POLITECNICO DI MILANO
DEIB
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

C-BASED HIGH LEVEL SYNTHESIS OF PARALLEL APPLICATIONS TARGETING ADAPTIVE HARDWARE COMPONENTS

Doctoral Dissertation of:
Vito Giovanni Castellana

Supervisor:
Prof. Fabrizio Ferrandi

Tutor:
Prof. Donatella Sciuto

The Chair of the Doctoral Program:
Prof. Carlo Ettore Fiorini

2013 - XXVI

To my family

Acknowledgments

Hey I know it's late, but we can make it if we run. The boss wrote these words, which more or less summarize my last few years. This is what I did: I ran. And this is what a PhD student is supposed to do, no matter how late it is, no matter how far the objective, you have to move forward trying to reach it. But sometimes, even for just a few seconds, it's good to take a look behind. If I do that, I realize how far these three years took me. The journey has been amazing though. So I just want to say thank you to all the traveling companion who made this journey so special, and who supported me while trying to reach my goals.

Il mio primo ringraziamento va ad Antonino. A tuo modo hai saputo amplificare la mia voglia di fare, di non arrendermi, e di raggiungere gli obiettivi, anche quelli più improbabili. E sei stato oltre che un buon mentore, un grande amico.
Grazie a tutti gli amici e parenti, per essere sempre riusciti a farmi avvertire la vostra prensenza e il vostro affetto.
Grazie a tutti coloro che mi hanno aiutato a sentirmi un pò a casa, anche quando a migliaia di chilometri di distanza.
Grazie a Fabrizio. Grazie per avermi insegnato a domare le idee, e solo dopo a seguirle. Grazie per aver reso più acuto e critico il mio giudizio, e più profonda e ampia la mia preparazione. Per avermi bacchettato quando ho sbagliato, e per avermi dato fiducia. Grazie per avermi trasmesso la tua passione per quel che facciamo.
Grazie a Marco, molto più che un semplice amico. Hai condiviso con me sogni e passioni, gioie e delusioni, mi hai offerto il tuo supporto quando ne

ho avuto bisogno, sei stato la chitarra che ha supportato la mia voce. E non mi riferisco alle (tante) volte in cui ci siamo improvvisati musicisti..
Grazie a Michele, il miglior fratello che si possa desiderare. Grazie per la tua discreta attenzione, per il tuo affetto, per aver saputo spesso essere anche amico e complice.
E infine, il GRAZIE più grande va ai miei genitori. Come non dirò mai abbastanza, ogni mio successo è prima di tutto il vostro. Senza il vostro amore, il vostro supporto e la vostra stima, ogni piccolo traguardo che ho raggiunto sarebbe ancora lontano.
Thank you.

Abstract

THE EVER INCREASING COMPLEXITY of embedded systems is driving design methodologies towards the use of abstractions higher than the Register Transfer Level (RTL). In this scenario, High Level Synthesis (HLS) plays a significant role by enabling the automatic generation of custom hardware accelerators starting from high level descriptions (e.g., C code). Typical HLS tools exploit parallelism mostly at the Instruction Level (ILP). They statically schedule the input specifications, and build centralized Finite Stat Machine (FSM) controllers. However, the majority of applications have limited ILP and, usually, centralized approaches do not efficiently exploit coarser granularities, because FSMs are inherently serial. Novel HLS approaches are now looking at exploiting coarser parallelism, such as Task Level Parallelism (TLP). Early works in this direction adopted particular specification languages such as Petri nets or process networks, reducing their applicability and effectiveness in HLS. This work presents novel HLS methodologies for the efficient synthesis of C-based parallel specifications. In order to overcome the limitations of the FSM model, a parallel controller design is proposed, which allows multiple flows to run concurrently, and offers natural support for variable latency operations, such as memory accesses. The adaptive controller is composed of a set of interacting modules that independently manage the execution of an operation or a task. These modules check dependencies and resource constraints at runtime, allowing as soon as possible execution without the need of a static scheduling. The absence of a statically determined execution order has required the definition of novel synthesis algorithms, since most of the common HLS techniques require the definition of an operation schedule. The proposed algorithms have allowed the design and actual implementation of a complete HLS framework. The flow features automatic parallelism identification and exploitation, at different granularities. An analysis step, interfacing with a software compiler, processes the input specification and identifies concurrent operations or tasks. Their parallel execution is then enabled by the parallel controller architecture. Experimental results confirm the potentiality of the approach, reporting encouraging performance improvements against typical techniques, on a set of common HLS benchmarks. Nevertheless, the interaction with software compilers, while profitable for the optimization of the input code, may represent a limitation in parallelism exploitation: compilation techniques are often over-conservative, and in the presence of memory operations accessing shared resources, force serialization. To overcome this issue, this work considers the adoption of parallel programming paradigms, based on the insertion of pragma annotations in the source code. Annotations, such as OpenMP pragmas, directly expose TLP, and enable a more accurate dependences analysis. However, also in these settings, the concurrent access to shared memories among tasks, common in parallel applications, still represents a bottleneck for performance improvement. In addition, it requires concurrency and synchronization management to ensure correct execution. This work deals with such challenges through the definition of efficient memory controllers, which support distributed and multi-ported memories and allow concurrent access to the memory resources while managing concurrency and synchronization. Concurrency is managed avoiding, at runtime, multiple operations to target the same memory location. Synchronization is managed providing support for atomic memory operations, commonly adopted in parallel programming. These techniques have been evaluated on several parallel applications, instrumented through OpenMP pragmas, demonstrating their effectiveness: experimental results show valuable speed-ups, often close to linearity with respect to the degree of available parallelism.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Since the inception of information technology, the synthesis of digital circuits has been a main concern for researchers and scientific community. Over the years, the proposed design methodologies have been forced to move to higher abstraction levels due to the constant improvements in silicon technology and to the increasing complexity of applications and architectures. Such factors, since early '70s, made more inadequate lower abstraction level methodologies such as Logic-level or Physic-level synthesis, and led to their overcome in favor of High-Level Synthesis (HLS). Over the years HLS has been able to capture and renew the interests of the research community, even if in the past its adoption in industry has often resulted in failures. However current systems complexity is quickly turning HLS from an undeniably promising idea to an actual need. For example, in [186] the authors report a study from NEC showing that a 1M-gate design typically requires about 300k lines of RTL code, which are difficult to be handled by a human designer. Adopting high level description languages, such as C/C++, the code density dramatically reduces (up to 10X): behavioral languages can describe 1M-gate designs in 30-40k lines of code. Other reviews reported in literature indicate that working at a higher level of the design hierarchy using high-level synthesis reduces the amount of code that must be developed by as much as two thirds [128]. Constant improvements in HLS have finally led to the (at least partial) automation of the design process, significantly decreasing the development cost. Furthermore, the error rate is reduced by the presence of a proper verification phase. While writing behavioral code is inherently simpler than writing RTL code, separating the design intent from the physical implementation avoids the tedious process of rewriting and retesting code to make architectural changes. This also facilitates the design space exploration process since a good synthesis system produces several RTL implementations for the same specification in a reasonable amount of time, allowing the developers to consider different solutions and trade-offs. Generally, it is reported an overall reduction of the design effort, with respect to lower level methodologies, of 50% or more [128]. This characteristics shorten the design cycle, thus increasing the chance for companies of hitting the market window. Almost four decades of research have resulted in impressive improvements in HLS methodologies: modern frameworks support C-like programs as input specification, providing a wide language coverage; they integrate engines for quick and efficient design space exploration; they are able to exploit modern reconfigurable platforms, and directly interface with the logic synthesis tools; finally, and most important of all, they provide good Quality of Results (QoR), especially for specific domains. However there are still several aspects which need further investigation, and several opportunities for improvement, which motivate the work described by this thesis. For example, most of the proposed design methodologies targeted the same architectural model as a result of synthesis, i.e. the Finite State Machine with Datapath (FSMD) model. FSMDs are composed by a datapath, which includes the hardware resources which *perform* the computation, and a FSM controller, which *manages* the computation. As detailed in the next chapters, FSMDs are inherently serial: they model execution as a sequence of control steps, thus limiting parallelism exploitation within a single execution flow. Architectural alternatives exist, but they have not been comprehensively explored yet. This work investigates one of such alternatives, i.e. parallel controllers. Parallel controllers are able to manage multiple execution flows concurrently, thus overcoming one of the main limitation of the FSM-based paradigms. A parallel controller design is proposed, suitable for HLS. Such model allows the synthesis of adaptive accelerators featuring dynamic scheduling. Typical FSMDs are built reflecting a statically computed schedule; in the proposed architecture instead, execution is managed through a set of communicating elements, which establishes directly at runtime if an operation can be executed. The absence of a pre-computed execution ordering has required the definition of novel algorithms for most of the steps composing the synthesis process. Typical approaches in fact, are based on the definition of a static schedule, thus resulting not applicable. The designed synthesis flow has been automated through the definition and actual implementation of a complete C-based HLS tool. Compared with existing approaches, the generated adaptive components provide unique benefits. Among them, the most relevant are the efficient management of variable latency operations, poorly supported by statically scheduled FSMDs, and the support for coarse grained parallelism exploitation. Coarse grained parallelism mostly occurs in the form of Task Level Parallelism (TLP): however, exploiting and identifying TLP introduce several challenges. TLP exploitation is a complex problem: usually tasks share memory resources, and if they are allowed to execute at the same time, concurrency and synchronization between them must be managed. TLP identification, instead, is difficult because most of the adopted specification languages are programming languages such as C/C++, which have been designed for serial execution. This work has considered both these aspects. The first one has been addressed through the definition of a Memory Controller Interface (MIC). The MIC is a hardware component which allows fine grained parallelism exploitation on memory accesses, automatically manages concurrency on shared resources, and supports atomic memory operations for synchronization. It has been designed as a custom, parameterizable IP, suitable for easy adoption in both RTL and HLS flows. The second challenge has been considered introducing in the designed flow, support for parallel programming APIs. This allow the input specification to directly expose parallelism, and to take advantage of atomic memory operations, supported by the MIC, for tasks synchronization. The resulting HLS flow is then able to efficiently exploit TLP, identified through pragma annotations in the input code. According to the particular characteristics of the specification, it is also able to automatically adopt different synthesis approaches: functions characterized by ILP only are synthesized with FSM-based techniques, while code featuring TLP is synthesized targeting the proposed adaptive architectures.

1.1 Main Contributions

The main contributions of this work may be summarized as follows:

- the definition of a parallel controller design, suitable for the HLS of adaptive hardware components featuring dynamic scheduling: execution is managed through a set of components, interacting through a lightweight token-based communication schema. Each element establish directly at runtime when an operation can execute;

- the design of novel HLS algorithms for the automated synthesis of the proposed architecture;
- the development of a complete C-based HLS tool, targeting the adaptive accelerator design;
- the design of hardware components for managing concurrency and synchronization on shared memories, thus providing support for efficient TLP exploitation;
- the support for parallel programming APIs, which allow efficient identification of TLP;
- the definition of a modular architecture which couples statically and dynamically scheduled components, to best adapt to the characteristics of the input specification;
- the experimental evaluation of each of the techniques proposed in this thesis, comparing them to existing approaches.

1.2 Dissemination of Results

Techniques and methodologies described in this thesis work have been presented at several international conferences, leading to the publication of the following papers:

1. C. Pilato, V.G. Castellana, S.Lovergine, F. Ferrandi, *A Runtime Adaptive Controller for Supporting Hardware Components with Variable Latency*, in Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2011), San Diego, California, USA, June 2011
2. V.G. Castellana and F. Ferrandi, *Speeding-Up Memory Intensive Applications through Adaptive Hardware Accelerators*, in Proceedings of High Performance Computing, Networking, Storage and Analysis, 2012 SuperComputing Companion (SCC), Salt Lake City, Utah, USA, November 2012
3. V.G. Castellana and F. Ferrandi, *Scheduling Independent Liveness Analysis for Register Binding in High Level Synthesis*, in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013, Grenoble, France, March 2013

4. V.G. Castellana and F. Ferrandi, *Applications Acceleration Through Adaptive Hardware Components*, in Proceedings of Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Boston, Massachussets, USA, May 2013

5. V.G. Castellana and F. Ferrandi, *An Automated Flow for the High Level Synthesis of Coarse Grained Parallel Applications*, in Proceedings of International Conference on Field-Programmable Technology (ICFPT) 2013, Kyoto, Japan, December 2013

6. V.G. Castellana, A. Tumeo, F. Ferrandi, *An Adaptive Memory Interface Controller for Improving Bandwidth Utilization of Hybrid and Reconfigurable Systems*, in Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, Dresden, Germany, March 2014

7. V.G. Castellana, A. Tumeo, F. Ferrandi, *High Level Synthesis of Memory Bound and Irregular Parallel Applications with Bambu*, International Workshop on Electronic System-Level Design towards Heterogeneous Computing, 2014, Dresden, Germany, March 2014

Informal proceedings and other dissemination activities:

- V.G. Castellana, C. Pilato, F. Ferrandi, *Accelerating Embedded Systems with C-Based Hardware Synthesis*, HiPEAC ACACES-2011, Fiuggi, Italy, July 2011 [Meeting proceedings, ISBN:978 90 382 17987]

- V.G. Castellana, A. Tumeo, F. Ferrandi, *A Synthesis Approach for Mapping Irregular Applications on Reconfigurable Architectures*, High Performance Computing, Networking, Storage and Analysis, 2013 SuperComputing (SC), Denver, Colorado, USA, November 2013 [Technical Program poster, extended abstract available online]

- V.G. Castellana, F. Ferrandi, *C-Based High Level Synthesis of Adaptive Hardware Components*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, Dresden, Germany, March 2014 [PhD forum presentation]

1.3 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 provides a background for this work, introducing basic concepts and definitions. It also proposes Bambu as an example of modern HLS tools. The methodologies presented in this thesis have been developed within the Bambu framework, and its release version, freely available on-line, has represented a baseline for evaluation. Chapter 3 illustrates related work, tracking the chronological evolution of HLS, and identifying which aspects need further improvements, thus highlighting the contribution of this work to the state of the art. Chapter 4 introduces the proposed parallel controller architecture, describing all the designed components, and showing its adaptive behavior and main offered features through some examples. Chapter 5 describes the complete HLS flow for the automatic generation of the proposed architecture, detailing each of the algorithm composing the synthesis process. Chapter 6 presents the Memory Controller Interface, designed to manage concurrency and synchronization for memory intensive specifications. Chapter 7 focuses on TLP identification and exploitation, proposing refinements and improvements to the HLS flow, enabling efficient support of parallel applications instrumented through pragma annotations. Chapter 8 concludes this thesis, identifying opportunities for further improvements and promising future research directions.

