE, Los Alamitos, CA, 126–135.

GOKHALE, M. AND GOMERSALL, E. 1997. High-level compilation for fine grain FPGAs. In *Proceedings of the IEEE 5th Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*. IEEE, Los Alamitos, CA, 165–173.

GOKHALE, M. AND MARKS, A. 1995. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic. In *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications (FPL'95)*. Lecture Notes in Computer Science, vol. 975, Springer, Berlin, 399–408.

GOKHALE, M. AND CARLSON, W. 1992. An introduction to compilation issues for parallel machines. *J. Supercomput.* 283–314.

GOKHALE, M., HOLMES, W., KOPSER, A., KUNZE, D., LOPRESTI, D. P., LUCAS, S., MINNICH, R., AND OLSEN, P. 1990. SPLASH: A reconfigurable linear logic array. In *Proceedings of the International Conference on Parallel Processing (ICPP'90)*. 526–532.

GOLDSTEIN, S. C., SCHMIT, H., BUDIU, M., CADAMBI, S., MOE, M., AND TAYLOR, R. R. 2000. PipeRench: A reconfigurable architecture and compiler. *Computer 33*, 4, 70–77.

GOLDSTEIN, S. C., SCHMIT, H., MOE, M., BUDIU, M., CADAMBI, S., TAYLOR, R. R., AND LAUFER, R. 1999. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*. IEEE, Los Alamitos, CA, 28–39.

GONZALEZ, R. E. 2000. Xtensa: A configurable and extensible processor. *IEEE Micro 20*, 2, 60–70.

GUCCIONE, S., LEVI, D., AND SUNDARARAJAN, P. 2000. Jbits: Java based interface for reconfigurable computing. In *Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD'00)*. 1–9.

GUO, Z. AND NAJJAR, W. 2006. A compiler intermediate representation for reconfigurable fabrics. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL'2006)*. IEEE, Los Alamitos, CA, 741–744.

GUO, Z., BUYUKKURT, A. B., AND NAJJAR, W. 2004. Input data reuse in compiling Window operations onto reconfigurable hardware. In *Proceedings of the ACM Symposium on Languages, Compilers and Tools for Embedded Systems (LCTES'04). ACM SIGPLAN Not. 39*, 7, 249–256.

GUPTA, S., SAVOIU, N., KIM, S., DUTT, N., GUPTA, R., AND NICOLAU, A. 2001. Speculation techniques for high-level synthesis of control intensive designs. In *Proceedings of the 38th IEEE/ACM Design Automation Conference (DAC'01)*. ACM, New York, 269–272.

HALDAR, M., NAYAK, A., CHOUDHARY, A., AND BANERJEE, P. 2001a. A system for synthesizing optimized FPGA hardware from MATLAB. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*. IEEE, Los Alamitos, CA, 314–319.

HALDAR, M., NAYAK, A., SHENOY, N., CHOUDHARY, A., AND BANERJEE, P. 2001b. FPGA hardware synthesis from MATLAB. In *Proceedings of the 14th International Conference on VLSI Design (VLSID'01)*. 299–304.

HARTENSTEIN, R. W. 2001. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. IEEE, Los Alamitos, CA, 642–649.

HARTENSTEIN, R. W. 1997. The microprocessor is no more general purpose: Why future reconfigurable platforms will win. In *Proceedings of the International Conference on Innovative Systems in Silicon (ISIS'97)*.

HARTENSTEIN, R. W., BECKER, J., KRESS, R., AND REINIG, H. 1996. High-performance computing using a reconfigurable accelerator. *Concurrency—Pract. Exper. 8*, 6, 429–443.

HARTENSTEIN, R. W. AND KRESS, R. 1995. A datapath synthesis system for the reconfigurable datapath architecture. In *Procedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'95)*. ACM, New York, 479–484.

HARTLEY, R. 1991. Optimization of canonic signed digit multipliers for filter design. In *Proceedings of the IEEE International Sympoisum on Circuits and Systems (ISCA'91)*. IEEE, Los Alamitos, CA, 343–348.

