ELSEVIER

Microprocessors and Microsystems 24 (2000) 199–211

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.nl/locate/micpro

A review of high-level synthesis for dynamically reconfigurable FPGAs

Xuejie Zhang[a,b,1,*], Kam W. Ng[a]

[a]*Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong, China*
[b]*Department of Computer Science, Yunnan University, Kumming, Yunnan, P.R. China*

Received 20 May 1999; revised 24 March 2000; accepted 28 April 2000

Abstract

Dynamically Reconfigurable Field Programmable Gate Arrays (DR FPGAs) change many of the basic assumptions of what hardware is. DR FPGA-based dynamically reconfigurable computing has become a powerful methodology for achieving high performance while minimizing the resource required in the implementation of many applications. The key to harnessing the power of DR FPGAs for most applications is to develop high-level synthesis tools for transforming automatically an algorithmic level behavioral specification into DR FPGA configurations. In this paper we survey the current state-of-the-art in high-level synthesis techniques for dynamically reconfigurable systems. The differences in high-level synthesis technology between classical systems and dynamically reconfigurable systems are discussed. Then, we describe the basic tasks in the high-level synthesis of dynamically reconfigurable systems. Finally, techniques that have been developed in the past few years for the high-level synthesis of dynamically reconfigurable systems are presented.

Keywords: Field programmable gate arrays; Dynamically reconfigurable systems; High-level synthesis

1. Introduction

Field Programmable Gate Arrays (FPGAs) are arrays of prefabricated logic blocks and wire segments. The functionality of the logic blocks and the inter-connection between the blocks are user programmable. The most popular type of FPGA technology is based on static memory (SRAM) technology, these FPGAs are programmed and reprogrammed by loading a circuit "bit-stream" into the internal configuration memory [39]. FPGAs have become the favored choice in implementing digital systems from "glue logic" to specific application accelerator to systems, which can achieve a high performance for general purpose computing [9,30].

Currently, Dynamically Reconfigurable FPGAs (DR FPGAs) have become viable with the introduction of devices that allow high-speed dynamic reconfiguration, e.g. the Xilinx XC6200 series (discontinued in 1998) [46] and the Virtex Series [47], the Atmel AT4000 and AT6000 series [2]. An FPGA is classified as dynamically reconfigurable if it allows reconfiguration of some logic blocks and wire segments, while some other programmable hardware is busy computing by having more than one on-chip SRAM bits controlling them [3,25]. Fig. 1 is a simplified representation of dynamic reconfiguration in progress. Several subcircuits are shown resident on the FPGA array, but only one is to be reconfigured. The operation of the appropriate subcircuit is suspended and only those logic cells that need to be modified are overwritten with new configuration data. The other active subcircuits continue to function throughout the reconfiguration period.

Dynamic reconfigurability offers important benefits for achieving high performance while minimizing the hardware resource required in the implementations of many applications. Several promising applications have already been reported in various areas including image processing [35], neural network [8,28], computer vision [21] and database searching [19].

While significant advances have been made, many obstacles still remain to be surmounted before dynamically reconfigurable technology can become widely adopted. Probably the most significant disadvantage of dynamic reconfigurability is the additional complexity that it introduces into the design cycle. Dynamically reconfigurable systems use a dynamic allocation scheme that re-allocates the FPGA's resources at run-time. Since the configuration of the FPGAs changes over time, it becomes difficult to understand the exact behavior of the system and there need to be some ways to make sure that the system behaves properly for all possible execution sequences. This is, in general, a

* Corresponding author. Fax: +852-2603-5024.
E-mail address: kwng@cse.cuhk.edu.hk (X. Zhang).

[1] The author is currently on leave from Yunnan University, People's Republic of China.

0141-9331/00/$ - see front matter
PII: S0141-9331(00)00074-0

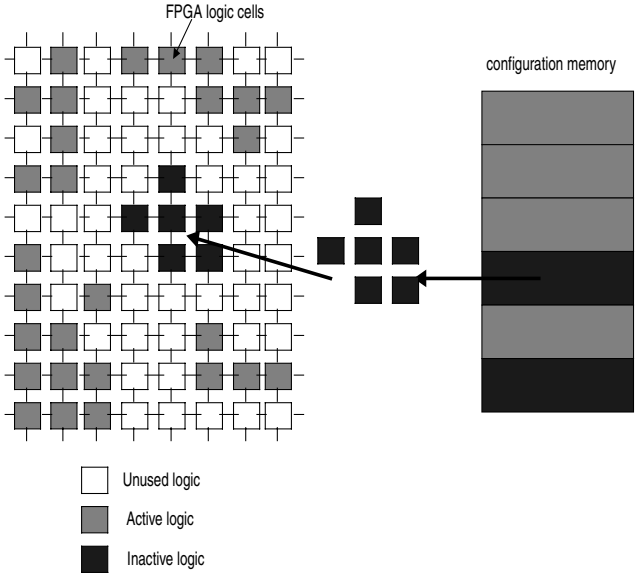

Fig. 1. Dynamically Reconfigurable FPGAs.

challenging problem to address [9]. Currently, the development of dynamically reconfigurable systems still uses the traditional capture-and-simulate design methodology that is an art involving tedious and error prone crafting of low-level design methodology [8,44].

With the growing complexity of current chips, including FPGAs, the use of automated synthesis techniques is an essential requirement of the design process [9,10,20]. Synthesis is a translation process from a behavioral description into a structural description, similar to the compilation of a high-level language program into an assembly program. Traditionally, synthesis has been subdivided into the following main categories:

- High-level synthesis (HLS) takes an abstract behavioral description of a digital circuit in the form of an algorithm and translates it into a structural description, while realizing the specified behavior at a register transfer level (RTL).
- Logic synthesis converts an RTL design into optimized combinational logic, and maps that logic onto available cells from a library in a particular technology.
- Layout synthesis converts an inter-connected set of cells, which describes the structure (topology) of a design, into the exact physical geometry (layout) of the design. It involves both the placements of the cells as well as their connection (routing).

Logic synthesis is the highest synthesis level currently in practical use for reconfigurable systems [9,12,27]. Application development with such logic synthesis tools still necessitates expertise in lower level hardware details. The developer has to be aware of the intricacies of the specific reconfigurable architecture in order to achieve a high performance. In addition, the compatibility issue among a range of implementations is also a problem. The logic synthesis results have to be modified before they can be ported even from one FPGA chip to another FPGA chip of the same family. The major challenge is to provide facilities for developing dynamically reconfigurable systems with much less effort and specialized knowledge than is required now.

On the other hand, in many applications employing dynamic reconfiguration the amount of time spent reconfiguring the FPGAs is critical. Long reconfiguration intervals can easily swamp the overall performance of the system. For reconfigurable applications, the following issues must also be dealt with:

- How can the use of dynamic reconfiguration be justified over conventional static FPGA-based methods?
- When is dynamic reconfiguration appropriate for an application?

In order to analyze these issues for dynamically reconfigurable applications, designers have to rely on the ability to specify their designs at higher levels of abstraction where dynamically reconfigurable design is easier to understand and tradeoff is more effective. This would also assist the designer in evaluating the option of employing dynamic reconfiguration. Therefore, the key to harnessing the power of DR FPGAs for most applications is to develop high-level synthesis tools for transforming automatically an abstract behavioral specification into DR FPGA configurations for the system. However, new applications and research of DR FPGAs are hindered by an almost complete absence of appropriate high-level synthesis tools [24].