CHAPTER 2

Background

This thesis work proposes novel adaptive architecture designs for the synthesis of hardware accelerators, together with algorithms for their automatic generation through a High Level Synthesis (HLS) flow. This chapter introduces some preliminary concepts and definitions, useful to briefly build a background for this work. As a representative of modern synthesis tools, the *Bambu* framework is proposed. Bambu is a state of the art HLS tool, producing Verilog implementations of behavioral specifications described through C-code. Methodologies and techniques described in this work, have been implemented customizing and extending this framework, and its release version, freely available on the Internet, has represented a baseline for evaluation. The remainder of this chapter is organized as follows: Section 2.1 introduces the High-Level Synthesis process; Section 2.2 describes the Finite State Machine with Data-path model, which is the most widespread design solution adopted in HLS. Section 2.3 formally characterizes a typical HLS flow, which can be coarsely partitioned in three main stages: the Front-end phase (2.3.1), the Synthesis phase (2.3.2), the Back-end phase (2.3.3). Section 2.4 describes Bambu, which represents an example of typical HLS tools targeting FSMDs architectures; Section 2.5 concludes the chapter.

2.1 Introduction to High-Level Synthesis

High-Level Synthesis (HLS), also known as *behavioral synthesis* or *algorithmic synthesis*, is a design process that, given an abstract behavioral specification of a digital system and a set of constraints, automatically generates a Register-Transfer Level (RTL) structure that implements the desired behavior [127]. In Table 2.1, the typical HLS *inputs* and *outputs* are shown; their functionalities will be presented in the following.

INPUT	*OUTPUT*
- Behavioral Specification - Design Constraints - Optimization Function - Resource Library	- RTL Implementation of the Input Behavior (Controller+Data-path)

Table 2.1: *Typical HLS inputs and outputs*

Inputs:

- The *behavioral specification* consists in an untimed or partially timed algorithmic description in high-level language (such as C language), that is transformed in a fully timed and bit-accurate RTL implementation by the *behavioral synthesis flow*.
- The *design constraints* represent targets that must be met for the design process to be considered successful. These include timing, area and power constraints. Timing constraints usually identify a target operating frequency for the final design, ruling all the synthesis process. For example, design characteristics that may alter the timing performance are the structure of steering logic interconnections and the number of stages of pipelined components, implementing high latency operations (e.g. division and multiplication). Area constraints may be expressed specifying resource availability at different granularities, e.g. indicating an upper bound for the number of logic cells to be used, or the maximum number of instances which may be allocated for each resource type. Power constraints set a power consumption threshold for the final design.
- The *optimization function* is a cost function whose argument represents the design target to optimize. As for the design constraint, the most common features desired to be maximized/minimized are execution time, area and power consumption. Clearly the optimization function, as it generally happens, can depend on two or more variables. In

such case, it is needed to manage a multi-objective optimization process, where a global optimum solution could not exists at all. Instead, a set of designs, all satisfying the constraints, for which is not possible to establish who is better, can coexist. These solutions are defined as *Pareto optimal*. Given a set S of feasible solutions, all functions of n parameters, s∈S is *Pareto optimal* if there not exists another solution s'∈S that improves one or more parameters without worsen at least another one.

- The *resource library* contains a collection of modules from which the synthesizer must select the best alternatives matching the design constraints and optimizing the cost function.

Output: it is a register transfer level description of the designed architecture, usually consisting of

- a *data-path*, i.e. the entity which *performs* the computation between primary inputs, which provide the data to be elaborated, and primary outputs, which return the results of computation.
- a *controller* i.e. the entity which *manages* the computation, handling the data flow in the data path by setting control signals values, e.g. Functional Units, registers and muxes selectors (see Figure 2.1). Controller inputs may come from primary inputs (control inputs) or from data path components (status signals as result of comparisons). It determines which operations have to be executed at each control step and the corresponding paths to be activated inside the data-path.

Different controller implementations approaches are feasible; however, in HLS the controller is usually designed as a centralized Finite State Machine (FSM). The resulting architectural model, detailed in the following section, is known as FSM with Data-path (FSMD).

2.2 The Finite State Machine with Data-Path Model

The most common architectural model in high level synthesis is the finite state machine with data-path, as shown in Figure 2.1.

2.2.1 Data-Path

The data-path includes a set of hardware resources, i.e. storage, functional and interconnection units, and defines how those modules are connected each other [171]. All the RTL components can be allocated in different

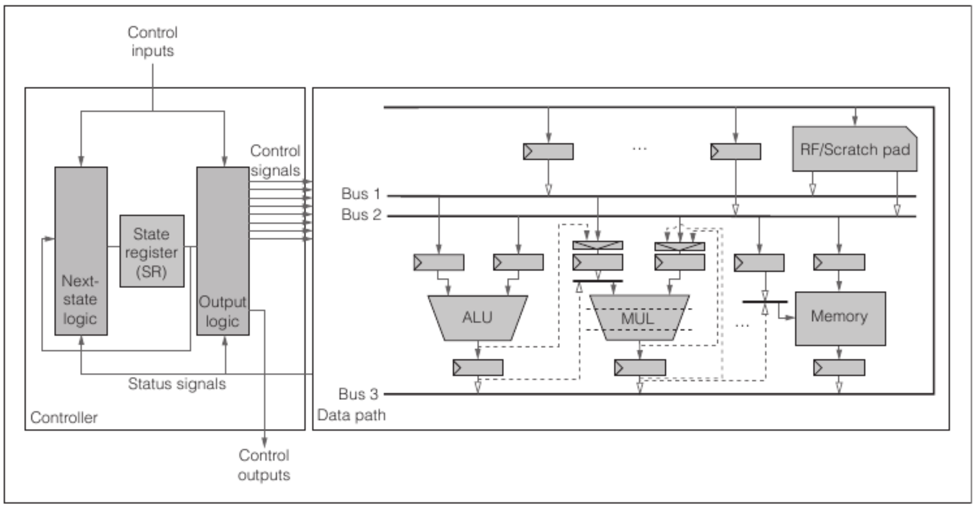


Figure 2.1: *Typical Architecture composed of a Finite State Machine (FSM) and a Datapath.*

quantities and types, and can be customly connected at design time through different interconnection schemes, e.g. mux or bus based. Different architectural solutions could be adopted, allowing optimizations such as:

- multicycling: if each instruction requires exactly one clock cycle, then the clock cycle is lower-bounded by the higher required execution time; to overcome this issue, expensive instructions in terms of delay are executed through subsequent clock cycles;
- chaining: it is another solution to the previous problem; instead of reducing the clock cycle, instructions requiring less time are executed subsequentially in the same clock cycle;
- pipelining: instructions are divided in stages, and the clock cycle set to the time required to execute the slower one; if stages are obtained in a way such that there is no concurrency on the resources that execute them, than different stages of different instructions may be executed in the same clock cycle.

Formally, a *data path* DP can be described as a graph $DP(M, T, I)$, where

- $M = M_o \cup M_s \cup M_i$ is the set of nodes, corresponding to the DP modules, i.e. instances of library components, where
  - M_o is the set of functional units such as adders, ALUs and shifters;

  - M_s is the set of storage elements, as registers, register files and memories;
  - M_i is the set of interconnect elements such as tristates, buses and multiplexers;

- $I \subseteq M \times M$ is the set of graph's edges, i.e. interconnection links.

2.2.2 Finite State Machine

The Finite State Machine (FSM) represent one of the most common models applied in architectural synthesis. Even though they can describe different kinds of sequential machines, FSMs are typically used for synchronous ones. *Synchronous machines* are characterized by the presence of an impulsive signal, i.e. the clock, propagated over the whole circuit, that determines the moment in which the inputs must be evaluated to possibly cause the transition from one state to another of the FSM. Hence, the order in which the inputs are received does not affect the execution, provided they come within the clock cycle. Instead, in the case of *asynchronous machines*, a global temporization does not exists, and an explicit communication protocol is required to ensure the computation correctness. The asynchronous machines can be classified in two main categories: *level machines*, in which the system state transitions are caused by changes in the level of the input variables, and *impulsive machines*, in which the presence or absence of impulses causes such transitions.
Formally, a *finite state machine* is defined as the tuple $M = (I, O, S, S_0, R)$, where

- I represents the input alphabet,
- O denotes the output alphabet,
- S denotes the set of states,
- S_0 denotes the initial state,
- R denotes the global relation.

The *global relation R* is defined as $R \subseteq S \times I \times S \times O \rightarrow \{0, 1\}$ such that $R(i, u, s, t) = 1$ iff given as input $i = (i_1, i_2, ..., i_n) \in I$, M goes from the current state $s = s_1, s_2, ..., s_k \in S$ to the next state $t = t_1, t_2, ..., t_k \in S$ producing as output $o = o_1, o_2, ..., o_k \in O$
The main FSM controller components are:

- a state register (SR), that stores the current state of the FSM model describing the controller's operation;

- the next state logic, that computes the next state to be loaded in the SR;
- the output logic, that generates the control signals.

State-Transition Graph

A finite state machine M can be represented by its corresponding *state-transition graph* $G(S, E)$ where

- nodes $s \in S$ are the states of M,
- edges $e \in E \subseteq S \times S$ denote transitions between states.

State-Transition Relation

Given the global relation R and an input i,the *state-transition relation* determines the relationship between current state and next state. It is defined as $\Delta(i, s, t) = \exists o R(i, o, s, t)$. The *state-transition function* is usually denoted by $\delta(i, s, t)$.

Output Relation

Given the global relation of M, the initial state S_0 and an input i, the *output relation* gives the output value of M. It is defined as $\Lambda(i, o, s) = \exists t R(i, o, s, t)$. The *output function* is usually denoted by $\lambda(i, o, s)$.
Starting from the above definitions, it is possible to classify the different types of FSMs in three classes:

- Autonomous - The input alphabet is the empty set (e.g. counters).
- State based - Also known as Moore's machines, their output relation Λ depends only on the current state.
- Transition based - Also known as Mealy's machines, their output relation Λ depends on input values also.

The classical logic implementation of a FSM stores the states in storage elements (registers) while state-transition and output functions are synthesized in combinational logic. Several encodings exist for identifying states in a FSM; the most widespread are logarithmic (binary) and *one-hot* encodings. In binary encoding, the different states are represented through variables: if the number of states is N, the FSM implementation requires $log_2(N)$ 1-bit registers However, this encoding requires combinational logic for mapping variables values to states. One hot encoding instead demands more sequential resources, i.e. N 1-bit registers for representing N states. Furthermore, one hot implementations are faster than binary implementations, since the latter require decoding logic.

Now the *Finite State Machine with Data Path* can be defined as the tuple $< S, I \cup B, O \cup A, \Delta, \Lambda >$, where:

- S, Δ, Λ denote the same sets defined for a FSM;
- $I \cup B$ denotes the input language; it is an extension of the set I defined for the FSM, in order to include some of the state variables $b \in B \subseteq Stat$;
- $O \cup A$ represents the output language, including some assignments $a \in A \subseteq Asg$

FSMDs can be adopted both to describe a design at RT level, or, at higher levels of abstraction, to represent a producer/consumer process where inputs are consumed and outputs are produced. Complex systems could be viewed as processes compositions, where each process is modeled as FSMDs and communicates with the other ones. Communication is intent to be between control units, between data paths and between control unit and data path. The number of signals and the temporal relations between them during communication define a protocol, e.g. *request-acknowledge handshaking* protocol [13].

2.3 Typical High Level Synthesis Flows

High-level synthesis is typically composed of different tasks, summarized in Figure 2.2. There exists many approaches in literature that perform these activities in different orders, using different algorithms. In some cases, several tasks can be combined together or performed iteratively to reach the desired solution. Nevertheless, the HLS flow steps can be grouped in three main macro-tasks:

- Front End:
  performs *lexical processing*, *algorithm optimization* and *control/data-flow analysis* in order to build and optimize an Internal Representation (IR) to be used in the subsequent steps. Internal representations describe the specification highlighting specific properties, on which a given task of the synthesis phase will focus on.
- Synthesis:
  in this phase the design decisions are taken, to obtain a RTL description of the target architecture that satisfies the design constraints. The

number and type of hardware modules is established and each instruction is scheduled and assigned to one of the available resources that can execute it.

- Back End:
  the resulting netlist is generated and reproduced in a hardware description language.