HAUCK, S., FRY, T. W., HOSLER, M. M., AND KAO, J. P. 2004. The Chimaera reconfigurable functional unit. *IEEE Trans. VLSI Syst. 12*, 2, 206–217.

HAUCK, S. 1998. The roles of FPGAs in reprogrammable systems. *Proc. IEEE 86*, 4, 615–638.

HAUSER, J. R. AND WAWRZYNEK, J. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97)*. IEEE, Los Alamitos, CA, 12–21.

HOARE, C. A. R. 1978. Communicating sequential processes. *Comm. ACM* 21, 8, 666–677.

IMPACT. The Impact Research Group. http://www.crhc.uiuc.edu/.

IMPULSE-ACCELERATED-TECHNOLOGIES INC. http://www.impulsec.com/.

INOUE, A., TOMIYAMA, H., OKUMA, H., KANBARA, H., AND YASUURA, H. 1998. Language and compiler for optimizing datapath widths of embedded systems. *IEICE Trans. Fundamentals E81-A*, 12, 2595–2604.

ISELI, C. AND SANCHEZ, E. 1993. Spyder: A reconfigurable VLIW processor using FPGAs. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'93)*. IEEE, Los Alamitos, CA, 17–24.

JONES, M., SCHARF, L., SCOTT, J., TWADDLE, C., YACONIS, M., YAO, K., ATHANAS, P., AND SCHOTT, B. 1999. Implementing an API for distributed adaptive computing systems. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 222–230.

JONG, G. D., VERDONCK, B. L. C., WUYTACK, S., AND CATTHOOR, F. 1995. Background memory management for dynamic data structure intensive processing systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*. IEEE, Los Alamitos, CA, 515–520.

KASTRUP, B., BINK, A., AND HOOGERBRUGGE, J. 1999. ConCISe: A compiler-driven CPLD-based instruction set accelerator. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 92–101.

KAUL, M. AND VEMURI, R. 1999. Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'99)*. 202–209.

KAUL, M., VEMURI, R., GOVINDARAJAN, S., AND OUAISS, I. 1999. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'99)*. ACM, New York, 616–622.

KAUL, M. AND VEMURI, R. 1998. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'98)*. IEEE, Los Alamitos, CA, 389–396.

KHOURI, K. S., LAKSHMINARAYANA, G., AND JHA, N. K. 1999. Memory binding for performance optimization of control-flow intensive behaviors. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'99)*. IEEE, Los Alamitos, CA, 482–488.

KOBAYASHI, S., KOZUKA, I., TANG, W. H., AND LANDMANN, D. 2004. A software/hardware codesigned hands-free system on a "resizable" block-floating-point DSP. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'04)*. IEEE, Los Alamitos, CA, 149–152.

KRESS, R. 1996. A fast reconfigurable ALU for Xputers. Tech. rep. Kaiserlautern University, Kaiserlautern.

KRUPNOVA, H. AND SAUCIER, G. 1999. Hierarchical interactive approach to partition large designs into FPGAs. In *Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications (FPL'99)*. Lecture Notes in Computer Science, vol. 1673, Springer, Berlin, 101–110.

KUM, K.-I., KANG, J., AND SUNG, W. 2000. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Circuits Syst. II*, 47, 9, 840–848.

LAKSHMIKANTHAN, P., GOVINDARAJAN, S., SRINIVASAN, V., AND VEMURI, R. 2000. Behavioral partitioning with synthesis for multi-FPGA architectures under interconnect, area, and latency constraints. In *Proceedings of the 7th Reconfigurable Architectures Workshop (RAW'00)*. Lecture Notes in Computer Science, vol. 1800, Springer, Berlin, 924–931.

LAKSHMINARAYANA, G., KHOURI, K. S., AND JHA, N. K. 1997. Wavesched: A novel scheduling technique for control-flow intensive designs. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*. IEEE, Los Alamitos, CA, 244–250.

LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. 1998. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the ACM 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*. ACM, New York, 46–57.

LEISERSON, C. E. AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 1, 5–35.