High-level synthesis bridges the gap between behavioral specifications and their hardware realization, automatically generating circuit descriptions that can be used by logic synthesis. Unfortunately, whilst there are well-established techniques for the high-level synthesis of ASICs with fixed or static programmable architectures, DR FPGAs pose a difficult challenge for the development of high-level automated design tools due to the need for dynamic reconfigurability. Currently available high-level synthesis tools assume a static hardware model, therefore, they provide no high-level synthesis schema to support the dynamic reconfiguration. Hence, conventional high-level synthesis problems (such as partitioning, scheduling and module allocation) have to be modified to account for dynamic reconfiguration. Moreover, additional design effort such as design complexity for the reconfiguration controller and reconfiguration overhead are also introduced. The high-level synthesis system may also have to ensure not only producing a functionally and electrically correct implementation of the desired behavior but also considering the time to reconfigure the system [9,24].

In this paper, we review the current state-of-the-art in high-level synthesis techniques for dynamically reconfigurable systems. Section 2 presents an overview of the dynamically reconfigurable paradigm. In Section 3, we discuss

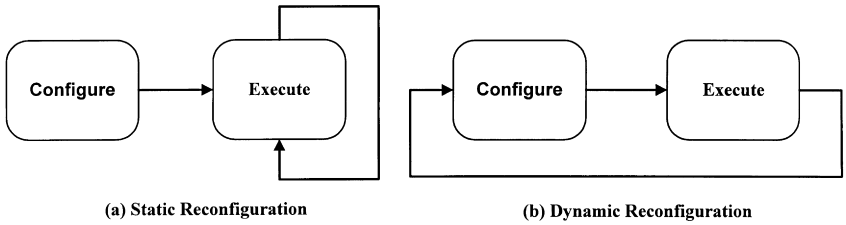

Fig. 2. Static and dynamic reconfiguration.

the problems and the tasks in high-level synthesis for dynamically reconfigurable systems. Then, we present the techniques that have been developed for solving these problems in Section 4. Finally, we discuss the challenges for future work, pointing out where development is still needed to let dynamically reconfigurable systems achieve all of their promises.

2. Overview of dynamically reconfigurable systems

FPGAs have received increasing attention recently due to their short turnaround time, user reconfigurability and low development costs. FPGA-based computing systems have become a powerful implementation methodology for achieving a high performance. By mapping applications onto FPGA hardware resources, extremely efficient computations can be performed [9]. The use of FPGAs has been classified broadly into three main categories: rapid prototyping, system implementation, and dynamically reconfigurable systems [24]. The last of these areas has been described as being perhaps the most innovative application for FPGAs [42].

Reconfigurable systems can be broadly classified as having one of two types of reconfigurability: static or dynamic reconfigurability [36]. In this paper, static reconfiguration refers to having the ability to reconfigure a system, but once programmed, its configuration remains on the FPGA for the duration of the application. In contrast, dynamic reconfiguration is defined as the selective updating of a subsection of an FPGA's programmable logic and routing resources while the remainder of the device's programmable resources continue to function without interruption [25]. Thus, whereas static reconfiguration applications configure the FPGAs once before execution, dynamic reconfiguration applications typically reconfigure them many times during the normal operation of a single application as seen in Fig. 2. The concept of dynamic reconfiguration has acquired various names: run-time reconfiguration [18], on-line reconfiguration [1], logic caching [25], virtual hardware [4] and DPGA [5]. Throughout this paper we will use the term dynamic reconfiguration [24].

2.1. The advantages of dynamic reconfiguration

The logic capacity of FPGA technology is always going to be poorer than that of tailored ASIC technology due to area and time overhead for providing uncommitted logic and routing, as well as the associated control circuitry. Thus, benefits must come from *reconfigurability* and *flexibility*, while still providing significant speed benefits over purely software solutions. Dynamic reconfiguration provides additional opportunities for reconfigurable system implementation that is unavailable within statically reconfigurable systems. Specifically, two different conditions motivate the use of dynamic reconfigurability: the presence of idle or underutilized hardware and the need to partition a large system onto limited FPGA resources. Each of these motivations will be described in detail below.

2.1.1. Supporting the temporal locality of applications

The temporal locality of an application is interpreted as the presence of idle or underutilized operations within a reconfigurable application. In other words, individual operations within a design may remain idle because they are not needed at a given time or they cannot immediately contribute to the computation. For example, data-dependencies within an algorithm may dictate that an operator must wait for the completion of a different operation before proceeding, or an application-specific operation may be infrequently needed in the schedule of a computation.

Dynamic reconfiguration can be used to remove such idle operations from the system and replace them with other more useful operations. Such dynamic removal of hardware allows an operation to proceed with fewer FPGA resources than possible within a static system. Therefore, dynamic reconfiguration allows a new design methodology for producing large designs that are too big for the available hardware resources on the FPGA chips.

FPGAs have been described as programmable active memories [25]. If viewed from this perspective, like the virtual memory in a normal processor, a dynamically reconfigurable FPGA can be viewed as "Virtual Hardware", that is, hardware paged into the FPGA only when needed. Therefore, the profile of the circuitry that is active on the DR FPGAs may be adjusted dynamically to match the requirements of the temporal locality of an application. Since idle operations within the application no long consume valuable resources, the application may operate with fewer resources than possible within a static reconfigurable system. *Dynamic reconfigurability* is used to ensure that FPGA resources are used more efficiently. The opportunities for deploying dynamic reconfiguration are increasing as the

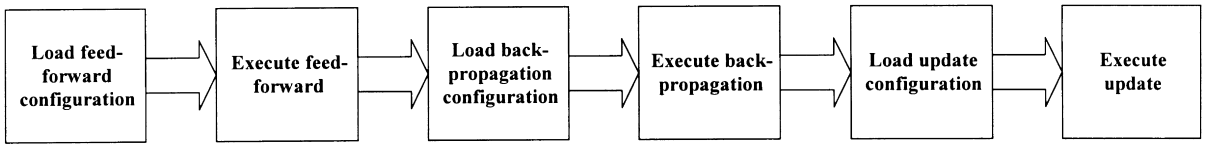

Fig. 3. Full reconfiguration.

gate counts of individual FPGAs continue to improve. As larger DR FPGAs become available, the complexity of the systems that can be integrated into a single FPGA increases. Currently, there are many research projects developing computing architectures for the exploitation of temporal locality within an application [6,12].

2.1.2. Supporting the functional locality of applications

The functional locality of operations is interpreted as the application-specific computing nature of an application. In other words, the functional locality of operations requires the operators used within an architecture to be the specialization required by the application.

For a static system, when a large computing system cannot fit within the finite resource of a reconfigurable system, the application must be partitioned and scheduled onto the fixed and static resource. However, a single static architecture designed to execute different algorithmic partitions within a sequential execution schedule must be general-purpose enough to support all computational variations found within the application. The reuse of hardware for several algorithmic partitions limits the amount of specialization that can take place and forces the inclusion of general-purpose architectural features. This usually results in less efficient structures and cannot be used to support the functional locality of operations.