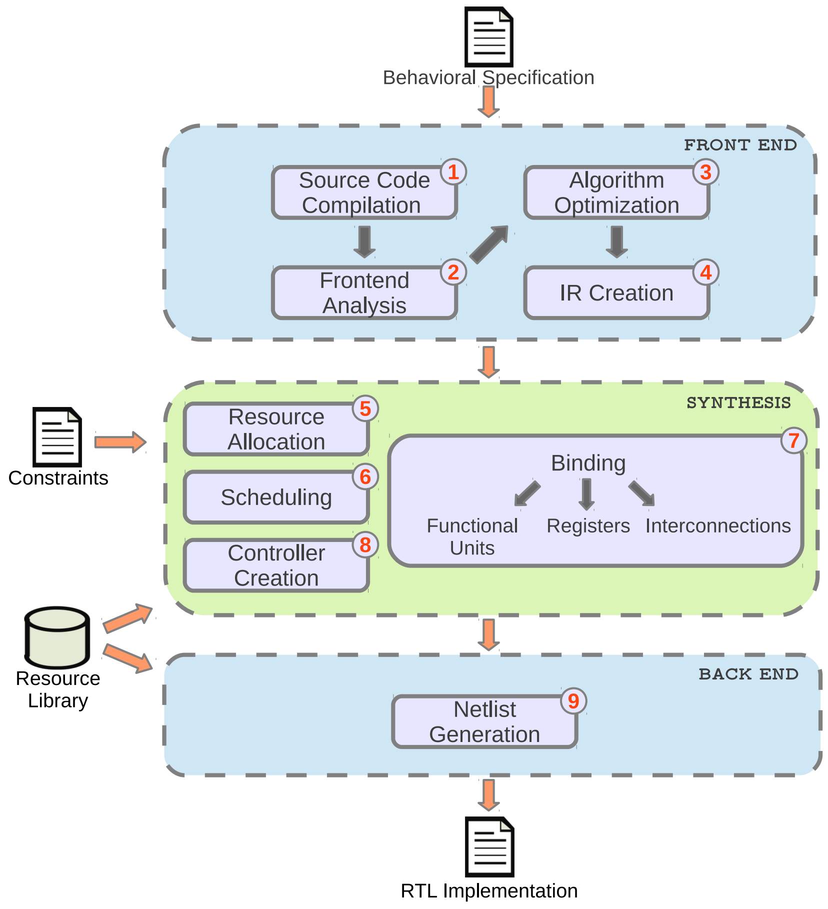


Figure 2.2: *Typical HLS flow.*

Next sections describe each task of the typical HLS-flow.

2.3.1 Front End

An Intermediate Representation (IR) can be described as an interface between a source language and a target language. Such notation should describe source code properties independently with respect to the source/target languages details. Given as input a behavioral specification, the front end translates it in a proper IR, performing lexical processing, algorithm optimization and control/data-flow analysis.

Compilation

The first step in the HLS flow consists in the source code parsing, that translates the high level specification into an IR, as common in conventional high level language compilation, to abstract from language details. The resulting IR is usually a proper graph representation of the parsed code, and can be optimized or transformed producing several additional representations of the same specification.

Front-end Analysis and Algorithm Optimization

The previous step provides a formal model that exhibits data and control dependencies between the operations. Such dependencies are generally represented in a graph notation, for example in the form of Control Data Flow Graphs. The control/data-flow analysis characterizes some properties of the specification, identifying for example its loop and call structure, or partioning it in Basic Blocks. The analysis step enables further optimizations to better exploit the available parallelism. These optimizations are the same ones widely used in optimizing and parallelizing compilers [19], such as dead code elimination and constant propagation, or translation in Static Single Assignment (SSA) form.

Internal Representations Creation

The optimized code is finally reproduced in the form of IR(s) which will be used in the synthesis steps. Common IRs include Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), Control Data Flow Graphs (CDFGs), and Program Dependencies Graphs (PDGs). Flow graphs may represent the input specification at different granularities, for example at the instruction level or at the Basic Block level.

2.3.2 Synthesis

The HLS core consists of different steps strictly connected each other. They can be summarized in:

- Scheduling,
- Resource Allocation,

- Resource Binding.

A desirable feature for HLS is to estimate as soon as possible timing and area overheads, so that later steps could optimize the design. A feasible approach is to start from one of the above mentioned tasks and consequently accomplish the other ones. Then the obtained results could be used to modify the solution of previous steps: in this way the final solution, optimized towards a performance metric, is built incrementally. Another possibility is to fulfill the tasks partially, and complete them as results of other steps are available. For example functional units could be allocated in a first time, and the interconnection allocation could be performed after the binding or scheduling tasks. Allocation, scheduling and binding are all NP-complete problems, thus solving them as a unique task makes the synthesis a too complex process to be applied to real-world specifications. The choice between different ordering possibilities is dictated from the design constraints and tool's objectives. For example, under resource constraints, allocation could be performed first and scheduling could try to minimize the design latency. Instead, in time constrained designs, allocation could be performed during the scheduling; the scheduling process, in this case, could try to minimize the circuit's area while meeting the timing constraints.

Scheduling

The scheduling task introduces the concept of time: according to the data and control dependencies extracted by the front end all the operations are assigned to specific control steps. Also the concept of parallelism is introduced: if dependences and resource availability allow it, more than one instruction can be scheduled in the same clock cycle. A common approach addressing the scheduling problem can be modeled as follows:

- input: DFG of the input specification;
- output: the scheduled DFG, i.e. a graph $G(V_0, E, C)$ where:
  - $v \in V_0$ are the nodes of the (C)DFG, i.e. the operations to be executed;
  - $e \in E$ are the edges of the (C)DFG, representing the data flow;
  - $c \in C$ are control steps.
- scheduling procedure: a function $\theta : V_0 \rightarrow \Pi(C)$ assigns to each node $v \in V_0$ a sequence of cycle steps, where $\Pi(C)$ is the power set of C, i.e. the set of all the subset of C.

In the presence of control constructs such as loops, the input specification is usually partitioned in Basic Blocks, which are sequences of code with a single entry point and a single exit point. The scheduling routine then applies to each Basic Block separately. As mentioned before, the scheduling process could be differently constrained, e.g. time or resource constrained.

For example, Figure 2.3 proposes the pseudo code of a simple specifica-

```
avg(a,b,c,d)
1: $t1 = a + b$
2: $t2 = c + d$
3: $t = t1 + t2$
4: $avg = t/4$
```

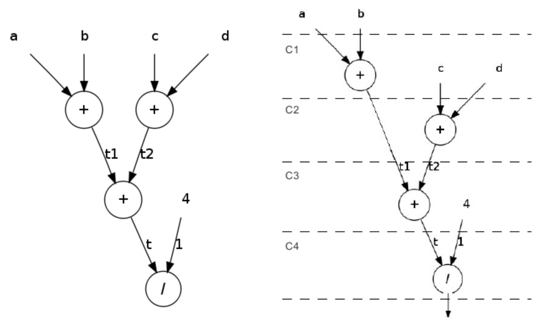


Figure 2.3: *Pseudo-code, DFG and scheduled DFG of a program that computes the average of 4 numbers.*

tion, togheter with its DFG and a possible scheduled DFG under resource constraints (1 adder and 1 divider available), assuming that each operation needs one control step to be executed.

Resources Allocation

A set of hardware resources is established to adequately implement the design, satisfying the design constraints. Allocation defines the number of instances and the type of different resources from the ones available in the resource library, which describes the relation between the operation types and the modules. Since different hardware resources have different characteristics, such as area, delay or power consumption, usually this informations is included in the resource library, to guide both the allocation and the other related tasks. A library $\Lambda(T, L)$ is characterized by:

- a set T of operation types;
- a set L of of library components (i.e. modules);

The library function $\lambda : T \rightarrow \Pi(L)$, where as usual $\Pi(L)$ denotes the power set of L, establishes which modules $l \in L$ could execute operations of type $t \in T$. On the other hand, the function $\lambda^{-1} : L \rightarrow \Pi(T)$ defines the *operation type set of l*, written $\lambda^{-1}(l)$, i.e. the subset of operation types

that a module $l \in L$ can execute. Given two operation of type $t_1, t_2 \in T$, if $t_1, t_2 \in \lambda^{-1}(l)$, then they can share module l. Moreover, $\lambda(t_1) \cap \lambda(t_2)$ describes the subset of modules that can be shared among operations of type t_1 and t_2.
Having defined the data-path as a graph $DP(M_o \cup M_s \cup M_i, I)$, thus allocation task must determine the components belonging to each module set M_k. This task defines the allocation functions for each module set.

Functional Units Allocation

The *module allocation function* $\mu : V_0 \rightarrow \Pi(M_0)$ determines which module performs a given operation. An allocation $\mu(v_i) = m_j, v_i \in V_0, m_j \in M_0$ is valid iff module m_j is an instance of $l_j \in \lambda(t_i)$, with t_i operation type of v_i, i.e. m_j can execute v_i.

Registers Allocation

Values produced in one clock cycle may be consumed in another one, and in this case such values must be stored in registers or in memory. Liveness analysis can allow different variables sharing a register, revealing if their life intervals overlaps or not, in order to reduce the number of registers, and the design overhead in terms of area. Even in this case, techniques developed in compiler theory are successfully applicable. In particular, after the scheduling, each edge that crosses a cycle step boundary represents a value that must be stored. Thus the scheduled DFG should be transformed to take in account such situation. Given a scheduled DFG $G(V_0, E, C)$, the *storage value insertion* is a transformation $G(V_0, E, C) \rightarrow G(V_0 \cup V_s, E', C)$ which adds nodes (storage values) $v \in V_s$ such that each edge $e \in E$ which traverse a cycle step boundary is connected to a storage value. The *register allocation function* could now be defined as $\psi : V_s \rightarrow \Pi(M_s)$; it identifies the storage modules holding a value from the set V_s.

Resources Binding

The allocation task defines the set M of modules that composes the data path. Each module $m \in M$ is an instance of a library component $l \in L$. Given a DFG $G(V_0, E)$, each operation $v \in V_0$ must be bound to a specific allocated module m. This task takes the name of module or resource binding. A resource binding is defined as a mapping $\beta : V_0 \rightarrow M_0 \times \mathbb{N}$; given an operation $v \in V_0$ with type $\tau(v) \in \lambda^{-1}(t), t \in L$, $\beta(v) = (t, r)$ denotes v will be executed by the component $t = \mu(v)$ and $r \leq \sigma(t)$, i.e. v is assigned to the r-th instance of resource type t. Module binding must ensure that the resource assigned to an operation is available in the cycle step in which the given instruction is scheduled. When the binding is performed before scheduling, the scheduling task will take care of schedule operations in order to avoid resource conflicts. Different approaches can be followed to perform the binding; in the simplest case β is a one-to-one mapping, associating each resource to one operation.

Interconnection Binding

The *interconnect binding function* is defined as $\iota : E \rightarrow \Pi(M_i)$, and describes how the allocated resources are connected and which interconnection is assigned for each data transfer. If a resource is shared among different operations, interconnections include steering logic, whose selectors are managed by the controller. Different solutions could differently affect the design in terms of delay, area occupancy or interconnections complexity.

Controller Synthesis

Once the data-path is built, it is possible to define the activation signals that the controller should generate to activate the data path modules. Controller synthesis can be performed following two main approaches:

- **Microprogrammed Controller** - each state of the FSM is coded as a microinstruction that specifies the data path activation signals and the next state; if the resulting microprogram is stored in a ROM, the next state can be represented by the ROM address of the next microinstructions (i.e. associated to the next state). If the CFG describing the specification is linear, i.e. there are not conditional nodes, the next state could be computed by a simple counter without indicating it in every microinstructions.

- **Hardwired Controller** - the controller is synthesized as a combinatory circuit and registers.

Both the models represent the abstract model of the synchronous FSM. The design complexity increases in the presence of hierarchical structures, or in control dominated specifications. Most HLS flows synthesize hardwired controllers.

2.3.3 Back-end

During the netlist generation phase, the final architecture design is written out as RTL code, usually described through a Hardware Description Language such as Verilog or VHDL. To facilitate this task, resource libraries usually include a HDL description of each available module.

2.4 Bambu: A Free Framework for the High-Level Synthesis of Complex Applications

As detailed in next chapter, a multitude of tools implementing the described typical HLS flow are available. Among them we consider as a representative the PandA open-source Framework [10]. PandA covers different as-

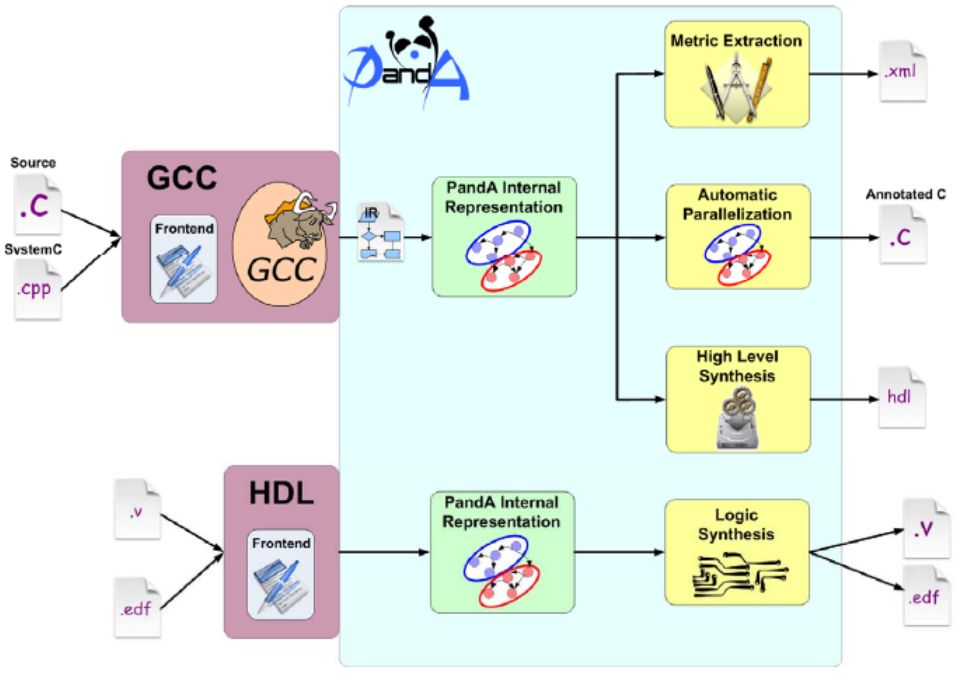


Figure 2.4: *Panda framework schematic overview.*

pects of the hardware/software co-design of embedded systems (Figure2.4), including methodologies to support:

- high-level synthesis of hardware systems;
- parallelism extraction for software and hardware/software partitioning;

- the definition of metrics for the analysis and mapping onto multiprocessor architectures and on dynamic reconfigurability design flow.