LEONG, M. P., YEUNG, M. Y., YEUNG, C. K., FU, C. W., HENG, P. A., AND LEONG, P. H. W. 1999. Automatic floating to fixed point translation and its application to post-rendering 3D warping. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'99)*. IEEE, Los Alamitos, CA, 240–248.

LEWIS, D. M., IERSSEL, M. V., ROSE, J., AND CHOW, P. 1998. The Transmogrifier-2: A 1 million gate rapid-prototyping system. *IEEE Trans. VLSI Syst. 6*, 2, 188–198.

LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., AND STOCKWOOD, J. 2000. Hardware-software co-design of embedded reconfigurable architectures. In *Proceedings of the IEEE/ACM Design Automation Conference (DAC'00)*. 507–512.

LIU, H. AND WONG, D. F. 1999. Circuit partitioning for dynamically reconfigurable FPGAs. In *Proceedings of the ACM 7th International Symposium on Field-Programmable Gate Arrays (FPGA'99)*. ACM, New York, 187–194.

LUK, W. AND WU, T. 1994. Towards a declarative framework for hardware-software codesign. In *Proceedings of the 3rd International Workshop on Hardware/Software Codesign (CODES'94)*. IEEE, Los Alamitos, CA, 181–188.

LYSAGHT, P. AND ROSENSTIEL, W. 2005. *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, Berlin.

MAGENHEIMER, D. J., PETERS, L., PETTIS, K. W., AND ZURAS, D. 1988. Integer multiplication and division on the HP precision architecture. *IEEE Trans. Computers 37*, 8, 980–990.

MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. 1992. Effective compiler support for predicated execution using the hyperblock. *ACM SIGMICRO Newsl.* 23, 1–2, 45–54.

MARKOVSKIY, Y., CASPI, E., HUANG, R., YEH, J., CHU, M., WAWRZYNEK, J., AND DEHON, A. 2002. Analysis of quasistatic scheduling techniques in a virtualized reconfigurable machine. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA'02)*. ACM, New York, 196–205.

MARUYAMA, T. AND HOSHINO, T. 2000. A C to HDL compiler for pipeline processing on FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00)*. IEEE, Los Alamitos, CA, 101–110.

MATHSTAR INC. http://www.mathstar.com.

MATHWORKS. Home page: http://www.mathworks.com/.

MEI, B., LAMBRECHTS, A., VERKEST, D., MIGNOLET, J.-Y., AND LAUWEREINS, R. 2005. Architecture exploration for a reconfigurable architecture template. *IEEE Des. Test Comput. Mag. 22*, 2, 90–101.

MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2002. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT'02)*. IEEE, Los Alamitos, CA, 166–173.

MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2003. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the International Conference on Field Programmable Logic and Application (FPL'03)*. Lecture Notes in Computer Science, vol. 2778, Springer, Berlin, 61–70.

MENCER, O., PLATZNER, M., MORF, M., AND FLYNN, M. J. 2001. Object-oriented domain-specific compilers for programming FPGAs. *IEEE Trans. VLSI Syst.* 9, 1, 205–210.

MICHELI, G. D. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, New York.

MICHELI, G. D. AND GUPTA, R. 1997. Hardware/software co-design. *Proc. IEEE 85*, 3, 349–365.

MIRSKY, E. AND DEHON, A. 1996. MATRIX: A reconfigurable computing device with reconfigurable instruction deployable resources. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96)*. IEEE, Los Alamitos, CA, 157–166.

MITRIONICS A. B. http://www.mitrionics.com/.

MIYAMORI, T. AND OLUKOTUN, K. 1998. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 2–11.

MOLL, L., VUILLEMIN, J., AND BOUCARD, P. 1995. High-energy physics on DECPeRLe-1 programmable active memory. In *Proceedings of the ACM 3rd International Symposium on Field-Programmable Gate Arrays (FPGA'95)*. ACM, New York, 47–52.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

NALLATECH INC. http://www.nallatech.com.