However, for large special-purpose computing systems that require partitioning, dynamic reconfiguration can be used to support the functional locality of operations. Instead of providing a static circuit that is generalized to support all computational variations found within an application, the application is partitioned into special-purpose operations that are reconfigured at run-time. This *flexibility* allows hardware resources to be tailored to the run-time profile of the application more efficiently than with static architecture. The exploitation of functional locality of operations within an application has been reported in the literature [43].

2.2. Implementation strategy of dynamically reconfigurable systems

Several dynamically reconfigurable applications have been developed to demonstrate the significant performance improvements possible with DR FPGAs [19,35]. We could categorize these implementation approaches using three criteria: the grain size of temporal partitioning, the methodology of exploiting functional locality, and the reconfiguration control mechanism. Their differences and their impact on the high-level synthesis for dynamically reconfigurable design will be discussed.

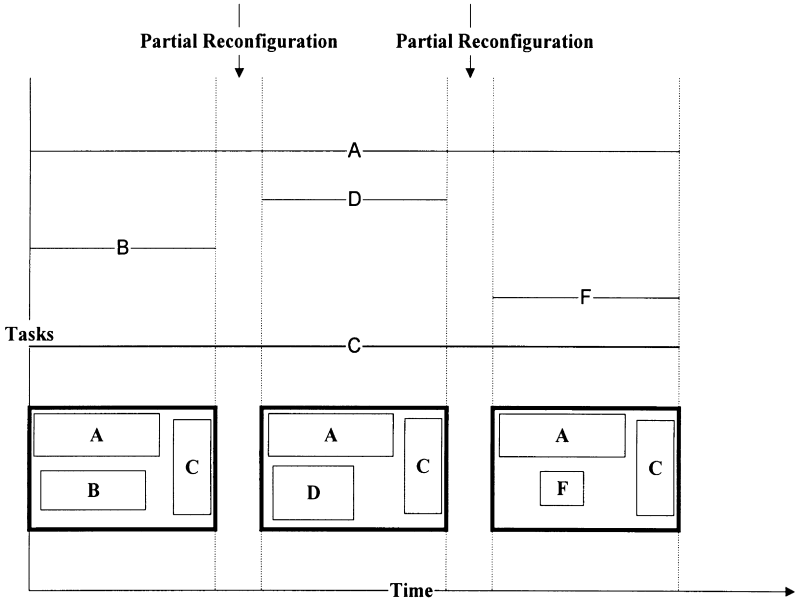

Fig. 4. Combined floorplan and chart perspectives for partial reconfiguration.

2.2.1. Fully reconfigurable systems and partially reconfigurable systems

From the perspective of supporting temporal locality, there are two basic approaches that can be used to implement dynamically reconfigurable applications: full reconfiguration and partial reconfiguration [12]. Both techniques use multiple configurations for a single application and both techniques reconfigure FPGAs during execution of the application. The principal difference between these two techniques is the grain size of the dynamic hardware allocated.

Fully reconfigurable systems allocate all FPGA resources in each configuration step. Full reconfiguration applications are partitioned into distinct temporal phases, where each phase is implemented as a single system-wide configuration that occupies all FPGA system resources. As an example, Brigham Young University's RRANN project demonstrates the full reconfiguration system implementation approach [6]. RRANN implements the popular neural network back-propagation-training algorithm as three time-exclusive FPGA configurations: feed-forward, back-propagation and update. Fig. 3 shows how the system operation consists of sequencing through these three configurations at run-time.

Observably, the primary task when implementing fully reconfigurable design is to partition temporally the application into approximately equal-sized partitions, to efficiently use reconfigurations [12]. If it is not possible to partition evenly the application, inefficient use of FPGA resources will result. Unfortunately, currently no tools support this equal-sized partitioning step. Since the partitions are coarse grained and circuit interfaces are fixed between configurations, conventional high-level tools can be used efficiently once the equal-sized partitioning step has been completed. Each partition can be designed and implemented as one independent configuration. After the partitioning and high-level synthesis are performed on each configuration, they are integrated at the lower levels, i.e. at RTL or at the logic level.

Partial reconfiguration may reconfigure any percentage of the reconfigurable resources at any time. Partially reconfigurable FPGAs offer a faster way to change an active FPGA circuit since only those parts that need to be reconfigured are interrupted. This can reduce the amount of time spent on downloading the configuration, and can lead to a more efficient run-time allocation of hardware. Fig. 4 provides a conceptual diagram of partially reconfigurable systems. Task A and Task C are to be permanently resident on the FPGAs, and Task B, Task D and Task F are to be swapped in and out during the operation of the circuit. Between each execution stage, only a subset of the reconfigurable hardware is reconfigured.

The main advantage that partially reconfigurable systems provides over fully reconfigurable systems is the ability to create fine-grained reconfigurations that make more efficient use of FPGA resources. Partially reconfigurable designs are typically implemented by partitioning an application into a set of fine-grained tasks based on the functional locality of the application [12]. Each of these tasks is implemented as a distinct configuration and these configurations are then downloaded to the FPGAs as necessary, during the operation of the application. However, the fine-grained partitioning is in general a challenging problem to address, it can cause a very high design penalty because of the increased flexibility and complexity of the system. Moreover, several of these configurations may be loaded simultaneously and each configuration may consume any portion of the FPGA resources. Unlike fully reconfigurable applications where configuration interfaces remain fixed, partially reconfigurable applications allow these interfaces to change with each configuration. In order to ensure that all configurations will interface correctly, there need to be some new ways to make sure that the system behaves properly for all possible execution sequences. Since the configuration of DR FPGAs can change dynamically, current high-level synthesis techniques are largely useless for partially reconfigurable designs.

2.2.2. Domain-specific and application-specific approach

From the perspective of supporting functional locality, there are two development methods for designs with reconfigurable elements at run-time: *domain-specific* and *application-specific* approach [13]. Both techniques perform dynamic reconfiguration by specializing the computing structure to the problem of interest. The principal difference between these two techniques is the way they exploit the functional locality in an application.

An application-specific architecture is defined here as a minimal reuse or maximal functional locality computing architecture. In this approach, nearly every part of a design is a custom circuit element that has been carefully tailored to apply only to a single or specific computation task. However, this can cause a very high design penalty because of the high flexibility and complexity of the system. The lack of reuse occurs because of the need to completely specialize each operation to secure the highest possible performance at the lowest possible cost. It is not feasible to support such architecture at a high level, therefore, the tools used in the architecture design are low-level design tools (schematic capture, logic synthesis) where the design is described in low-level terms.