The framework offers a complete freely available HLS tool: Bambu [1]. Bambu receives as input a behavioral description of the algorithm, written in C language, and generates the Verilog description of the corresponding RTL implementation as output, along with a test-bench for the simulation and validation of the behavior. This HDL description is then compatible with commercial RTL synthesis tools. From the software design point of view, Bambu is extremely modular, implementing the different tasks of the HLS process, and specific algorithms, in distinct C++ classes. The overall flow acts on different IRs depending on the synthesis stage. The modularity of Bambu has allowed a complete integration of the methodologies and techniques described in this thesis, implementing only the algorithms required for the generation of the proposed architectures, while sharing the architecture independent components of the flow, such as the test-bench generation routines. Bambu assists the designer during the HLS of complex applications, aiming at supporting most of the C constructs (e.g., function calls and sharing of the modules, pointer arithmetic and dynamic resolution of memory accesses, accesses to array and structs, parameter passing either by reference or copy).

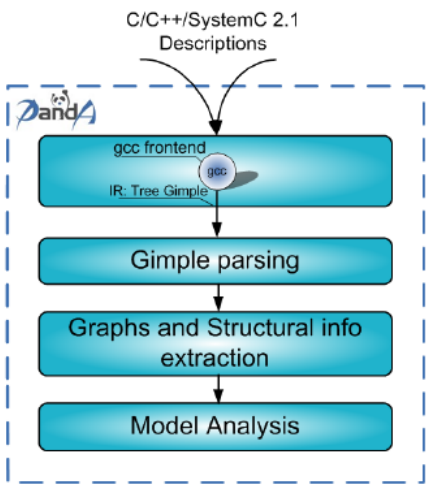


Figure 2.5: *PandA analysis flow.*

2.4.1 Front-end

Bambu has a compiler-based interface interacting with the GNU Compiler Collection (GCC) ver. 4.7 (Figure 2.5) and builds the internal representation in Static Single Assignment form of the initial C code. In particular, the source code is parsed, producing GENERIC trees: GENERIC is a language independent representation, which interfaces the parser and the code optimizer. The GENERIC code is then translated into the GIMPLE IR, with the purpose of target and language-independent optimizations. GIMPLE data structures provide enough information to perform a static analysis of the specification, stored in ASCII files. Following the grammar of these files, a parser reconstructs the GIMPLE data structure, thus allowing further analysis and the construction of additional internal representations, such as Control Flow Graphs, Data Flow Graphs and Program Dependence Graphs. The front end analysis process generates a call graph of the whole application, and the afore mentioned IRs are generated for each call, after GCC optimizations.

2.4.2 Synthesis

The synthesis process (figure 2.6) acts on each function separately. The resulting architecture is modular, reflecting the structure of the call graph. The generated data-path is a custom mux-based architecture based on the dimension of the data types, aiming at reducing the number of flip-flops and bit-level multiplexers. In its release version, Bambu generates the controller as a centralized FSM. For each HLS step, the user can choose among a variety of state of the art algorithms, through configuration files or command line options. The following sections briefly describe some of them, composing the *default* flow.

Resource Allocation

Resource allocation associates operations in the specification to Functional Units (FUs) in the resource library. During the front-end phase the specification is inspected, and operations characteristics identified. Such characteristics include the kind of operation (e.g. addition, multiplication, ...), and input/output value types (e.g. integer, float, ...). Floating point operations are supported through FloPoCo [61], a generator of arithmetic Floating-Point Cores. The allocation task maps them on the set of available FUs: their characterization includes additional features, such as latency, area, and number of pipeline stages. Usually more operation/FU matchings are

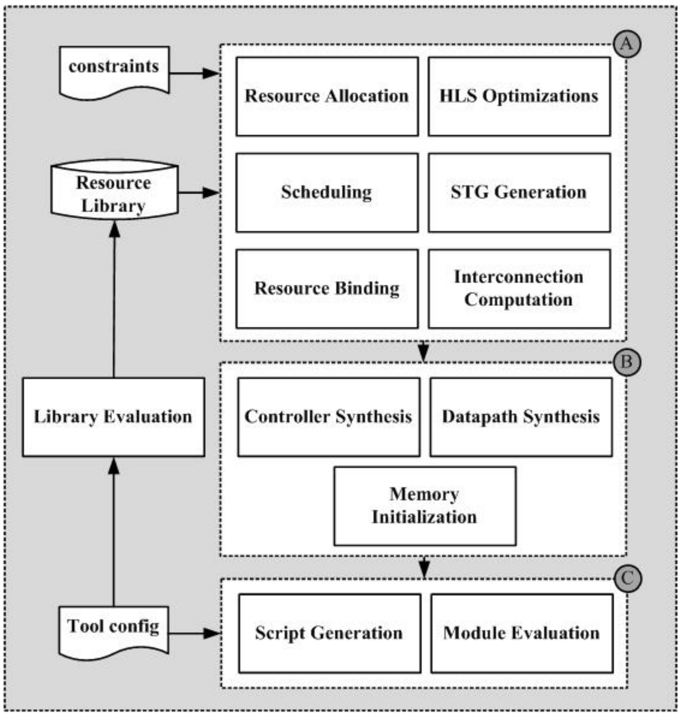


Figure 2.6: *Bambu Synthesis Flow.*

feasible: in this case the selection of a proper FU is driven by design constraints. In addition to FUs, also memory resources are allocated. Local data in fact, may be bound to local memories.

Scheduling

Scheduling of operations is performed through a LIST-based algorithm [148], which is constrained by resource availability. In its basic formulation, the LIST algorithm associates to each operation a *priority*, according to particular metrics. For example, priority may reflect operations mobility with respect to the *critical path*. Operations belonging to the critical path have zero-mobility: delaying their execution usually results in an increase of the overall circuit latency. Critical path and mobilities can be obtained analyzing As Soon As Possible (ASAP) and As Late As Possible (ALAP) schedules. The LIST approach proceeds iteratively associating to each control step, operations to be executed. Ready operations (e.g. whose dependencies have been satisfied in previous iterations of the algorithm) are scheduled in the current control step considering resource availability: if multiple ready operations compete for a resource, than the one having higher priority is scheduled. After the scheduling task it is possible to define a State Transition Graph (STG) accordingly: the STG is adopted for further analysis and to build the final Finite State Machine implementation for the controller.

Module Binding

Operations that execute concurrently, according to the computed schedule, are not allowed to share the same FU instance, thus avoiding resource conflicts. In Bambu, binding is performed through a clique covering algorithm [178] on a weighted compatibility graph. The compatibility graph is built by analyzing the schedule: operations scheduled on different control steps are compatible. Weights express how much is profitable for two operations to share the same hardware resource. They are computed taking into account area/delay trade-offs as a result of sharing; for example, FUs that demand a large area will be more likely shared. Weights computation also considers the cost of interconnections for introducing steering logic, both in terms of area and frequency. Bambu offers several algorithms also for solving the covering problem on generic compatibility/conflict graphs.

Register Binding

Register binding associates storage values to registers, and requires a preliminary analysis step, the Liveness Analysis (LA). LA analyzes the scheduled program, and identifies the *life intervals* of each variable, i.e. the sequence of control steps in which a temporary needs to be stored. Storage values with non overlapping life intervals may share the same register. In default settings, the Bambu flow computes liveness information through a non-iterative SSA liveness analysis algorithm [29]. Register assignment is then reduced to the problem of coloring a conflict graph. Nodes of the graph are storage values, edges represent the conflict relation.

Interconnection Binding

Interconnections are bound according to the previous steps: if a resource is shared, then the algorithm introduces steering logic on its inputs. It also identifies the relation between control signals and different operations: such signals are then set by the controller.

2.4.3 Back-end

Netlist Generation

During the synthesis process, the final architecture is represented through a hyper-graph, which also highlights the interconnection between modules. The netlist generation step translates such representation in a Verilog description. The process access the resource library, which embeds the Verilog implementation of each allocated module.

Generation of Synthesis and Simulation Scripts

Bambu provides the automatic generation of synthesis and simulation scripts based on XML configuration. Table 2.2 lists the tools already supported.

Table 2.2: *External synthesis and simulation tools supported by the Bambu framework.*

SYNTHESIS TOOLS	*SIMULATION TOOLS*
- Xilinx ISE	- Mentor Modelsim
- Xilinx VIVADO	- Xilinx ISIM
- Altera Quartus	- Xilinx XSIM
- Lattice Diamond	- Icarus Verilog

This feature allows the automatic characterization of the resource library, providing technology-aware details during the High-Level Synthesis.

2.5 Conclutions

This chapter has introduced High Level Synthesis, describing which are the inputs and outputs of the process, and providing a formal definition of the main phases composing a typical HLS flow. Such flow has been also characterized describing Bambu, showing how the different HLS tasks are addressed in a state-of-the-art framework. The typical architectural model generated through HLS consists of a data-path and a controller: the latter is usually implemented as a Finite State Machine, built according to a statically computed schedule. One of the major contribution of this work, is to provide an alternative model for the controller, which enables the synthesis of adaptive accelerators featuring dynamic scheduling. The following chapters will describe the limitations of the FSM design, and will detail how the main tasks composing a HLS flow have been addressed, finally allowing the definition and actual implementation of a complete HLS tool, obtained customizing and extending the Bambu framework.

CHAPTER 3

Related Work

High Level Synthesis has a long history behind: almost four decades of continuous improvements characterize its evolution, turning HLS from a revolutionary idea to a design approach adopted in industry. According to the chronological classification proposed in [125], it is possible to recognize three generations in HLS evolution, in addition to a prehistoric period. Each of them has focused on different aspects of HLS, investigating different abstraction layers and specification languages, proposing several architectural alternatives, defining and improving novel synthesis algorithms and techniques. This chapter provides a general overview of the HLS evolution, describing the peculiarities which have characterized each generation, and highlighting novelties and limitations of the proposed methodologies. Section 3.1 provides a brief characterization of HLS methodologies, and accordingly, a coarse classification of design alternatives proposed in literature. Sections from 3.2 to 3.5 describe each HLS generation, showing how it has evolved year over year. Then, Section 3.6, concludes this chapter, identifying the aspects needing improvements and future research directions, thus highlighting how this work improves the State of the Art.

3.1 HLS Design Methodologies Characterization

In order to characterize an HLS design methodology, three main aspects must be considered: the application domain, the adopted specification language and the final architecture obtained from the synthesis. Applications can be coarsely classified into two categories according to their domain: it is possible to distinguish between data-oriented and control-oriented applications. The first includes those applications performing computation on a massive amount of data, as dataflow intensive specifications and Digital Signal Processing. It is the case, for example, of multimedia applications, since they work on a stream of data. The latter includes applications designed for the control. For example, protocol handlers fall in this category, since they implement the set of formal rules that must be observed when two or more entities communicate. Applications falling in one rather than the other category have different peculiarities, often making HLS design methodologies developed for one unsuitable for the other. Hence, one of the most relevant challenge in developing an HLS methodology is to make it adequate for both the application domains. Indeed, complex embedded system designs often include heterogeneous components. Another variable to define in designing an HLS methodology is the input language adopted to describe the specification. From this point of view it is possible to distinguish between two macro-categories: low abstraction level descriptions and high abstraction level descriptions. The first category includes languages such as Behavioral Hardware Description Languages (BHDL). There exist several kind of HDLs adopted in HLS, from the primitive ISP and KARL, developed both around 1977, to the more common Verilog and VHDL. Such languages can precisely describe operations and circuit's organization. For this reason, recent HLS tools consider RTL designs described through HDLs as the *result* of the HLS process. The following steps, turning such designs in physical implementations, are delegated to vendor-provided (for FPGA designs) or other logic synthesis tool (such as Synopsis Design Compiler). The category of high abstraction level descriptions can be in turn divided into the one of high-level programming languages, such as C, C++ or SystemC, and the one of graphical models, such as extended Finite State Machines (FSMs) or Petri Nets (PNs). The choice of the description language is often tightly related to the application domain. For example, describing a control-oriented application as an extended FSM may result simpler than specifying it by means of an high-level C-like language, often leading to better synthesis results. Finally, the target architecture must be defined. As anticipated in Chapter 2, the architectural model is usually composed by datapath and controller. The most common approaches adopt FSMs for the controller. In addition to centralazied FSMs, which represents the dominant solution, architectural altenatives exist such as hierarchical, distributed and/or parallel FSMs.

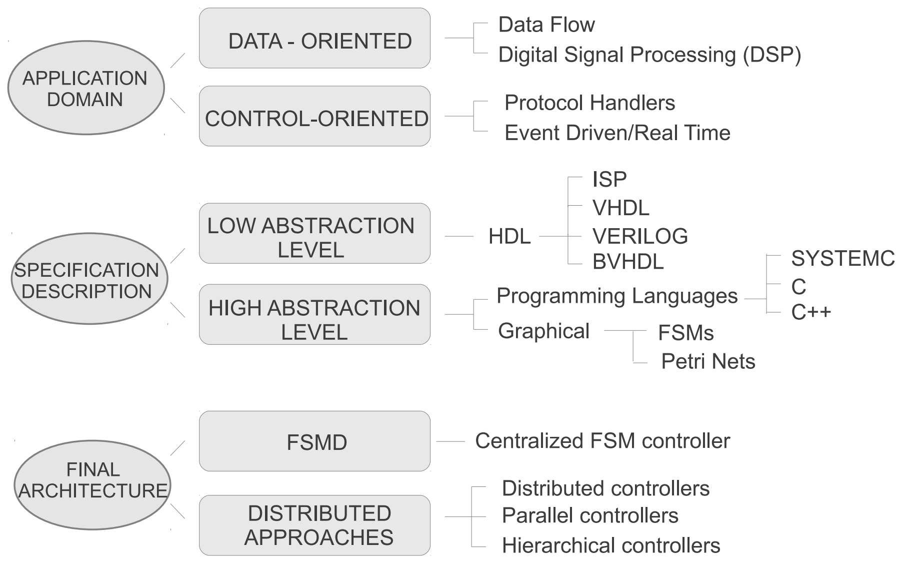


Figure 3.1: *Main Features Characterizing Design Methodologies in HLS.*

Figure 3.1 summarizes the described HLS features. The three identified variables are strictly related each other. A good design methodology should find the right combination between the choice of the specification description and the definition of the final architecture, while obtaining good performances for both the application domains. In the following sections HLS chronological evolution will be tracked, focusing on the choice made about these three features.