NAYAK, A., HALDAR, M., CHOUDHARY, A., AND BANERJEE, P. 2001a. Parallelization of Matlab applications for a multi-FPGA system. In *Proceedings of the IEEE 9th Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA, 1–9.

NAYAK, A., HALDAR, M., CHOUDHARY, A., AND BANERJEE, P. 2001b. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Proceedings of the Design, Automation and Test Conference in Europe (DATE'01)*. IEEE, Los Alamitos, CA, 722–728.

NISBET, S. AND GUCCIONE, S. 1997. The XC6200DS development system. In *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications (FPL'97)*. Lecture Notes in Computer Science, vol. 1304, Springer, Berlin, 61–68.

OGAWA, O., TAKAGI, K., ITOH, Y., KIMURA, S., AND WATANABE, K. 1999. Hardware synthesis from C programs with estimation of bit- length of variables. *IEICE Trans. Fundamentals E82-A*, 11, 2338–2346.

ONG, S.-W., KERKIZ, N., SRIJANTO, B., TAN, C., LANGSTON, M., NEWPORT, D., AND BOULDIN, D. 2001. Automatic mapping of multiple applications to multiple adaptive computing systems. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, Los Alamitos, CA, 10–20.

OUAISS, I. AND VEMURI, R. 2001. Hierarchical memory mapping during synthesis in FPGA-based reconfigurable computers. In *Proceedings of the Design, Automation and Test in Europe (DATE'01)*. IEEE, Los Alamitos, CA, 650–657.

OUAISS, I. AND VEMURI, R. 2000. Efficient resource arbitration in reconfigurable computing environments. In *Proceedings of the Design, Automation and Test in Europe (DATE'00)*. 560–566.

OUAISS, I., GOVINDARAJAN, S., SRINIVASAN, V., KAUL, M., AND VEMURI, R. 1998a. An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures. In *Proceedings of the 5th Reconfigurable Architectures Workshop (RAW'98)*. Lecture Notes in Computer Science, vol. 1388, Springer, Berlin, 31–36.

OUAISS, I., GOVINDARAJAN, S., SRINIVASAN, V., KAUL, M., AND VEMURI, R. 1998b. A unified specification model of concurrency and coordination for synthesis from VHDL. In *Proceedings of the International Conference on Information Systems Analysis and Synthesis (ISAS'98)*. 771–778.

PAGE, I. 1996. Constructing hardware-software systems from a single description. *J. VLSI Signal Process.* 87–107.

PAGE, I. AND LUK, W. 1991. Compiling Occam into FPGAs. In *FPGAs*, Abingdon EE&CS Books, Abingdon, UK, 271–283.

PANDEY, A. AND VEMURI, R. 1999. Combined temporal partitioning and scheduling for reconfigurable architectures. In *Proceedings of the SPIE Photonics East Conference*. 93–103.

PARK, J. AND DINIZ, P. 2001. Synthesis of memory access controller for streamed data applications for FPGA-based computing engines. In *Proceedings of the 14th International Symposium on System Synthesis (ISSS'01)*. 221–226.

PELLERIN, D. AND THIBAULT, S. 2005. *Practical FPGA Programming in C.* Prentice Hall, Englewood Cliffs, NJ.

PETERSON, J. B., O'CONNOR, R. B., AND ATHANAS, P. 1996. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures. In *Proceedings of the 4th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'96)*. IEEE, Los Alamitos, CA, 178–179.

PURNA, K. M. G. AND BHATIA, D. 1999. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Trans. Computers 48*, 6, 579–590.

RADETZKI, M. 2000. Synthesis of digital circuits from object-oriented specifications. Tech rep. Oldenburg University, Oldenburg, Germany.

RAIMBAULT, F., LAVENIER, D., RUBINI, S., AND POTTIER, B. 1993. Fine grain parallelism on an MIMD machine using FPGAs. In *Proceedings of the IEEE Workshop FPGAs for Custom Computing Machines (FCCM'93)*. IEEE, Los Alamitos, CA, 2–8.