The domain-specific approach is implemented by using libraries of reusable components that are highly optimized routines capable of being reused across a range of applications within a given domain. This approach exploits directly the reconfigurability of DR FPGAs through reuse of these domain-specific library elements to implement a wide variety of applications within the specific domain. The domain-specific elements will typically be developed by FPGA-design experts and because they will be reused in many different applications, substantial design effort can be employed to achieve highly optimized circuits. Once a large library of domain-specific circuits becomes available, it forms a new level of computing primitives that hide the

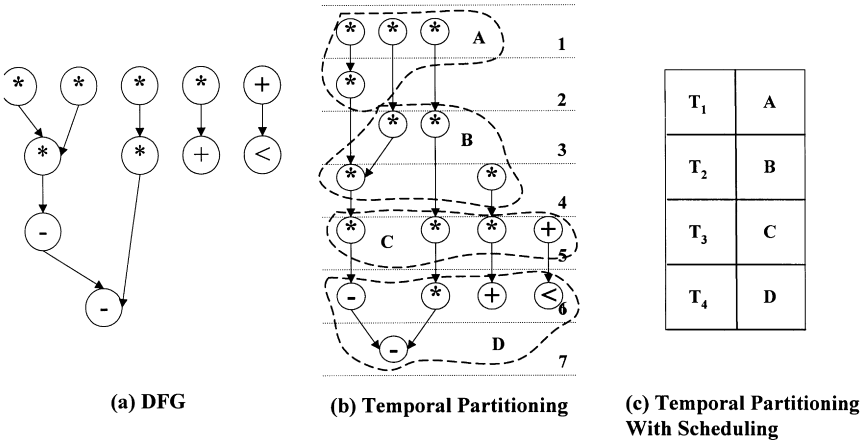

Fig. 5. Example of temporal partitioning.

underlying complexity of optimized DR FPGA implementations. By hiding the DR FPGA details, the application can be described in terms of higher-level operations that are selected from a library and can be synthesized at a higher level. This reduces both development and compilation time. Therefore, the domain-specific approach brings important opportunities for the development of high-level synthesis systems [13].

2.2.3. Co-processors vs. embedded processors

In the final implementation of a dynamically reconfigurable system, the responsibility for controlling reconfiguration of the FPGAs must be allocated to an external microcontroller, the FPGA itself, or to another hardware device. If we examine the way dynamic reconfiguration is accomplished, we could divide dynamically reconfigurable systems into the following groups: co-processors and embedded processors.

While the controller may be implemented in hardware or software, the use of a microprocessor to control reconfiguration is the more flexible option and is currently the most common choice. This is commonly referred to as the process of hardware/software codesign [7].

If the controlling microprocessor of the above architecture is removed, the DR FPGAs become a stand-alone embedded processor. The DR FPGAs, used in this way, yield an architecture of a self-controlling dynamically reconfigurable system [25]. Since the configuration changes over time under the control of the configuration itself, there is a need for new techniques to support the self-modifying system behavior [25].

3. High-level synthesis issues for dynamically reconfigurable systems

The traditional high-level synthesis task is to take a behavioral or functional level specification (e.g. an algorithm) of a system and a set of constraints and goals to be satisfied, and to find a register transfer level (RTL) structure that implements the behavior while satisfying the goals and constraints [20,29,34]. The RTL structure is composed of a data-path and a controller. In other words, from the input specification, the synthesis system produces a description of a data-path that is a network of functional units, registers, multiplexers and busses (typically described by a netlist). Moreover, the synthesis system must also produce the specification of the control part. In synchronous systems, the only kind we consider in this paper, control can be provided by one or more finite state machines, specified in terms of microcode or random logic.

However, dynamically reconfigurable design changes many of the basic assumptions in the high-level synthesis process. The flexibility of dynamically reconfigurable systems (multiple configurations, partial reconfiguration, etc.) requires new methodologies and high-level synthesis algorithms to be developed as conventional high-level synthesis techniques do not consider the dynamic nature of dynamically reconfigurable systems.

3.1. Basic differences in the high-level synthesis process

A high-level synthesis system acts as a compiler that maps a high-level specification into a structure. The main steps involved in a high-level synthesis system are:

- Compilation of the high-level description into an internal representation based on the design model, usually a graph-based or an algebraic process-based model.
- Scheduling, module allocation and binding in synthesis are the core of transforming behavior into structure. Scheduling involves assigning each operation to a time step. A time step is a fundamental sequencing unit in synchronous systems, it corresponds to a control step which is equivalent to a state in a FSM or a microprogram step in a microprogrammed controller. Allocation and binding assigns each operation to hardware. Module allocation involves both the selection of the type and the

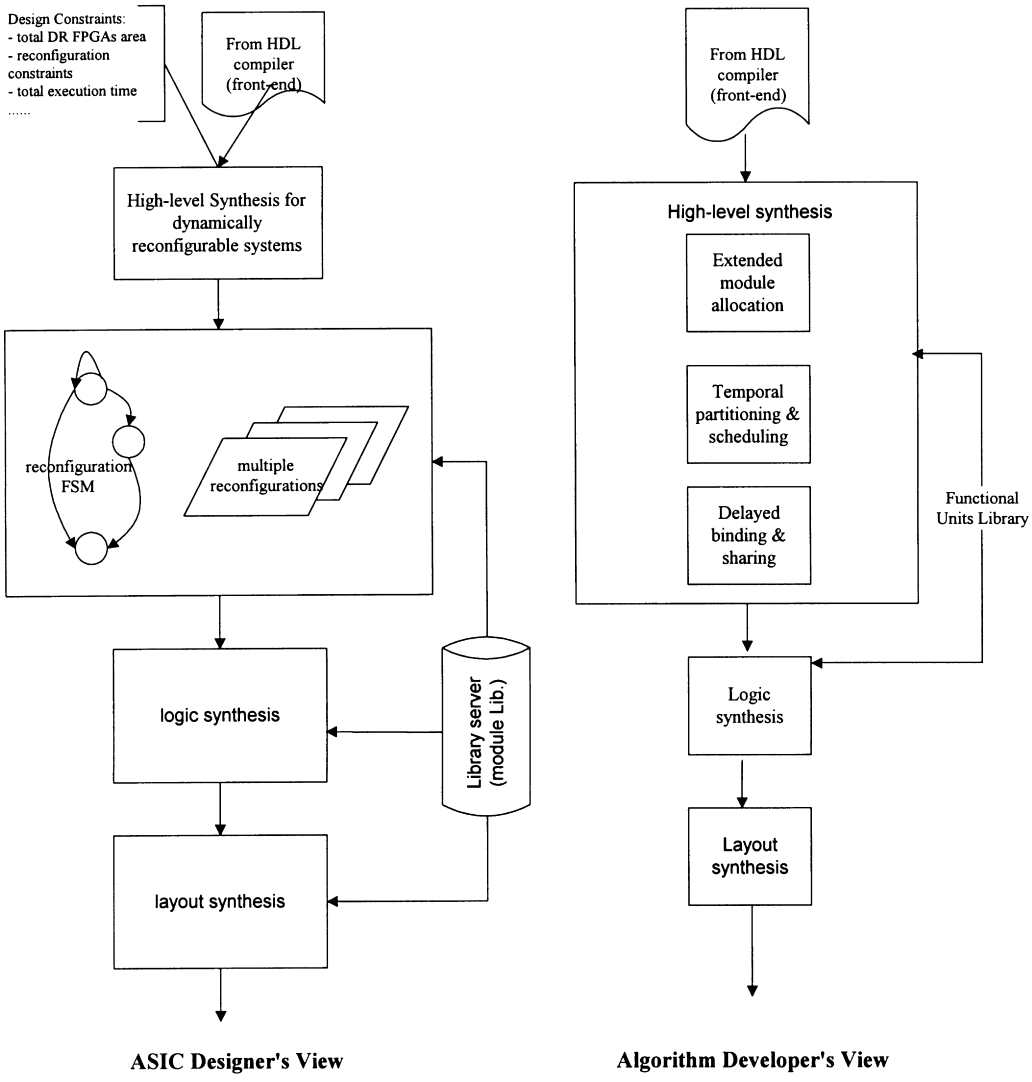

Fig. 6. High-level design flow for a dynamically reconfigurable system.

quantity of hardware modules from a library, and binding involves mapping of each operation to the selected hardware. Although scheduling and binding are two distinct tasks, their performances are closely related. In order to deal with the inter-dependency between these two synthesis subtasks, many strategies exist [20,29,34].