3.2 Early Efforts

The seventies provided the basic ideas on which HLS is based, hence literature refers to this years as prehistoric period. In 1974 M. Barbacci noted that, theoretically, it is possible to "compile"a behavioral description of the specification into hardware, without any information about its structural description, such as synthesizable Verilog, thus setting up the notion of design synthesis from a high-level language specification [84]. His research group at Carnegie Mellon University investigated description languages such as Instruction Set Processor (ISP), Instruction Set Processor Language (ISPL), and Instruction Set Processor Specification (ISPS) [23], developed mainly to describe DSP algorithms. They built a pioneering HLS tool, Carnegie-Mellon University design automation (CMU-DA) [67], [146], considering as input ISPS specifications. The target hardware circuit consisted of a structural composition of data path, control and memory elements. CMU-DA also supported hierarchical design and included a simulator of the original ISPS language. Although the innovative flow drew interesting and highly groundbreaking traits for the research, the methods used were very preliminary, and the new approach had a negligible impact in that years on the EDA industry.

3.3 First HLS Generation

The period from 1980s to early 1990s characterized the first generation, that have seen a decomposition of the HLS tasks into several subtasks, such as hardware modeling and controller generation, operation scheduling, resource allocation and binding. This characteristic was common in most of the HLS tools developed in that period, mostly for research and prototyping. Examples include ADAM [82], [100], HAL [150], MIMOLA [126], Hercules/Hebe [62], [63], [115], and Hyper/Hyper-LP [49], [157]. Even if many of the mentioned subtasks are NP-hard problems, and strictly related each other, they are almost independent from a logical point of view, representing different problems that can be faced with different techniques. Such decomposition, nowadays still adopted, had the aim of simplifying the whole problem solving, focusing on one problem at a time through a *divide et impera* paradigm. Factorizing the problem allowed to obtain better performing circuits as result of the synthesis process, addressing each subtask with a different and appropriate methodology. For example, the list scheduling algorithm [16] and its variants are still widely used to solve scheduling problems with resource constraints [147]; the force-directed scheduling algorithm developed in HAL [152] [151] is able to optimize resource requirements under a performance constraint. The Sehwa tool in ADAM is able to generate pipelined implementations and explore the design space by generating multiple solutions [99], [145]. The relative scheduling technique developed in Hebe is an elegant way to handle operations with unbounded delay [113]. Conflict-graph coloring techniques were developed and used in several systems to share resources in the datapath [117], [150]. These improvements led to a successful appliance in the design of filters and other DSP functions, leading to an early proliferation of commercial HLS tools. These include Cathedral and its successors

[124], Yorktown Silicon Compiler [35], and BSSC [190]. However, most of them used custom domain domain-specific languages, generally oriented to describe DSP algorithms. For example, the Silage language used in Cathedral/Cathedral-II, while providing support for customized data-types and code transformations [49] [157], was specifically designed for the synthesis of DSP hardware [124]. The first effort in adopting C-like languages in HLS is represented by HardwareC [114], designed for use in the Hercules system. However, these efforts did not lead to wide adoption among designers. In most commercialization attempts, the technology changed input languages, application scope, and user interfaces many times, making HLS methodologies no more behind the times. Such technological changes led companies to stop HLS tools promotion on the market, although their internal use went on. The fundamental aspect restraining first HLS generation spread was a limited acceptance by final users, i.e. the designers, that found most drawbacks than advantages in using HLS. More in detail, such limitations were concerning:

- *neither necessary nor useful*: concurrent changes in design technologies for integrated circuits, due to recent adoption of RTL synthesis, were revolutionizing design methodologies. In such a scenario, automatic placement and routing technologies offered by RTL synthesis seemed more desirable, and the idea that HLS could fill a design productivity need was considered unlikely.

- *input languages*: the type of input languages adopted in this first generation presented great difficulties, since they consisted in domain-specific HDLs developed ad hoc. Adopting new input languages for a new and unfamiliar design approach was an obstacle for many.

- *quality of results*: the resulting design was often inadequate, due to primitive scheduling, simple underlying architectures and expansive allocation criteria.

- *domain specialization*: methodologies and techniques implementing the early tools were focused and properly worked on DSP design, concentrating on dataflow and signal processing. They were not appropriate for the vast majority of early Application-Specific Integrated Circuit (ASIC) designs, which concentrated on control logic.

In conclusion, it is possible to say that early works in HLS were mainly focused on scheduling heuristics for dataflow-dominated specifications as shown in [149], [76], [36] and [130], with first attempts to automate the synthesis of data paths, as described in many works, the most important of which are [179], [111], [91] and [147]. As well explained in [125], it was the era in which the research mainly concentrated on *datapath-domain-specific* applications.

3.3.1 Architectural Models

In this era the dominant model adopted for the final architecture consisted in a composition of datapath and controller, typically designed as a *centralized Finite State Machine* (FSM). Many works (e.g. [73], [69], [172] and [173]) identified the FSM as one of the fundamental concepts in the design of synchronous electronic circuits. Such approach is still the most widespread.

Centralized Deterministic FSM

The centralized deterministic FSM approach is the simplest from the architectural model point of view. The controller is modeled by a single FSM. As shown in Figure 3.2, the controller structure is composed by a combinational block implementing the state transition function and the output function. The states are generally represented through *edge-triggered* registers, responding to the change of state on the variation front rather than on the

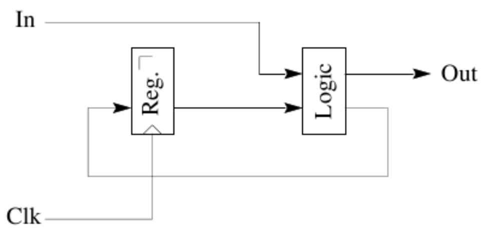


Figure 3.2: *Standard Structure of a Centralized FSM.*

level of voltage. An important advantage of this structure is a simple clocking scheme, with a single clock which is not gated, so that correct function and timing can be easily verified. The maximum speed of such a structure is determined by the time required for changing the flip-flops output and by the maximum propagation delay through the combinational block. Since this delay directly corresponds to the size of combinational block, decentralized approaches seems more desirable. Moreover, since the FSM is intrinsically sequential, parallelism is allowed only for those operations that are assigned to the same control step. In other words, the parallelism extraction task is totally moved to scheduling and binding phases, often leading to an explosion in the number of states [75]. Mitigating this issue is one of the main goals of this thesis work.

3.4 Second HLS Generation

The period from mid 1990s to 2000s characterized the second-generation in the HLS evolution. The substantial improvements in the synthesis tools made their adoption in industry more practical. Major semiconductor design companies developed proprietary HLS tools. These include HIS from IBM [25], Matisse from Motorola [116], Cyber from NEC [185], and CALLAS from Siemens [28]. At the same time the major EDA companies, such as Synopsys, Cadence, and Mentor Graphics, started to commercialize their HLS tools. Synopsys offered Behavioral Compiler [110], Cadence's Alta group provided Visual Architect [90], and Mentor Graphics proposed the Monet tool [71]. Although these tools were tried out seriously by a number of users, the technology achieved a failure again. The main failure reasons can be summarized as follows:

- *input languages*: behavioral Hardware Description Languages (HDLs) were used as inputs. This choice was determined by the mistaken assumption about who would use HLS, i.e. RTL synthesis users, that were at least familiar with such languages. This led to a failure on several fronts. First, RTL designers decided to keep their own tools, instead of switching to new tools that neither provided substantial improvements in performance, nor gave substantial reductions in development effort with the same quality of results. Second, algorithm and software designers were discouraged by the need, that HLS adoption would mean, of learning HDL. Third, using HDL as input language implied simulation times as long as with RTL synthesis, hence HLS adoption did not gave any advantage in terms of time. Finally, it resulted impossible to exploit compiler-based language optimizations.

- *quality of results*: the resulting designs were still inadequate, as well as unpredictable and widely variable. Moreover, formal validation methods were not been proposed yet. Hence, understanding if the synthesis results were correct resulted very hard.

- *lacks in control-dominated specifications synthesis*: designers often used specialized datapath compilers in conjunction with RTL synthesis, as shown for example in [79]. However, HLS could be applied also to control-dominated algorithms. First attempts of applying this extension were proposed in this period, but with worse results with respect to dataflow-dominated algorithms synthesis. Therefore, HLS started to be considered a partial solution, giving quite good results only in certain conditions. At that time, many researchers thought that understanding the reasons of poor results in control synthesis was not a right investment of their time. Among the causes for this lacks there were the use of inappropriate or insufficient Intermediate Representations (IRs), that did not include informations about the control. Indeed, the most common IR adopted in that period was the DCFG, as shown in [139] and [138].

The second generation was the first age of commercial EDA and behavioral-synthesis tools driven by hardware description languages. From the research point of view, many works of that period, as [160], [155] or [174], were focused on new scheduling techniques and strategies. Moreover, about the application domain, most works (e.g. [66]) were still focused on datapath synthesis. However, there was some attempt in control-dependent specifications synthesis, as in [158], [119] and [184].

3.4.1 Architectural Models

In early second generation, first attempts to modify the target architecture were proposed. Indeed, researchers realized that using a centralized FSM, synthesizing the controller, could lead to severe overheads in terms of both area and performance (frequency). A first approach to solve such problems consisted in FSM decompositions, leading to the definition of *distributed controllers*, possibly organized in a *hierarchical structure*. Such decompositions aimed to reduce the area and/or the delay within an FSM. Many works focused on this technique, such as [165] and [140]. In mid nineties, a successive approach followed, based on the idea that the elements having greater impact on area were the interconnections. Moreover, interconnections comported the critical path delay to increase, determining a longer circuit clock cycle. The critical path delay in a RTL circuit with a datapath and a controller is the register to register data flow path with the longest delay. It consists generally of three components: controller delay, control wire delay and datapath delay. Several techniques were proposed to reduce the critical path delay, at different design levels. For instance, careful module generation techniques at RTL level could produce faster modules, improving the performance in the critical path. Moreover, logic minimization methods could be used at the logic level reducing the number of gate levels in the critical path. Unfortunately, these techniques did not provide benefits in wiring delay reduction. Furthermore, especially at high frequencies, the interconnect wiring delay results the dominant factor in the circuit delay. On the other hand, as described in [96], the control path delay has been found to be the slowest segment of the overall critical path delays. For all these reasons, some mid-90s approaches, such as [96] and [51], performed FSMs decomposition with the aim to reduce control path and control wire delays. Finally, FSMs decomposition techniques were proposed to address another problem, i.e. power overhead. In the following the main techniques proposed in these years about FSM decomposition will be deeply explained, after a brief overview on formal FSMs decompositions methodologies. After that, some considerations about distributed and hierarchical controllers will be highlighted.

Formal FSM Decompositions Methodologies

Since sixties FSM decomposition problem was treated from a formal and theoretical point of view. Three main decomposition techniques were identified: *parallel decomposition*, *cascade decomposition* and *generalized de-*

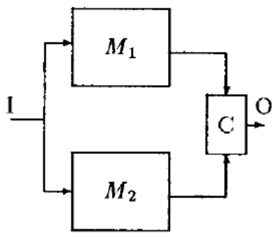


Figure 3.3: *Parallel Decomposition of a Finite State Machine.*

composition.
Parallel decomposition is the simplest technique. As shown in Figure 3.3, the submachines M_1 and M_2 are supplied with the same input sequence I. Such submachines operate independently, providing informations about their internal state to a combinatorial circuit C, whose job is to generate the output sequence O.
Cascade decomposition divides a given finite state machine into a sequence of communicating components. In Figure 3.4 a FSM cascade decomposition is shown. Observe that the initial FSM is partitioned into two submachines M_1 and M_2, each driven by the same input sequence I. The obtained sub-machines do not operate independently. Indeed, M_2 is supplied, by means of auxiliary inputs (see the edge from M_1 to M_2 in Figure 3.4), with information about the current internal state of M_1. Such information

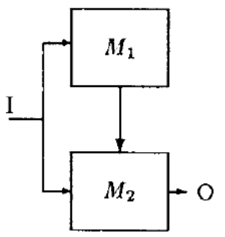


Figure 3.4: *Cascade Decomposition of a Finite State Machine.*

influences the state transitions of M_2, and enable M_2 to generate the appropriate output sequence O. The possibility of passing state information from M_1 to M_2 makes cascade decomposition a more powerful technique than parallel decomposition. Then the prior can be viewed as a generalization of

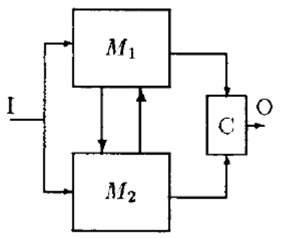


Figure 3.5: *Generalized Decomposition of a Finite State Machine.*

the latter.
Generalized decomposition produces a model in which each submachine is provided with information about the current state of the other, as shown in Figure 3.5. In this case the internal behavior of each machine depends both by the behavior of the other and by the input sequence I. Parallel and cascade decompositions can be viewed as particular cases of the generalized one.
Among the works treating FSMs decomposition in a formal way there is the one proposed by Hartmanis [97] in 1960, who applied an algebra on partitions of states. This work focused on *cascade decomposition*. Such methodology was extended in the few subsequent years in [123], [87] and [98] to preserve the covers for the cascade decomposition found. Parallel decomposition was considered [78], generalized decomposition in [165].