RAJAN, J. V. AND THOMAS, D. E. 1985. Synthesis by delayed binding of decisions. In *Proceedings of the 22nd IEEE Design Automation Conference (DAC'85)*. IEEE, Los Alamitos, CA, 367–373.

RALEV, K. R. AND BAUER, P. H. 1999. Realization of block floating point digital filters and application to block implementations. *IEEE Trans. Signal Process. 47*, 4, 1076–1086.

RAMACHANDRAN, L., GAJSKI, D., AND CHAIYAKUL, V. 1994. An algorithm for array variable clustering. In *Proceedings of the European Design Test Conference (EDAC'94)*, IEEE, Los Alamitos, CA, 262–266.

RAMANUJAM, J. AND SADAYAPPAN, P. 1991. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. Parall. Distrib. Syst. 2*, 4, 472–482.

RAU, B. R. 1994. Iterative module scheduling: An algorithm for software pipelining loops. In *Proceedings of the ACM 27th Annual International Symposium on Microarchitecture (MICRO-27)*. ACM, New York, 63–74.

RAZDAN, R. 1994. PRISC: Programmable reduced instruction set computers. Tech. rep. Division of Applied Sciences, Harvard University, Cambridge, MA.

RAZDAN, R. AND SMITH, M. D. 1994. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th IEEE/ACM Annual International Symposium on Microarchitecture (MICRO-27)*. 172–180.

RINKER, R., CARTER, M., PATEL, A., CHAWATHE, M., ROSS, C., HAMMES, J., NAJJAR, W., AND BÖHM, A. P. W. 2001. An automated process for compiling dataflow graphs into hardware. *IEEE Trans. VLSI Syst. 9*, 1, 130–139.

RIVERA, G. AND TSENG, C.-W. 1998. Data transformations for eliminating cache misses. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, 38–49.

RUPP, C. R., LANDGUTH, M., GARVERICK, T., GOMERSALL, E., HOLT, H., ARNOLD, J. M., AND GOKHALE, M. 1998. The NAPA adaptive processing architecture. In *Proceedings of the IEEE 6th Symposium on Field-Programmable Custom Computing Machines (FCCM'98)*. IEEE, Los Alamitos, CA, 28–37.

SALEFSKI, B. AND CAGLAR, L. 2001. Re-configurable computing in wireless. In *Proceedings of the 38th Annual ACM IEEE Design Automation Conference (DAC'01)*. 178–183.

SANTOS, L. C. V. D., HEIJLIGERS, M. J. M., EIJK, C. A. J. V., EIJNHOVEN, J. V., AND JESS, J. A. G. 2000. A code-motion pruning technique for global scheduling. *ACM Trans. Des. Autom. Electron. Syst. 5*, 1, 1–38.

SCHMIT, H. AND THOMAS, D. 1998. Address generation for memories containing multiple arrays. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits Syst. 17*, 5, 377–385.

SCHMIT, H. AND THOMAS, D. 1997. Synthesis of applications-specific memory designs. *IEEE Trans. VLSI Syst. 5*, 1, 101–111.

SCHMIT, H., ARNSTEIN, L., THOMAS, D., AND LAGNESE, E. 1994. Behavioral synthesis for FPGA-based computing. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*. IEEE, Los Alamitos, CA, 125–132.

SÉMÉRIA, L., SATO, K., AND MICHELI, G. D. 2001. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Trans. VLSI Syst. 9*, 6, 743–756.

SHARP, R. AND MYCROFT, A. 2001. A higher level language for hardware synthesis. In *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and*

Verification Methods (CHARME'01). Lecture Notes in Computer Science, vol. 2144, Springer, Berlin, 228–243.

SHIRAZI, N., WALTERS, A., AND ATHANAS, P. 1995. Quantitative analysis of floating point arithmetic on FPGA-based custom computing machines. In *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines (FCCM)*. IEEE, Los Alamitos, CA, 155–162.

SINGH, H., LEE, M.-H., LU, G., BAGHERZADEH, N., KURDAHI, F. J., AND FILHO, E. M. C. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers 49*, 5, 465–481.