- Partitioning deals with the division of the internal representation into subrepresentations in order to reduce the problem size.
- Output generation produces a design that is passed to logic or RTL synthesis. The resulting design is usually composed of a data-path and a controller.

Traditional hardware design focuses on the development of a static circuit of fixed size, topology and functionality. The static nature of the target design entity is reflected in the high-level synthesis process and design methodologies. In the area of high-level synthesis, the two key differences between classical systems and dynamically reconfigurable systems are how to exploit the temporal and functional localities of applications in the high-level synthesis process and how to minimize the reconfiguration overhead of synthesizing such systems.

Observably, the most important difference between static systems and dynamically reconfigurable systems is the temporal nature of developing such systems. The exploitation of temporal locality within an application necessitates a new temporal partitioning process of the specification [12,15,16]. Temporal partitioning divides an application into time-exclusive segments that do not need to or cannot run concurrently. For example, consider a popular differential equation solver high-level synthesis benchmark. The complete data-flow graph (DFG) for the benchmark is shown in Fig. 5(a). A possible implementation of the DFG as a dynamically reconfigurable system involves grouping operations such that they may be subsequently scheduled. In other words, this problem may be solved by partitioning temporally the DFG along data-flow boundaries. Fig. 5(b)

illustrates a temporal partitioning example where the DFG is partitioned into four (labeled A, B, C, and D) segments. Fig. 5(c) illustrates one such temporal partitioning where segments A, B, C and D are mapped to time steps T_1, T_2, T_3 and T_4, respectively.

Similar to structural partitioning where circuit elements are organized into strongly connected partitions, temporal partitioning is a design process that organizes operations into strongly concurrent partitions. Typically, a dynamically reconfigurable application is temporally partitioned by breaking it down into distinct phases or operational modes. Each of these phases or modes is then designed as a distinct circuit module (FPGA configuration). These configurations are then downloaded into the FPGAs as required by the application at run-time. Therefore, dynamic reconfiguration extends the scheduling problem by an additional dimension—temporal partitioning. The effectiveness of temporal partitions has been presented in Refs. [11,45]; however, the partitioning is accomplished manually. Therefore, high-level synthesis techniques to automatically partition designs temporally are needed.

Another key advantage of dynamic reconfiguration is that we can specialize the computing engine to closely match the needs of the functional locality in an application. This flexibility allows hardware resources to be tailored to the run-time profile of the application more efficiently than with static architecture. In particular, the later binding time associated with reconfigurable logic netlists gives the reconfigurable architect the opportunity to specialize his design more precisely to the particular use to which the device will be employed during an operational epoch. The functions performed by a reconfigurable logic design, however, need not be bound until loaded into the reconfigurable logic device. At this point the bindings need only to be generic enough to operate until reconfiguration, or, more precisely, they only need to be generic enough that reconfiguration resource requirements do not preclude proper operation or deviate from design performance. To support the exploitation of functional locality within an application, the high-level synthesis framework needs to pay careful attention to the binding time of values in the dynamically reconfigurable designs.

Furthermore, with conventional systems, the module allocation process attempts to map the logic spatially so that it occupies the smallest area, and produces results as quickly as possible. In a dynamically reconfigurable system one must also consider the time to reconfigure the system, and its effects on the performance of the system. In many applications employing reconfiguration, the amount of time spent reconfiguring the FPGA is critical. Long reconfiguration intervals can easily swamp the overall performance gain of the system and therefore reconfiguration should be kept to a minimum [9]. Thus, it will be critical to allocate logic together properly so that a given configuration can do as much work as possible, allowing a greater portion of the task to be completed between reconfigurations. The configuration allocation process has a key impact on the quality of the final design. A good configuration allocation will be tightly coupled and performs its task relatively independent of other configurations. Since the module allocation step must make a trade-off between maximizing resource usage and minimizing reconfiguration overhead in both space and time, the synthesis methodologies must consider how to automatically vary the granularity of reconfiguring regions and estimate the resulting impact on size, performance and complexity of the reconfiguration controller. The optimization issue that must be addressed is how to allocate multiple configurations of a dynamically reconfigurable design that optimizes for resource usage and reconfiguration overhead.

Therefore, dynamically reconfigurable design extends the high-level synthesis design space by dynamic reconfigurations. For a dynamically reconfigurable design, the structure realizable on the DR FPGAs is no longer a static structure but a set of structures that describes the possible configurations. Fig. 6 shows the design flow involved in the high-level synthesis of dynamically reconfigurable systems from two different views. The fundamentally different concept makes conventional high-level synthesis techniques unsuitable for dynamically reconfigurable applications. The higher flexibility of dynamically reconfigurable applications complicates the overall high-level synthesis process from design model to synthesis algorithms.

3.2. The design model

High-level synthesis is a transformation process generally based on a well-defined design model. This is generally called an intermediate form. The different steps of the high-level synthesis process can be explained as a transformation of this design model. In general, a design model refers to an abstraction over the target system, capturing important relationships between important components of the system. Any design model must be able to abstract the following:

- functionality or the behavior of the system;
- constraints on properties of the behavior.

The intermediate representations model the system from several design aspects. Requirements, behavior, algorithms, resources, along with constraints between these aspects are specified in the appropriate models. The design models define a design space of potential alternative implementations, involving choices of synthesis algorithms, optimization strategies, and process placement and technology implementation.

A variety of intermediate representations have been proposed to capture various specification elements in high-level system specification in a form suitable for further processing during synthesis. These include DFGs [40], mixed control data-flow graphs (CDFG) [38], timed decision tables (TDT) [33], and various flavors of graph-based

and table-based formalisms sometimes augmented with global flow information such as module call graphs (MCG). Although these representations capture common features such as data and control dependencies among operations [29], they are not able to capture the temporal or functional locality among operations as well as the inter-configuration relationships. Since traditional design models do not allow for a dynamic structured approach that can be used to define the run-time reconfiguration behaviors, new modeling techniques need to be developed for expressing the reconfigurability of operations in developing a dynamically reconfigurable design. It is the key to understanding dynamically reconfigurable designs and to synthesizing them effectively.

Two difficult issues must be tackled in a formal treatment of the high-level synthesis of dynamically reconfigurable systems:

- the abstract specification of the temporal dimension of the system; and
- the concise formulation of the dynamic reconfiguration behavior restricted with run-time constraints.

3.3. The synthesis algorithm and the optimization strategies

The algorithms used for dynamically reconfigurable design (such as scheduling, module allocation and binding) differ from classical high-level synthesis because of different problem formulations. Conventional high-level synthesis problems (such as scheduling, module allocation and binding) have to be modified to account for dynamic reconfiguration. For example, an important task of high-level synthesis is scheduling. Scheduling in high-level synthesis is performed under resource constraints and/or timing constraints [14]. The following difference between classic and dynamic scheduling exists.