FSM Decomposition reducing area and delay within a FSM

From late eighties, researchers started to apply decomposition techniques, formalized years before from a theoretically point of view, in FSMs design for logic implementation, as reported in [165] and [140]. They found soon that cascade decomposition has limited use in FSMs design, since specifications of centralized controllers in microprocessor chips do not usually have good cascade decompositions. Obviously, also parallel decomposition resulted inadequate, being less general than cascade one. Hence, they started to develop techniques for the identification of *factors* (i.e. controller partitions) producing a good generalized decomposition. In [165], for instance, such factors were identified in sets of states and transition edges obtained from a State Transition Table [167] specification of the initial FSM. These factors were extracted and represented as *factoring sub-machines*. Then the occurrences of these factors in the original machine were replaced by calls to the factoring submachine. Procedures were defined to find all the *exact factorizations*, i.e. those that maximally reduce the number of states and transition edges in the original machine. An important result of this research is expressed by the following theorem, that shows the significant advantages of exact factorization in terms of area, due to the great reduction in the total number of edges and states that such technique involves.

Theorem 1. *A decomposed submachine M_i, produced by factorization from an original machine M, via an exact factor with $N_I(i)$ internal states and $N_E(i)$ exit states in each occurrence $O_F^i \in M$, will have*

$$\sum_{i=1}^{N_R} (|e(i)| - N_I(i))$$

edges less than the original machine M, where $e(i)$ is the set of internal edges in $O_F^i \in M$, and N_R is the number of possible factors for M.

Unfortunately, exact factorization often produced too small submachines, resulting in useless decompositions. Moreover, exact factorization may not exists at all for a given machine. Hence, techniques to find good, though inexact, factors in an FSM were proposed.

FSM Decomposition reducing control path and control wire delays

Another group of researchers concentrated on the critical path reduction, as [51] and [96]. More in detail, techniques to reduce control path and control wire components of the overall delay were proposed. Researchers focused on such components since the prior was found to be the slowest segment of the critical path, and the latter was found to be the dominant factor, especially at high frequencies.

In [96] the authors identified *control points* in a machine, representing minimal partitions of a centralized controller, aiming to divide it into multiple local controllers. Each control point can be viewed as the controller managing one operation in the machine. Hence, if an FSM contains N operations, then N control points can be individuated inside that machine, one for each instruction. The approach used a *wire length extraction* technique followed by *clustering* of control points into local groups. Clustering was targeted at minimizing wire lengths.
Papachristou and Alzazeri extended such work in [51]. They firstly partitioned an RTL based controller output into control points, and then partitioned the datapath around its constituent. At this point the control points individuated were grouped so that all control points enabling the same kind of operation were placed in the same datapath partition. Hence, in this case, clustering resulted from datapath partitioning. In this way multiple local controllers were generated, each controlling one datapath partition, also said functional block. The last step of such technique was in the layout phase, in which each controller were placed physically close to its corresponding functional block, shortening wire lengths and thus reducing delays, especially at high frequencies. Observe that such approach starts from an RTL circuit, and thus it is not included in the HLS flow. Better results can be obtained integrating this work in the RTL generation process. Moreover, datapath partitioning and close physical placement of local controllers can be used to infer useful hints for resource binding.

Distributed Controller

As above mentioned, a distributed controller structure can be obtained by FSM decomposition. From what said so far, it is possible to infer that the objective of decomposition was to reduce the delay inside an FSM, or to reduce the critical path delay, rather than to reduce the power overhead. The concept of parallelism extraction is not included in the aims of realizing a pure distributed controller structure. As in the case of centralized FSM, the task of individuating set of instructions that can be simultaneously executed does not concern the controller. Consider, for example, the above described technique proposed by Eppling in [96], or the one presented by Papachristou et al. in [51]. From the analysis of these methodologies it is possible to infer that pure decomposition can directly be applied on an RTL circuit, without being included in a HLS flow. In other words, decomposition takes as input a centralized FSM, that can be obtained through HLS, and manipulate it to reduce area and delay. Scheduling and binding have been already performed on the centralized FSM and do not change after decomposition.

Hence the set of possible states for the distributed machine is the same obtained for the centralized one. However, in this case the state of the entire machine has not to be explicitly represented, since can be obtained as a composition of the submachines states, leading to a considerable reduction in the number of total states.

Hierarchical Controller

Inside Hierarchical Controllers different subcontrollers are organized in a hierarchical structure, usually in accordance with the hierarchical relation obtained from the *Hierarchical Task Graph* (HTG) [131] of the specification. In the multi-level hierarchy, a controller at one level distributes groups of operations among its direct descendant. Each controller can start its computation after the activation signal from its father has been received. When a local computation is terminated, the corresponding subcontroller send a signal to its father, implementing a synchronization mechanism that enable controllers at higher levels to properly activate subcontrollers.
A hierarchical controller structure is completely compatible with distributed one. Moreover, it is also compatible with parallel controllers, that will be presented in the next. Hence, it is possible to implement distribute and hierarchical controllers, or distribute, parallel and hierarchical controllers.

NFA-Regular Expressions Based Controller

Usually, machines such as protocol handlers or communication encoders, results too complex to be described through a deterministic FSM model. They need instead *Nondeterministic Finite Automata* (NFA) to be concisely described. An NFA, or *Nondeterministic Finite State Machine* (NFSM) [132], is a finite state machine in which for each pair $\{state, input\}$ there may be several next states. It was formally proved that for any given NFA it is possible to construct an equivalent deterministic FSM through standard methods, such as *powerset construction* or *subset construction* (see [167], Theorem 1.19, section 1.2, and [103]). Obviously, the first step needed to synthesize an application, described with an NFA-based model, is transforming such description in its deterministic equivalent. For this reason, NFAs have unfeasible complexity when implemented in hardware with standard approaches, that simply transform it in a deterministic FSM. For example, one of the most widespread techniques were based on the use of *Esterel* [74], a synchronous reactive language allowing an inherently nondeterministic machine description. Its commands reacted to inputs from the outside world, by performing tasks and sending outputs. Each reaction to a specified input was allowed to occur independently of other reactions, creating an NFA model. However, the Esterel compiler, as above mentioned, created a deterministic *State Transition Graph* (STG) [172] from the NFA specification, often having unacceptable complexity. As a consequence, various techniques were proposed to address this problem, based on the concept of classical *Regular Expressions* (REs) [17]. It is well known that any FSM can be specified as a regular expression, representing the set of all the strings belonging to the *formal language* recognized by the automaton. Even though this means that the classical regular expression description is always allowed, such specification is not guaranteed to be as concise as other types. This is the main reason why regular expressions alone are not enough, and should be used together with a nondeterministic description. Starting from such considerations, in [162] and [20], regular expressions were used as a specification for Programmable Logic Array (PLA) designs, to be converted into an NFA state transition diagram, which in turn was directly encoded as product terms of a PLA implementation. However, such technique may lose some of the informations present in the regular expression. For example consider the regular expression $e = (a|b)^+(b|c)^*$ over the alphabet $\Sigma = \{a, b, c\}$. Such expression can be partitioned in two components: $(a|b)^+$ and $(b|c)^*$. Such kind of decomposition, defined *natural partitioning* in [14], can be useful to identify points in which the machine can be divided in sub-machines, distributing the control for a possible FSM factorization. When the correspondent NFA state transition diagram is obtained from e, such information about natural partitions may go lost. Indeed, both the partitions contain the character b. Hence, when b is read, if we consider only the NFA representation, there is no way to understand what natural partition this character belongs to. A subsequent approach, proposed in [15], was the *Production Based Specification* one. They considered, as specification language, the productions derived from the *grammar* corresponding to the formal language recognized by the associated automaton. Indeed, as described in [137], there exist three equivalent models for the description of languages: automata, regular expressions and grammars. Similarly, from the finite state machines point of view, there exist three equivalent models for a (deterministic or nondeterministic) FSM description: the language it recognizes, the regular expression describing such language and its associated grammar. The production based specification model provided a hierarchical regular expression language augmented with some unique operators. An algorithm for direct construction of the circuit from a regular expression based tree was presented, which did not require conversion of the RE to a NFA state transition diagram. This direct

construction often produced fast circuits, but with redundant state bit encodings.
Finally, Crews et al. [14] discussed techniques for high-performance controller synthesis, from the complexity point of view. More in detail, they proposed sequential optimization techniques whose complexity scales with the number of state bits, rather than the number of states. This work aimed to provide viable synthesis techniques for designs which are too large for synthesis with conventional methods. The methodology proposed in [14] started from classical *regular expressions* [17] specifications, deriving from it an NFA description. Once obtained, the NFA model was encoded as a *tree-based extended regular expression*. They assumed as controller specification regular expressions in the form of *Directed Acyclic Graphs* (DAGs) [106]. Table 3.1 describes the notation adopted in the specification DAG. DAG representation allows to specify any completely deterministic automata without making use of traditional deterministic models, such as STGs or actual state encoding. Moreover, from this specification, there are direct gate level implementations which scale with the number of state bits in the controller, which can be logarithmically smaller than the number of machine states. Once obtained the DAG representation, they performed natural partitioning, identifying proper sub-DAGs. For example, considering the manipulation rules they adopted to minimize the original ER, specified as follows

Table 3.1: *Regular Expression DAG Symbols.*

symbol	meaning	nodes type
,	concatenation of events (left then right)	sequential non-leaf nodes
$\|\|$	OR (either event below)	sequential non-leaf nodes
&&	AND (events occur simultaneously)	sequential non-leaf nodes
$*$	Kleene closure (0 or more)	sequential non-leaf nodes
$+$	1 ore more	sequential non-leaf nodes
action	designates an output activation	sequential non-leaf nodes
function	boolean function (of inputs only)	combinational (terminal) nodes

$$(A,B)||(A,C) \rightarrow (A),(B||C) \qquad \text{(Rule 1)}$$
$$(A,C)||(B,C) \rightarrow (A||B),(C) \qquad \text{(Rule 2)}$$
$$A||A \rightarrow A \qquad \text{(Rule 3)}$$
$$A,(A)^* \rightarrow (A)^+ \qquad \text{(Rule 4)}$$
$$(A^*)^* \rightarrow (A)^* \qquad \text{(Rule 5)}$$

$$A, (A\{action\}) \rightarrow (A, A)\{action\} \qquad \text{(Rule 6)}$$

it is possible to identify a sub-DAG inside each open/close round-parentheses pair. Minimization is the main optimization technique proposed in this work. It aimed to remove unobservable states from the system. After that, each unique action in the DAG was put into correspondence with a unique output of the controller. The output was set high only if the sub-DAG below the action accepted, i.e. when the sequence of inputs for that sub-machine matched the entire sequence specified in the DAG. After reducing the number of terminals in the tree, the circuit was synthesized by traversing the resulting DAG. The construction required one register for each path to a terminal node. The circuit was generated recursively, by allocating registers at the terminals and constructing logic functions of the register outputs (present state bits) according to the type of sequential operator at each node. Logic functions were stored as Binary Decision Diagrams (BDDs) [21] during construction.
Despite such methodologies presented some advantages in controller synthesis for complex applications, regular expressions and nondeterministic automata based approaches resulted too much computationally expensive, mitigating their adoption.

3.5 Third HLS Generation

The early 2000s introduced the third (and current) HLS generation. The most important feature characterizing this era is the adoption, as HLS inputs, of C-like specifications described through C, C++ and SystemC code [181]. SystemC is a C++ class library which provides features and semantics similar to HDLs such as Verilog and VHDL. These features include structural hierarchy and connectivity, clock-cycle accuracy, four-level logic and bus resolution functions, making SystemC specification more detailed than behavioral code, and suitable for HLS. Nevertheless, SystemC still exposes programmers to low-level details, and in recent years this aspect slightly lowered its popularity in favor of behavioral C and C++.

Table 3.2 summarizes some of the most useful high-level languages features for effective C-based design and synthesis, as identified in [56]. Support for C-based specifications in HLS:

1. extends the range of potential users to designers more familiar with programming languages than with HDLs or Domain Specific Languages (DSLs)

Table 3.2: *Useful high-level languages features for C-based design and synthesis.*

Language	Constructs	Benefits
C	Arbitrary-precision integer types	Bit-accurate designs, QoR
	Floating point types	Floating point arithmetic
	Function calls	Modular design hierarchy
	Pointers	Efficiency and flexibility
	Structs	Data encapsulation
C++	Fixed point types	Fixed point arithmetic
	Templates	Parameterizable design
	classes	Object-oriented modeling
	Pointers	Efficiency and flexibility
SystemC	Processes	Coarse grained concurrency
	Clocks	Multi-clock design
	TLM	Fast simulation

2. facilitates complex tasks such as HW/SW codesign, HW/SW partitioning, and Design Space Exploration. For example, modern Systems-on-a-Chip (SoCs) usually embed soft processors, which interface with custom hardware: using the same description language for hardware and software allows the designers to quickly explore and evaluate different system configurations. Other emerging platforms (i.e., hybrid architectures) couple general purpose multi-core processors with reconfigurable devices to accelerate some portions of code. On these systems, a unified description paradigm may facilitate the mapping of software kernels to custom hardware

3. allows HLS to take advantage of optimizations techniques inspired or based on software compilation (e.g. [86], [85], [153], [95], such as expression rewriting, constant propagation, dead code and common subexpression elimination [19]. For this reason, several HLS flows (e.g. Autopilot [192], LegUp [37], Bambu [1]) rely on software compilers, such as GCC or LLVM, to generate the optimized Internal Representations (IRs) starting from which they perform the hardware synthesis

4. improves the design quality: designers can take advantage of rich constructs to maximize for example design reusability, modularity, as well as synthesis QoR

5. reduces the verification effort, through the automated generation of RTL test-benches starting from high-level test benches provided by the user (Figure 3.6)

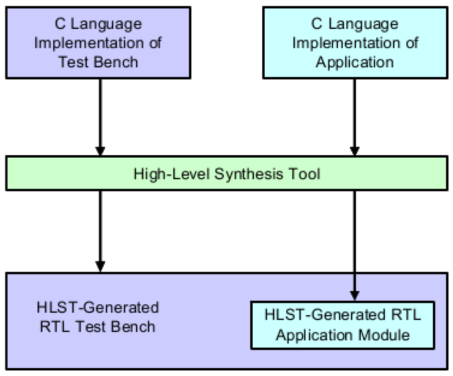

C Language Implementation of Test Bench
C Language Implementation of Application
High-Level Synthesis Tool
HLST-Generated RTL Test Bench
HLST-Generated RTL Application Module

Figure 3.6: *Tipical HLS tools output including RTL and RTL test benches.*

Another key aspect which has influenced the evolution of HLS in the last years has been the rise of Field Programmable Gate Array (FPGA) devices. FPGAs have constantly improved in capacitance, speed, and often include hardware components such as embedded multipliers, local memories and even soft cores, which can be profitably exploited by target-aware HLS frameworks [109]. For example soft cores may be exploited when processing complex application whose complete hardware implementation may not fit on the reconfigurable fabric: the user or HW/SW partitioning techniques may identify critical portion of code to synthesize as custom hardware, while the rest of the code executes, as software, on the processor [180]. Many recent HLS tools have been designed specifically targeting FPGAs; these include GAUT [141], ROCCC [83], [183], SPARK [86], [85], LegUp [37], and Trident [177], [176]. While for FPGA designs the adoption of automated synthesis flows is mainly exploited to quickly map algorithms onto hardware with limited costs, in the ASIC industry HLS has been considered mainly for DSE and fast prototyping [163]. Indeed HLS represents an efficient means for design space exploration: it can generate several design alternatives, with different characteristics, and provide an estimates of the resource utilization and clock frequency without invoking the logic synthesis tools. Thus it enables early design exploration, enabling a fast identification of achievable cost-performance points. In rapid prototyping, a system can be quickly modeled in an FPGA, enabling for example system simulation and system-level performance analysis. Moreover, area and timing estimates can be used to assess the synthesis results and, as necessary, to make improvements to the implementation by modifying the high-level representation.