SNIDER, G. 2002. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proceedings of the ACM 10th International Symposium on Field-Programmable Gate Arrays (FPGA'02)*. ACM, New York, 177–186.

SNIDER, G., SHACKLEFORD, B., AND CARTER, R. J. 2001. Attacking the semantic gap between application programming languages and configurable hardware. In *Proceedings of the ACM 9th International Symposium on Field-Programmable Gate Arrays (FPGA'01)*. ACM, New York, 115–124.

SO, B., HALL, M., AND ZIEGLER, H. 2004. Custom data layout for memory parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, Los Alamitos, CA, 291–302.

SO, B. AND HALL, M. W. 2004. Increasing the applicability of scalar replacement. In *Proceedings of the ACM Symposium on Compiler Construction (CC'04)*. Lecture Notes in Computer Science, vol. 2985, Springer, Berlin,185–201.

SRC COMPUTERS INC. http://www.srccomp.com/.

STARBRIDGE-SYSTEMS INC. http://www.starbridgesystems.com.

STEFANOVIC, D. AND MARTONOSI, M. 2000. On availability of bit-narrow operations in general-purpose applications. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL'00)*. Lecture Notes in Computer Science, vol. 1896, Springer, Berlin, 412–421.

STEPHENSON, M., BABB, J., AND AMARASINGHE, S. 2000. Bidwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, 108–120.

STRETCH INC. http://www.stretchinc.com/.

SUTTER, B. D., MEI, B., BARTIC, A., AA, T. V., BEREKOVIC, M., MIGNOLET, J.-Y., CROES, K., COENE, P., CUPAC, M., COUVREUR, A., FOLENS, A., DUPONT, S., THIELEN, B. V., KANSTEIN, A., KIM, H.-S., AND KIM, S. J. 2006. Hardware and a tool chain for ADRES. In *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC'06)*. Lecture Notes in Computer Science, vol. 3985, Springer, Berlin, 425–430.

SYNOPSYS INC. 2000. Cocentric fixed-point designer. http://www.synopsys.com/.

SYNPLICITY INC. http://www.synplicity.com/.

TAKAYAMA, A., SHIBATA, Y., IWAI, K., AND AMANO, H. 2000. Dataflow partitioning and scheduling algorithms for WASMII, a virtual hardware. In *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications (FPL'00)*. Lecture Notes in Computer Science, vol. 1896, Springer, Berlin, 685–694.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMAN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2002. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro* 22, 2, 25–35.

TENSILICA INC. http://www.tensilica.com/.

TESSIER, R. AND BURLESON, W. 2001. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Process. 28*, 1–2, 7–27.

TODMAN, T., CONSTANTINIDES, G., WILTON, S., CHEUNG, P., LUK, W., AND MENCER, O. 2005. Reconfigurable computing: architectures and design methods. *IEE Proc. (Comput. Digital Techniques) 152*, 2, 193–207.

TRIMBERGER, S. 1998. Scheduling designs into a time-multiplexed FPGA. In *Proceedings of the ACM 6th International Symposium on Field-Programmable Gate Arrays (FPGA'98)*. ACM, New York, 153–160.

TRIPP, J. L., JACKSON, P. A., AND HUTCHINGS, B. 2002. Sea Cucumber: A synthesizing compiler for FPGAs. In *Proceeedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL'02)*. Lecture Notes in Computer Science, vol. 2438, Springer, Berlin, 875–885.

TRIPP, J. L., PETERSON, K. D., AHRENS, C., POZNANOVIC, J. D., AND GOKHALE, M. 2005. Trident: An FPGA compiler framework for floating-point algorithms. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'05)*. IEEE, Los Alamitos, CA, 317–322.

TRISCEND CORP. 2000. Triscend A7 CSoC family.

VAHID, F. 1995. Procedure exlining: A transformation for improved system and behavioral synthesis. In *Proceedings of the 8th International Symposium on System Synthesis (ISSS'95)*. ACM, New York, 84–89.

VAHID, F., LE, T. D., AND HSU, Y.-C. 1998. Functional partitioning improvements over structural partitioning for packaging constraints and synthesis: tool performance. *ACM Trans. Des. Autom. Electron. Syst. 3*, 2, 181–208.