1. Classic scheduling: find a schedule with the shortest execution time for a given number and type of resource.
2. Dynamic scheduling: find a schedule and a temporal partitioning for subsequent reconfigurations leading to the shortest execution time (including reconfiguration time) in a given total available area of DR FPGAs [41].

In addition, in the high-level synthesis of dynamically reconfigurable systems, temporal partitioning, scheduling and module allocation are closely interrelated: whenever one is performed before the other tasks, additional constraints are imposed on the operations with respect to the other tasks. Therefore, the issue here is how to deal with the inter-dependency among partitioning, scheduling and module allocation. The optimality criteria now shift from basic blocks to the whole design. The flexibility of dynamically reconfigurable systems (multiple configurations, partial reconfiguration, etc.) requires new methodologies and high-level synthesis algorithms to be developed as the conventional high-level synthesis techniques do not consider the dynamic nature of dynamically reconfigurable systems.

3.4. The configuration controller

Whenever dynamic reconfiguration is employed, an overhead associated with controlling the sequence of reconfiguration is implicit. In other words, in the final implementation of a dynamically reconfigurable system, the responsibility for controlling reconfiguration of the DR FPGA must be allocated to an external microcontroller, to the FPGA itself, or to another hardware device. We call the circuit or software that performs this function the reconfiguration controller. In any case, it would be advantageous to be able to synthesize the algorithm for the controller automatically from the information contained in the schedule control modules.

4. Existing high-level synthesis techniques

In this section, we survey some existing high-level synthesis techniques for dynamically reconfigurable systems.

4.1. Design models for dynamically reconfigurable systems

A variety of design models have been proposed to capture the semantics of the specification in a form suitable for further processing during the synthesis of dynamically reconfigurable systems.

Rath et al. [33] used an intermediate tabular model called the TDT to separate the control and data-path and to divide the data-path into different control paths. A TDT representation of system behavior consists of three major parts: (1) a control section which is a set of rules, or a list of control path segments; (2) a delay table which lists the execution delay of each action or data-path operation in the model; and (3) an additional table describing the data-dependency, serialization and concurrency type specified between each pair of operations. Their synthesis methodology first translates the high-level system specification (e.g. VHDL or HardwareC) into TDTs. For abstracting out the temporal dimension of systems, the TDT specification is transformed to a control-flow graph (CFG) representation. The temporal locality of operations is specified by state transitions. The main synthesis task involves partitioning this CFG into connected subgraphs. Each subgraph groups a set of transitions starting from a “root” state. The grouping is done so that the cost of all transitions in each group is less than the available resources on the DR FPGAs. A group is constructed by performing a breadth-first coverage of successive transitions from the root state. The subgraphs identified by the partitioning step form the different configurations that can be swapped in and out of the DR FPGAs. However, the synthesis method described in [33] is only suitable for designing fully reconfigurable and control-dominated systems, as the

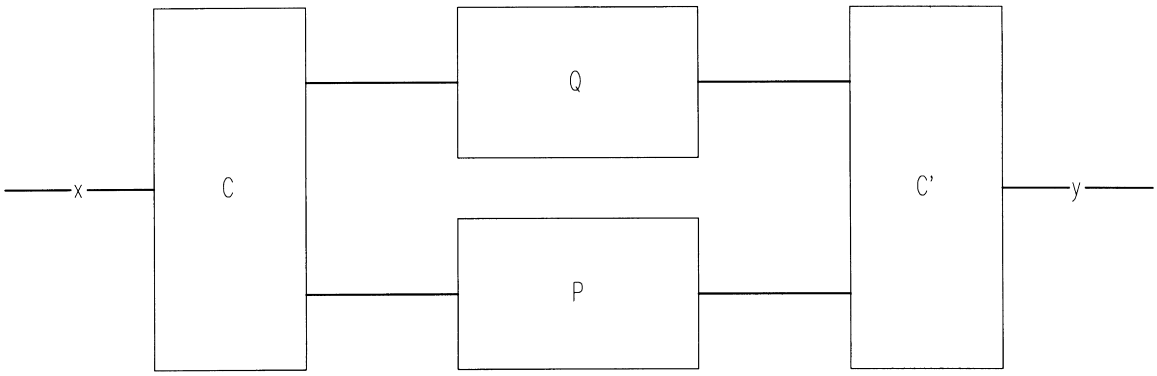

Fig. 7. A static network modeling a dynamically reconfigurable design.

TDT expressions will not be able to capture the inter-configuration relations and data-flow boundaries that are present in high-level descriptions. Another limitation of their approach is that they analyze and partition the system specification at synthesis-time and not at run-time. This simplifies the design task but may not provide the most optimal reconfigurable design partitions.

The Unified Specification Model (USM) developed at the University of Cincinnati [32] is a hierarchical representation to capture succinctly inter-task level control and data-flow, as well as intra-task (operation) level dependencies. At the highest level of the USM are two types of objects called tasks and memory segments. Tasks in the USM represent elements of computation, and memory segments represent elements of data storage. USM objects can be connected through edges that are either channels or dependencies. USM provides a task graph representation with the control flow explicitly captured and the data-flow clearly specified through channels and memories. USM provides a suitable representation for the temporal partitioning process: inter-task dependency edges and the communication model using memories and channels directly help the temporal partitioning step in deciding the partitions [17,31].

Luk and Cheung presented a static network-based model for specifying, visualizing and developing dynamically reconfigurable designs [23]. Their idea is to use a static network to capture the dynamic behavior of reconfiguration. A logic block that can be configured to behave either as P or as Q is described by a network with P and Q sandwiched between two control blocks C and C' as shown in Fig. 7. After the traditional high-level synthesis is performed, the netlist is transformed into a static network with control blocks connecting possible configurations for each reconfigurable component. The model provides a methodology for developing dynamically reconfigurable systems. However, the model is specified at the lower level (RTL). At this level, the results are already suboptimal and harder to optimize.

4.2. Temporal partitioning and scheduling techniques

Temporal partitioning involves dividing the design specification into multiple segments that execute one after the other on the reconfigurable system. Currently, techniques to automatically partition designs temporally have extended existing scheduling and clustering techniques of high-level synthesis [15,17,31,37,40,41].

A scheduling and temporal partitioning algorithm for dynamically reconfigurable systems based on a DFG model has been reported in Refs. [40,41], and is based on a static-list scheduling heuristic enhanced to consider dynamic area constraint. The algorithm is executed in two phases. First, the static priority list is computed, the operation selection criteria is based on the mobility of the individual operations. Secondly, while all operations have not been scheduled, the ready operations from the priority list are placed in the available area until the complete area is used up or data-dependency conflict occurs. Next, the redundant function units from the previous control steps are “removed” from the scheduler and the next iteration begins. This process incrementally builds the data-path schedule and partitions it into full or partial configurations. However, the approach is limited to data-path synthesis and does not consider other interdependent synthesis tasks. Their algorithm is also unable to handle efficiently large designs with complex control and reconfigurability features.