3.5.1 HLS and EDA Industry

Despite the interest of both the research and the design communities, most of previous generation commercial HLS efforts have failed. [56] identifies among the others, five main reasons for such failures:

1. Lack of comprehensive design language support: most of the approaches described in Section 3.3 and Section 3.4 raised only slightly the abstraction level of the design process, using partially timed behavior HDL for specification. Early C-based approaches instead, had a poor coverage of the considered languages

2. Lack of reusable and portable design specification: many tools forced the users to embed in the specification details such as timing and interface information, as well as design constraints.

3. Narrow focus on datapath synthesis: little effort if any was spent on aspects such as interfaces with other hw/sw modules and system integration

4. Lack of satisfactory QoR: serious problems with timing closure between logic and physical design

5. Lack of a compelling reason/event to adopt a new design methodology: moving towards HLS despite of classical RTL design flows appeared a risk, also because there was no evidence on the HLS effectiveness in terms of QoR.

By mitigating these issues, third generation commercial tools have started to to meet users' needs and expectations. Most widespread frameworks include AutoESL's AutoPilot [192] (recently acquired by Xilinx), Cadence's C-to-Silicon Compiler [5], Forte's Cynthesizer [141], Mentor's Catapult C [31] [189], NEC's Cyber Workbench [141], and Synopsys Synphony C [12] (formerly, Synfora's PICO Express, originated from a long-range research effort in HP Labs [108]). This success is confirmed by market statistics provided by the EDA Consortium (EDAC) Market Statistics Service (MSS) [2]: Figure 3.7 reports EDA revenues for the major categories, i.e. Computer-Aided Engineering (CAE), Printed Circuit Board & Multi-Chip Module (PCB & MCM), IC Physical Design and Verification, and Semiconductor IP and Services), from Q1 1996 through Q1 2013. CAE, in which HLS is located and which represent the main source of income in the EDA industry, while reaching its revenues peak in 2013, has seen a slight decrease in sales between 2006 and 2010. However, more detailed

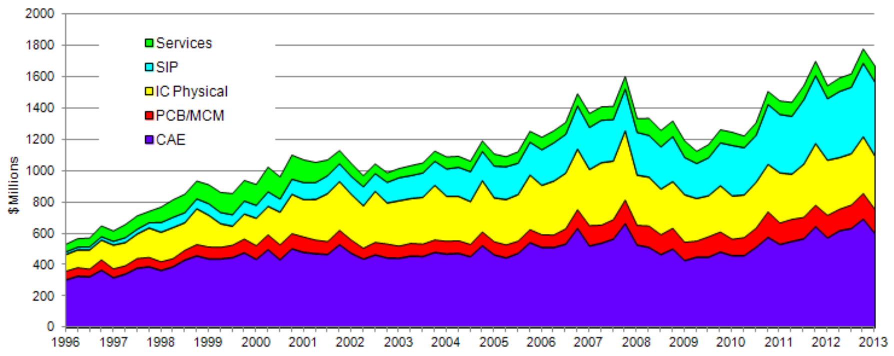


Figure 3.7: *EDA revenue history, 1996 - present.*

Table 3.3: *EDA segments revenue analysis ($ MILLIONS). Source: EDAC MSS Statistics Report and Desaisive Technology Research analysis.*

	2006	**2007**	**2008**	**2009**	**2010**
High-level design and verification (ESL)	121.6	139.5	159.5	180	205
Year-over-year growth	16%	15%	14%	13%	14%
Percentage on CAE	5%	6%	7%	8%	9%
RTL design and verification (traditional)	651.2	696.9	634.6	625	625
Year-over-year growth	7%	7%	-9%	-2%	0%
Percentage on CAE	29%	28%	29%	28%	28%
Overall CAE segment	2213.4	2467	2199.7	2200	2250
Year-over-year growth	14%	11%	-11%	0%	2%
Total EDA market	4078.3	4464.8	3853.7	3850	3960
Year-over-year growth	14%	9%	-14%	0%	3%

analysis shows that the High Level Design and Verification segment, on the contrary, has grown in this time-frame, as reported in Table 3.3. The RTL design and verification segment has lost market share; however in 2010, the latter has registered revenues for more than three times higher with respect to the prior. This indicate that RTL remains the dominant specification and synthesis level. Detailed statistics for the 2010-2013 time frame are not reported, since released to consortium members only.

Recently, third-party companies such as Berkeley Design Technology Inc. (BDTI) [3] offer HLS tool certification programs for the evaluation of HLS tools for FPGAs, in terms of both QoR and usability. For example, [33] analyzes productivity and QoR of the Autopilot tool for the synthesis of DSP applications, comparing performance of the automatically generated accelerators against mainstream DSP processors. For the con-

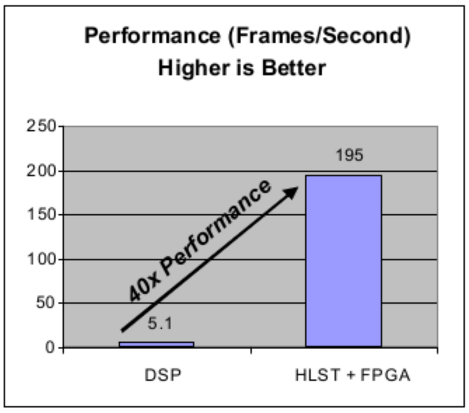


Figure 3.8: *Maximum frame rate achieved for a video application on a DSP processor and an FPGA plus HLS tools.*

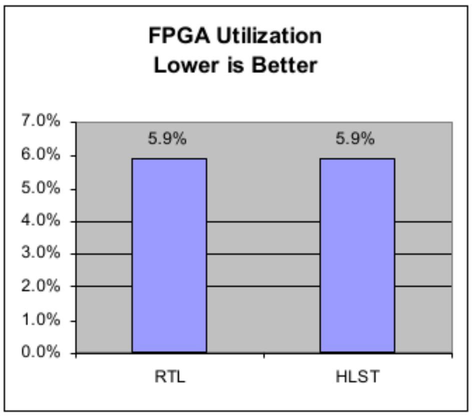


Figure 3.9: *FPGA resource utilization on a wireless receiver, implemented using HLS tools versus hand-written RTL code.*

sidered application, the HLS approach provided 40X better performance (as shown in Figure 3.8) with FPGA resource utilization levels comparable to hand-written RTL code (Figure 3.9). Moreover, the analysis report a similar development effort for both the HLS and DSP-processor approaches, despite historically DSP processor programming resulted easier. Similarly, [26], show that, in some high parallelizable signal processing applications, FPGAs could achieve up to 100X higher performance and 30X better cost-performance than DSP processors. However, these results are strictly related to the particular application domain.

3.5.2 Architectural Models

Third HLS generation introduces a new controller architecture, i.e. *parallel controllers*. In literature the two terms "distributed"and "parallel"have been often used as synonyms. Despite many similarities, there exists a subtle difference between them. Both the terms indicate the presence of an underlying structure composed by sub-controllers that work simultaneously, possibly interacting each other to compute their next state. However, the term distribution, only refers to this aspect, highlighting the distributed nature of the design. Parallel controllers instead, are able to operate as completely independent modules, making their adoption particularly profitable for parallelism exploitation at the process level, especially when statical analysis cannot provide accurate information on the latency of each process. It is difficult to obtain this behavior with a centralized FSM model, since it is inherently serial: computation proceeds as a sequence of control steps, and the system in each given step is in a well defined state. In parallel controllers instead, each sub-controllers is characterized by its own state, and the overall system state can be obtained through the composition of local states. In addition, identifying coarse-grained parallelism from a C-like serial specification is a complex task. However, it can be addressed, as demonstrated in [122] and in this thesis work, by choosing a proper IR for the synthesis flow. Nevertheless, most of C-based HLS tools use CDFG-based IRs: while allowing ILP exploitation, usually CDFG analysis fails in recognizing available coarse grained parallelism, as in the presence of concurrent loops or function call. For this reason, proposed approaches exploiting parallel controllers do not consider C-like specification, but adopt description languages which directly expose coarse grained parallelism, and which facilitate the management of the interactions between processes. Relevant examples of such approaches describe specifications through Petri Nets [39] [104] and Communicating sequential processes [92]. A *Petri Net* [101] (PN) is a mathematical modeling language for the description of distributed systems. Thanks to its ability in representing the parallelism inside an application, this model has been proposed as graphical specification description language in HLS, as an alternative to high-level programming languages. A Petri net can be viewed as a marked version of a *Petri Net Graph* [101].

definition 3.5.1. *A Petri Net Graph (PNG) is a 3-tuple (S, T, W), where:*

- *S is a finite set of places, i.e. nodes representing conditions*

- *T is a finite set of transitions, i.e. nodes representing events that may occur*

- *$W : (S \times T) \bigcup (T \times S) \to N$ is a multiset of arcs, i.e. it defines arcs and assigns to each arc a non-negative integer arc multiplicity.*

Observe that no arc may connect two places or two transitions.

definition 3.5.2. *A Petri Net is a 4-tuple (S, T, W, M_0), where:*

- *(S, T, W) is a Petri net graph*
- *M_0 is the initial marking, a marking of the Petri net graph*

A Petri net representation of a controller structure is equivalent to the one obtained by dividing the specification of a controller into a number of concurrent processes, producing a set of sub-controllers. Then controllers are implemented as FSMs and linked with control lines and/or semaphore bits. An example of PN specification of a controller is shown in Figure 3.10.

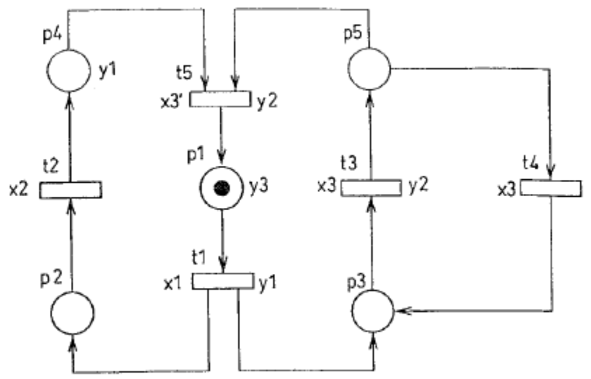


Figure 3.10: *Example of Petri Net Specification of a Controller.*

Early work from Biliński et al. [112] [105], proposed to exploit the graphical representation of concurrency provided by Petri nets to synthesize a parallel controller structure. They based this choice on the observation that such graphical representation is often easier to understand, hence it can reduce the likelihood of parallel synchronization errors. Examples of HLS techniques based on Petri nets may be found in [161] and [57]. Similar ideas are considered in [188], where the authors propose a methodology for the HLS of multi-process behavioral descriptions; the HLS flow considers as inputs applications described through the concurrent communicating processes specification paradigm [92]. However, in these techniques the user is required to provide low level details on synchronization between the concurrent processes, reducing the benefits of HLS.

3.6 Conclusions

Latest generation HLS tools have demonstrated the maturity to be successfully used in industry for specific domains, and showed significant progress in providing wide language coverage, robust compilation technology, platform-based modeling, and system-level integration. However, there are several aspects which need further improvements. To represent the mainstream approach, HLS must provide good QoR in a wider range of domains not limited to DSP. When considering complex applications, in which parallelism does not occur in the form of ILP, the gap between automatically generated and hand-written design is still significant. The adoption of C/C++ as specification language can be identified as one of the main reasons for this lack [135]: C/C++ offers the highest level of algorithmic exploration, but is characterized by sequential execution semantics which makes difficult to identify coarse grained parallelism. This limitation is strengthened by the architectural model usually considered, based on the implementation of centralized FSM controllers, whose execution paradigm is also inherently serial. Literature shows design alternatives, i.e. parallel controllers, which are particularly suitable for supporting parallel execution. However, proposed approaches act on specifications described through Petri nets or process networks, and this assumption nullifies the valuable advantage of having a common description language for hardware and software. This thesis work instead, proposes an HLS methodology targeting a parallel controller architecture while adopting C code as specification language. The proposed techniques improve parallelism exploitation at different granularities, regardless of the application domain.
The designed flow embeds an efficient front-end analysis phase, which enables automatic parallelism identification. However Task Level Parallelism (TLP) extraction may still be limited when considering arbitrary sequential specifications. This aspect is highlighted, for example, in [56], which suggest further investigations on parallel programming models. Languages and APIs such as CUDA, pthreads and OpenMP can explicitly expose TLP, and facilitate concurrency management through explicit synchronization directives. In this work these opportunities have been explored, designing the HLS flow also to target parallel applications, where concurrency is specified through pragma insertion. Another feature that [56] recognizes

as a research priority, is the support for complex memory hierarchies. Together with exploitation of parallel programming paradigms, this represents a crucial aspect for the emerging field of high-performance reconfigurable computing [70]. Scientific applications common in HPC require efficient access to gigabytes (often terabytes) of external memories, shared between different processes (for task parallel applications) or hardware components (e.g. host processors/accelerators in SoCs). Current HLS solutions lack in a sufficient abstraction of the external memory accesses, often exposing the programmer to low level details of bus interfaces [136]. This lack has been alleviated through the definition of a custom Memory Interface Controller (MIC), which introduces an abstraction layer between hardware accelerators design and external memory structure. The MIC allows fine grained parallelism exploitation on memory accesses, automatically managing concurrency on shared resources, and embeds support for atomic memory operations for synchronization. The MIC has been initially designed as a custom IP, suitable for easy adoption in both RTL and HLS flows. Then it has been explored an alternative design, hierarchically structured, which facilitates the MIC allocation during the automated synthesis process, and improves reusability of generated components. The contribution of this thesis work will be described in depth in the next chapters. Each chapter will cover different aspects, and for each specific topic, it will provide additional details on related work.