VASILKO, M. AND AIT-BOUDAOUD, D. 1996. Architectural synthesis techniques for dynamically reconfigurable logic. In *Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL'96)*. Lecture Notes in Computer Science, vol. 1142, Springer, Berlin, 290–296.

WAINGOLD, E., TAYLOR, M., SRIKRISHNA, D., SARKAR, V., LEE, W., LEE, V., KIM, J., FRANK, M., FINCH, P., BARUA, R., BABB, J., AMARASINGHE, S., AND AGARWAL, A. 1997. Baring it all to software: Raw machines. *Computer 30*, 9, 86–93.

WEINHARDT, M. AND LUK, W. 2001a. Memory access optimisation for reconfigurable systems. *IEE Proc. (Comput. Digital Techniques) 148*, 3, 105–112.

WEINHARDT, M. AND LUK, W. 2001b. Pipeline vectorization. *IEEE Trans. Comput.-Aid. Des. Integrat. Circuits Syst. 20*, 2, 234–233.

WILLEMS, M., BÜRSGENS, V., KEDING, H., GRÖTKER, T., AND MEYR, H. 1997. System level fixed-point design based on an interpolative approach. In *Proceedings of the IEEE/ACM 37th Design Automation Conference (DAC'97)*. ACM, New York, 293–298.

WILSON, R., FRENCH, R., WILSON, C., AMARASINGHE, S., ANDERSON, J., TJIANG, S., LIAO, S., TSENG, C., HALL, M., LAM, M., AND HENNESSY, J. 1994. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Not*. 29, 12, 31–37.

WIRTH, N. 1998. Hardware compilation: Translating programs into circuits. *Computer 31*, 6, 25–31.

WIRTHLIN, M. J. 1995. A dynamic instruction set computer. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95)*. IEEE, Los Alamitos, CA, 99–107.

WIRTHLIN, M. J., HUTCHINGS, B., AND WORTH, C. 2001. Synthesizing RTL hardware from Java byte codes. In *Proceedings of the 11th International Conference on Field Programmable Logic and Applications (FPL'01)*. Lecture Notes in Computer Science, vol. 2147, Springer, Berlin, 123–132.

WITTING, R. AND CHOW, P. 1996. OneChip: An FPGA processor with reconfigurable logic. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96)*. IEEE, Los Alamitos, CA, 126–135.

WO, D. AND FORWARD, K. 1994. Compiling to the gate level for a reconfigurable coprocessor. In *Proceedings of the 2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*. IEEE, Los Alamitos, CA, 147–154.

WOLFE, M. J. 1995. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA.

XILINX INC. http://www.xilinx.com/.

XILINX INC. 2001. Virtex-II 1.5V, field-programmable gate arrays (v1.7). http://www.xilinx.com.

XPP. XPP: The eXtreme processor platform, PACT home page. http://www.pactxpp.com, PACT XPP Technologies AG, Munich.

YE, Z., SHENOY, N., AND BANERJEE, P. 2000a. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proceedings of the ACM Symposium on Field Programmable Gate Arrays (FPGA'2000)*. ACM, New York, 95–100.

YE, Z. A., MOSHOVOS, A., HAUCK, S., AND BANERJEE, P. 2000b. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, 225–235.

ZHANG, X. AND NG, K. W. 2000. A review of high-level synthesis for dynamically reconfigurable FPGAs. *Microprocess.Microsys. 24*, 4, 199–211.

ZIEGLER, H., MALUSARE, P., AND DINIZ, P. 2005. Array replication to increase parallelism in applications mapped to configurable architectures. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*. Lecture Notes in Computer Science, vol. 4339, Springer, Berlin, 63–72.

ZIEGLER, H., SO, B., HALL, M., AND DINIZ, P. 2002. Coarse-grain pipelining on multiple FPGA architectures. In *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*. IEEE, Los Alamitos, CA, 77–86.

Received November 2004; revised June 2008; accepted December 2008