The SPARCS system accepts a behavioral specification of an application in the form of a set of tasks [17,31]. Tasks are modeled in behavior-level VHDL. The VHDL representation is compiled as an internal representation, a task graph and a memory graph. A task graph captures the task-level dependencies and data-flow between the tasks. A memory graph represents the relationship among the tasks and memories explicitly and will be used during configuration partitioning. The SPARCS system contains a temporal partitioning tool to divide and schedule temporally the tasks on the reconfigurable architecture, and a spatial partitioning tool to map the tasks to individual FPGAs. The temporal partitioning tool heuristically estimates the upper bound on the number of temporal segments for building a Non-linear Programming (NLP) formulation. They use a variation of Bournemouth University’s list-scheduling algorithm [40,41] for this estimation. The temporal partitioning

problem is solved by an Integer Linear Programming (ILP) solver. However, their approach suffers from long runtimes, and is only worth investigating if the estimation of costs and performance can be proven to be highly accurate and the design size is rather small.

The University of Cincinnati has presented a technique for temporal partitioning and scheduling of DFGs in the time domain [15]. Temporal partitioning and scheduling is formulated as an extended k-way-partitioning problem where partitioning and design point selection are scheduled sequentially. Related research that addresses the temporal partitioning issue involves partitioning gate-level designs [37]. Since the design to be partitioned is already synthesized, different synthesis options for achieving lesser latency partitioned solutions cannot be explored.

4.3. Configuration controller synthesis technique

The strategy for implementing a configuration controller has been reported in Ref. [26], which presented a simulation tool for DR FPGAs and introduced a new technique called dynamic circuit switching (DCS) for simulation purposes [24]. The tool can also be used as a means of specifying the behavior of the reconfiguration controller used to control a dynamically reconfigurable system. However, no existing tools permit the reconfiguration controller to be synthesized automatically from the high-level specification.

4.4. Library-based approach

To provide realistic estimates on technology-dependent metrics at various stages during the high-level synthesis process (temporal partitioning, scheduling and binding), high-level synthesis systems must provide a library server of generic functional modules. The parameterized FPGA libraries are becoming increasingly popular in the high-level synthesis process [22]. However, reconfigurable designs impose additional requirements on the parameterized FPGA libraries to support dynamic reconfiguration. The libraries will be used in the high-level synthesis process, they must be enhanced with information about the amount of resources consumed and the reconfiguration time associated with each configuration.

4.5. High-level synthesis framework for dynamically reconfigurable design

To produce a usable framework for the high-level synthesis of dynamically reconfigurable designs, the desirable features for a synthesis system can be summarized as follows:

- support for partitioning of an application subjected to complex dynamic resource and interface constraints;
- ability to produce a wide range of implementations that are fully or partially reconfigurable;
- support for synthesizing reconfiguration controller structures automatically;
- support for simulating, optimizing and validating reconfigurable designs.

Lysaght and Stockwood [24] have developed a simulation approach for dynamically reconfigurable systems. In their methodology, a design description is annotated with information on what signals will cause a given subcircuit to be loaded into hardware, and how long the reconfiguration process takes. This initial description is then transformed automatically into a new description with explicit isolation switches between the subcircuits controlled by schedulers based upon the circuit annotations. Whenever a given component should not be present in the FPGA, or has not yet been completely loaded into the chip, these isolation switches set the value of all those subcircuit signals to unknown. Thus, any circuitry that is sensitive to the value of signals from circuitry that is not currently loaded will also go to the unknown state, demonstrating the logical flow. When the circuit is properly loaded into the FPGA, the isolation switches simply pass on all signal values. This new representation of the circuit can now be simulated normally allowing standard software simulators to support the execution of dynamically reconfigurable systems.

Another system which is concerned with an integrated synthesis and partitioning strategy for adaptive and reconfigurable computer systems is the SPARCS system, which is an integrated design environment for partitioning and synthesizing behavioral specifications onto multi-FPGAs [17,43]. This system contains a temporal partitioning tool to temporally divide and schedule the tasks on to the reconfigurable architecture; a spatial partitioning tool to map the tasks to individual FPGA. One of the unique features of the SPARCS system is the tight integration of the architecture-independent partitioning and synthesis techniques to accurately predict and control design performance and resource utilization. Although SPARCS allows the system to be specified using task-level modules, the partitioning and synthesis techniques applied in these modules consider only one task at a time, preventing the synthesis from utilizing the degrees of freedom from the other tasks during the synthesis of a single module.

5. Future directions

Many research projects have been carried out, however, the high-level synthesis for dynamically reconfigurable application is still in its infancy. There are a number of areas where high-level synthesis must continue to develop if it is to become a useful tool in designing dynamically reconfigurable systems. Some of these are specific design problems that still need to be solved:

- Dynamic reconfigurability is still one of the most

important issues that need to be investigated further. Future work should focus on developing more general design models for representing the dynamic reconfigurable design, and on improving run-time support. To be an effective design model for dynamically reconfigurable designs, the model must support not only the definition and enforcement of the constraints required by dynamically reconfigurable designs, but it must allow for a dynamic structured approach that can be used to abstract, analyze and synthesize designs which contain elements that can be reconfigured at run-time.

- Prior to logic or layout synthesis, some means of estimating the reconfiguration metrics are necessary so that system performance evaluation can be taken into account as early in the design process as possible. Work must be carried out in dealing with the interaction between high-level synthesis and layout. The high-level synthesis interface with logic synthesis is still a neglected issue.
- Techniques need to be developed for parameterized libraries to support reconfiguration.
- High-level synthesis for dynamically reconfigurable applications that can synthesize dynamic reconfiguration implementations from C, C++ or Java language descriptions must be developed.

Acknowledgements

The work described in this paper was partly supported by the following grants: the Research Grants Council of the Hong Kong Special Administrative Region (Direct Grant for Research 1998/99—Project Code: 2050196; Earmarked Grant 1999/2000—CUHK4408/99E); and Yunnan Province Young Scholar Grant.

References

[1] U. Apel, On-line software extension and modification, Electrical Communication 64 (4) (1990) 66–72.
[2] Atmel Inc, AT 6000 Series, 1997.
[3] M. Butts, Future directions of dynamically reprogrammable systems, Proceedings of the IEEE 1995 Custom Integrated Circuits Conference, 1995, pp. 487–494.
[4] S. Casselman, Virtual computing and the virtual computer, in: Proceedings of the IEEE Workshop FPGA for Custom Computing Machines, IEEE Computer Society Press, 1994, pp. 31–39.
[5] A. DeHon, DPGA-coupled microprocessor: commodity Ics for the early 21st century, in: Proceedings of the IEEE Workshop FPGA for Custom Computing Machines, Napa, CA, 1993, pp. 43–48.
[6] J.G. Eldredge, B.L. Hutchings, Density enhancement of a neural network using FPGAs and run-time reconfiguration, in: D.A. Buell, K.L. Pocek, (Eds.), Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April 1994, pp. 180–188.
[7] R.K. Gupta, Co-synthesis of Hardware and Software for Digital Embedded Systems, Kluwer Academic, Dordrecht, 1995.
[8] J.D. Hadley, B.L. Hutchings, Design Methodologies for Partially Reconfigured Systems, T. R., Department of Electrical and Computer Engineering, Brigham Young University.
[9] S. Hauck, A. Agarwal, Software Technologies for Reconfigurable Systems, Northwestern University, Department of ECE, Technical report, 1996.
[10] S. Hauck, The roles of FPGAs in reprogrammable systems, Proceedings of the IEEE 86 (4) (1998) 615–638.
[11] R.D. Hudson, D.I. Lehn, P.M. Athanas, A run-time reconfigurable engine for image interpolation, IEEE Symposium on FPGAs for Custom Computing Machines, 1998.
[12] B.L. Hutchings, M.J. Wirthlin, Implementation approaches for reconfigurable logic applications, in: Proceedings of Fifth International workshop on Field-Programmable Logic and Application, 1995, pp. 419–428.
[13] B.L. Hutchings, Exploiting reconfigurability through domain-specific systems, in: Proceedings of Seventh International Workshop, FPL'97, London, UK, 1997, pp. 193–202
[14] R. Jain, A. Sharma, H. Wang, Empirical evaluation of some high-level synthesis scheduling heuristics, in: Proceedings of the 28th ACM/IEEE Design Automation Conference, June 1991, pp. 210–215.
[15] M. Karthikeya, P. Gajjala, B. Dinesh, Temporal partitioning and scheduling data flow graphs for reconfigurable computer, IEEE Transactions on Computers 48 (6) (1999) 579–590.
[16] M. Kaul, R. Vemuri, Temporal partitioning combined with design space exploration for latency minimization of run-time reconfigured designs: Proceedings of the Design, Automation and Test in Europe Conference, 1999, pp. 202–209.
[17] M. Kaul, R. Vemuri, Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures, Design, Automation, and Test in Europe Conference, 1998, pp. 389–396.
[18] S. Kung, VLSI Array Processors, Prentice-Hall, USA, 1988.
[19] E. Lemoine, D. Merceron, Run time reconfiguration of FPGAs for scanning genetic databases, in Proceedings FCCM95, 1995, pp. 85–89.
[20] Y.L. Lin, Recent developments in high-level synthesis, ACM Transactions on Design of Electronic System 2 (1) (1997) 2–21.
[21] T.W. Luk, I. Page, Hardware–software codesign of multidimensional programs, in: Proceedings. FCCM'94, 1994, pp. 82–90.
[22] W. Luk, S. Guo, N. Shirazi, N. Zhang, Field-Programmable Logic, Smart Applications, New Paradigms and Compilers, LNCS 1142, Springer, Germany, 1996 (pp. 24–33).
[23] W. Luk, N. Shirazi, P.Y.K. Cheung, Modeling and optimizing run-time reconfigurable system, in: Proceedings of the FCCM96, IEEE Computer Society Press, 1996, pp. 167–176
[24] P. Lysaght, J. Stockwood, A simulation tool for dynamically reconfigurable field programmable gate arrays, IEEE Transactions on VLSI 4 (3) (1996) 381–390.
[25] P. Lysaght, J. Dunlop, Dynamic reconfiguration of FPGAs, in: W. Moore, W. Luk, (Eds.), More FPGAs: Proceedings of the International Workshop on Field-programmable Logic and Applications, 1993, pp. 82–94.
[26] P. Lysaght, G.M. Gregor, Controller synthesis for dynamically reconfigurable systems, in: IEEE Colloquium on Hardware-Software Cosynthesis for Reconfigurable Systems, UK, February 1996.
[27] P. Lysaght, H. Dick, G. McGregor, D. McConnell, J. Stockwood, in: Prototyping environment for dynamically reconfigurable logic, in: Proceedings of Fifth International Workshop on Field-Programmable Logic and Applications, 1995, pp. 409–418.
[28] P. Lysaght, J. Stockwood, J. Law, D. Grima, Artificial neural network implementation on fine-grained FPGA, in: Proceedings of Fourth International Workshop on FPGA and Applications, 1994, pp. 421–431.
[29] D. Micheli, Synthesis and Optimisation of Digital Circuits, McGraw-Hill, New York, 1994.
[30] J.V. Oldfield, R.C. Dorf, Field-Programmable Gate Arrays: Reconfigurable Logic for Rapid Prototyping and Implementation of Digital systems, Wiley, London, New York, 1995.
[31] I. Ouaiss, An integrated partitioning and synthesis system for

dynamically reconfigurable multi-FPGA architectures, in: Fifth Reconfigurable Architectures Workshop (RAW'98), 1998,http://xputers.informatik.uni-kl.de/RAW/RAW98/adv_prg_RAW98.html.

[32] I. Ouaiss, S. Govindarajan, V. Srinivasan, R. Vemuri, A unified specification model of concurrency and coordination for synthesis from VHDL, in: Proceedings of International Conference on Information Systems Analysis and Synthesis (ISAS'9), 1998,http://www.ececs.uc.edu/∼ddel.

[33] K. Rath, J. Li, Synthesizing reconfigurable sequential machines using tabular models, in: Fifth Reconfigurable Architectures Workshop (RAW'98), 1998.

[34] A.W. Robert, R. Camposano, A Survey of High-level Synthesis Systems, Kluwer Academic, Dordrecht, 1994.

[35] O.V. Ross, M. Turner, An FPGA-based hardware accelerator for image processing, in: W. Moore, W. Luk, (Eds.), More FPGAs, 1994, pp. 299–306.

[36] E. Sanchez, M. Sipper, J.O. Haenni, P.U. Andres, Static and dynamic configurable systems, IEEE Transactions on Computers 48 (6) (1999) 556–564.

[37] J. Spillane, H. Owen, Temporal partitioning for partially-reconfigurable-field-programmable gate, in: Proceedings in Reconfigurable Architectures Workshop in IPPS/SPDP'98, 1998.

[38] U. Steinhausen, R. Camposano, System synthesis using hardware/software codesign, in: Proceedings of International Workshop on hardware–Software Co-Design, 1993.

[39] S.M. Trimberger, Field-Programmable Gate Array Technology, Kluwer Academic, Dordrecht, 1994.

[40] M. Vasilko, D. Ait-Boudaoud, Architectural synthesis techniques for dynamically reconfigurable logic, Lecture Notes in Computer Science 1142, pp. 290–296.

[41] M. Vasilko, D. Ait-Boudaoud, Scheduling for dynamically reconfigurable FPGAs, in: Proceedings of International Workshop on Logic and Architecture Synthesis, 1995, pp. 328–336.

[42] S. Vincentelli, Some considerations of field programmable gate arrays and their impact on system design, in: Proceedings of the Second International workshop on Field Programmable Logic, Vienna, Austria, September 1992, pp. 26–34.

[43] M.J. Wirthlin, Improving Functional Density Through Run-time Circuit Reconfiguration, PhD thesis, Department of Electrical and Computer Engineering, Brigham Young University, 1997.

[44] M.J. Wirthlin, B.L. Hutchings, Improving functional density through run-time constant propagation, in: Proceedings of ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 1997, pp. 86–92.

[45] M.J. Wirthlin, B.L. Hutchings, Sequencing run-time reconfigured hardware with software, in: ACM/SIGD International Symposium on Field Programmable Gate Arrays, 1996.

[46] Xilinx Inc,http://www.xilinx.com/products/virtex.htm.

[47] Xilinx Inc: Virtex Series FPGAs—Redefining the FPGA, http://www.xilinx.com/products/virtex.htm, 1999.