CHAPTER 4

The Parallel Controller Architecture

The analysis of the State of the Art has shown how High Level Synthesis has evolved over time, providing impressive improvements in each generation. Current C-based HLS frameworks provide wide language coverage, integrate design space exploration engines, exploit embedded resources provided by modern FPGAs, and most important of all provide good Quality of Results for specific domains. However, most of the improvements, in terms of QoR, have been obtained progressively refining the different algorithms composing the HLS process, developed considering the FSMD as the golden model for the target architecture, and tailoring the research around this assumption. This mimics the design of software compilers, which aim at maximizing the exploitation of already designed hardware. However, even if literature proposes several alternatives to the centralized FSM model suitable for HLS, they have not yet been exhaustively explored. Among them parallel controllers appear very promising, especially for parallelism exploitation. Nevertheless, proposed approaches in HLS targeting parallel controllers have not considered the adoption of programming languages for specification, thus resulting in contrast with the current trends. This thesis investigates the benefits of adopting parallel controllers in HLS, whose design enables the synthesis of adaptive hardware components. The designed controller architecture [154] consists of a set of interacting control elements, which independently manage the execution of one or more operations at runtime, without the need of a pre-computed schedule. This induces the adaptive behavior of the generated hardware. This chapter describes the controller design, and shows how the proposed techniques can improve both instruction level and coarse grained parallelism exploitation. Section 4.1 motivate the approach, highlighting the limitations of the FSM model which suggest alternative solutions; Section 4.2 analyzes the related work, describing similar approaches and FSM-based techniques for *mimicking* adaptive behaviors; Section 4.3 details the proposed controller design; Section 4.4 highlights the adaptive behavior of the designed architecture, through an illustratve example. Finally, Section 4.5 draws concluding remarks.

4.1 Motivation

Common HLS approaches implement the controller as a centralized Finite State Machine (FSM), which is built according to the statically computed schedule. The FSM-based execution paradigm is inherently serial: the FSM executes the application in a sequence of control steps. Thus, it can only exploit Instruction Level Parallelism (ILP) within a single execution flow. However, ILP is limited in most applications. Compiler-based optimizations such as loop fusion, loop unrolling and function inlining allow extracting more ILP from loops and parallel function calls, but these flattening strategies often provide limited benefits. First, they cannot always be applied; second, their adoption may heavily increase the complexity (in terms of number of operations) of the compiled specification, affecting the resulting designs in both area and performance. This is a severe restriction for several application classes. In addition to programs which present parallelism at the function call level, the mostly affected are Control-Flow Intensive (CFI) specifications and applications that include Variable Latency Operations (VLOs) [134] [159]. CFI specifications present parallel control constructs such as loops and conditionals. However, the typical scheduling techniques serialize those portions of code. VLOs, such as memory accesses and speculative operations [64], affect scheduling in a similar way. They require ways to notify execution completion at runtime, because their execution latency is not know before-hand, at design-time. Usual approaches to support VLOs do not allow concurrent execution. They exploit a *done* signal, which modifies the execution flow: while waiting for the done signal, the controller stalls all the other po-

```
int loops(int k1, int a1, int a2){
   int i1, i2, x1, x2, res;
   for(i1=0; i1<k1; i1++){
      a1 = a1*i1;
      a1 = a1−k1;
   }
   for(i2=0; i2<k1; i2++){
      a2 = a2*i2;
      a2 = a2−k1;
   }
   res = a1 + a2;
   return res;
}

int call_loops(int k1,int k2,int k3,int a1,int a2){
   int f1=loops(k1, a1, a2);
   int f2=loops(k2, a1, a2);
   int f3=loops(k3, a1, a2);
   int f4=loops(k1, f1, f2);
   int f5=loops(k1, f4, f3);
   return f5;
}
```

Figure 4.1: *Example C specification.*

tentially concurrent operations. Sophisticated scheduling techniques [118] can exploit parallelism also in these situations. They statically compute all the possible runtime execution orderings, and build the corresponding finite state machine. However, these approaches lead to very complex controllers, with unsustainable area requirements and very low operating frequencies, or even to controllers not feasible at all. Function *loops* (Figure 4.1), which presents two parallel for loops, is an example of CFI specification. To execute the loops concurrently, a FSM-based HLS flow must identify and replicate operations associated with loop bodies for all the possible situations that may happen at runtime. This allows exposing all the available parallelism within a single execution flow, as demonstrated through the (Basic Block level) State Transition Graph (STG) of Figure 4.2. The same issues affect the synthesis of function *call_loops*, where the parallelism is coarse grained, in the form of concurrent function calls. In such a case, the complexity of the STG needed to manage multiple execution flows greatly increases. The STG shown in Figure 4.3 features 11 states and 34 transitions, which only support a maximum of 3 parallel flows. In general, the complexity of product FSMs, in terms of number of states and transitions, grows exponentially with the number of concurrent flows. To manage n concurrent flows, such complexity is $O(2^n)$ for both the number of states and transitions [40]. This heavily affects the resulting design in

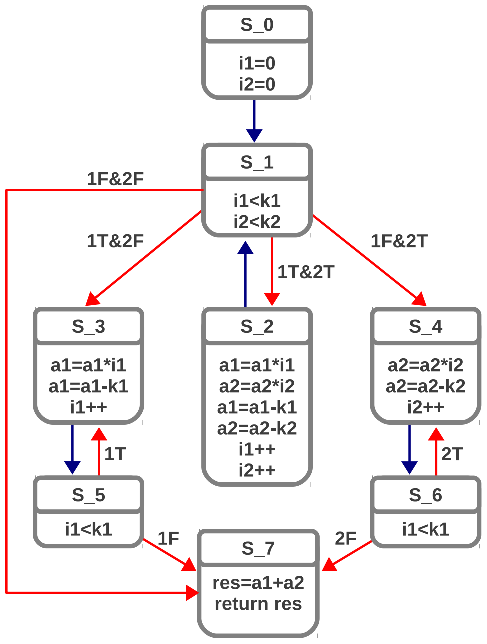


Figure 4.2: *Product STG for function* loops*, at basic block level.*

both area and frequency, leading to unfeasible solutions for highly parallel applications. This chapter proposes an adaptive parallel controller design, suitable for the automatic generation of accelerators supporting parallel execution and dynamic scheduling. The complexity of the controller only grows linearly with the number of operations, and is independent from the number of concurrent flows. In opposition to other HLS flows, which build centralized FSMs, generated controllers consist of a set of communicating modules, each one associated with an operation. The approach does not require the definition of any execution order (scheduling) at design time, and allows run-time exploitation of parallelism. The controller modules start execution of the associated operations as soon as all their dependencies are satisfied and resource conflicts resolved. They communicate through a lightweight token-based schema: a module receives a token signal whenever a dependency gets satisfied. When the controller has collected all the tokens (i.e., all dependencies are satisfied), it checks for resource availability. If the resource associated with the operation is free, execution starts. The approach does not introduce any communication overhead, because it does not use any sophisticated protocol. This activation mechanism, which allows as soon as possible execution, enhances parallelism exploitation for CFI specifications, specifications that include VLOs, and parallel function

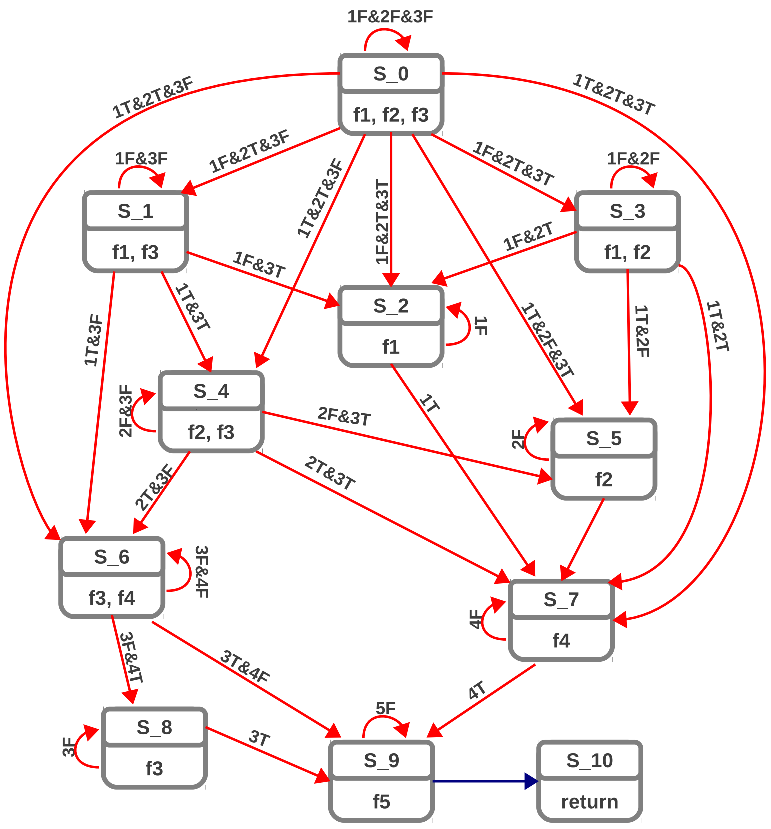


Figure 4.3: *Product STG for function* call_loops*; calls are represented by produced values.*

calls.

4.2 Related Work

The majority of HLS approaches generate a centralized FSM for controlling the data-path. However, these approaches do not efficiently deal with coarse grained parallel or VLOs specifications. Partial solutions identify and analyze all the possible behaviors at design time [118], and generate accordingly the FSM. Nevertheless, this significantly complicates the controller generation when there are many parallel control flows. Controller generation may not even be possible if worst cases for execution latencies are unknown. There are several alternatives to centralized FSMs. As detailed in Chapter 3, they include: distributed FSMs [165], parallel FSMs [105, 112], and hierarchical FSMs [80, 102]. So far, their main application has been reducing the critical path delay [51, 96]. The typical approach for designing distributed controllers is FSM decomposition. FSM decomposition is a top down approach, designed as a back-end process, that starts from a centralized machine and divides it in sub-machines [165]. The proposed approach does not use FSM decomposition. Instead, it relies on the generation of a set of communicating controllers. Parallel controllers enable concurrent execution of statically identified parallel code portions: several approaches implement parallel controllers starting from a description of the specification that already identifies the parallelism in the application, such as Petri nets [105, 112]. The proposed flow instead is able to synthesize C-code applications. [105] presents a formal controller decomposition methodology for a Petri net specification. The model represents a token-based architecture, but requires priority and synchronization schemes to share variables and to implement sub-controllers communication, increasing design costs. Hierarchical FSMs can activate, from the current FSM, either the same or another FSM, similarly to procedures in software programs. [102] presents an approach to generate HDL descriptions of hierarchical FSMs starting from hierarchical Petri net specifications. Some approaches that employs Hierarchical FSMs can also activate multiple FSMs in parallel [168], each one potentially exploiting different types of concurrency [80]. These solutions do not reuse functional units, require partitioning of the program to extract parallelism, and are able to support variable latency operations only with complex modifications. Furthermore, supporting variable latency operations still has the same limitations of a centralized FSM. The problem of supporting operations with variable latency is well known in the computer architecture community. Approaches for dynamic scheduling of independent operations dates back to the sixties, with scoreboarding and the Tomasulo algorithm [175]. [134] addresses the problem of the local scheduling for functional units with variable latency, proposing the design of a hardware scheduler that is also able to manage iterative components, if the number of iterations is limited. From the point of view of HLS, [113] proposes a "relative" scheduling formulation that supports operations with fixed and unbounded (variable) delays. However, the scheduling is still static: it defines the start times of operations as offsets from operations with unbounded delays. Del Barrio et al. [64, 65] present a controller design that enables efficient implementation of Speculative Functional Units (SFUs) in a data-path during HLS. Centralized FSMs, which employ a static execution schedule, cannot efficiently manage speculative units, because if a misprediction happens, they have to wait until obtaining the correct result, and cannot proceed with other parallel work, impacting the overall performance. The design proposed in [64, 65] adopts a pseudo-distributed approach, where a local controller for each SFU dynamically verifies if the speculation is correct by only checking the local state. The proposed methodology is more general: it can support SFUs, but also other types of variable latency operations, and it extracts parallelism from complex specifications. Sharp and Mycroft [166] survey the expressivity of current scheduling methods, and presents a new approach that automatically generates scheduling logic to dynamically resolve contention for shared resources. The authors then exploit static analysis techniques to remove redundant scheduling logic. The proposed approach supports dynamic scheduling through a distributed controller: the only static computation is the generation of the activation conditions that enable, at runtime, to start the execution of an instruction as soon as its dependencies have been satisfied. Moreira [133] introduces a technique to dynamically identify the conditions enabling concurrent execution of tasks that exploit parallelism at different levels of granularity. The approach operates on the Hierarchical Task Graph (HTG), an intermediate program representation that encapsulates the information on control and data dependencies. The proposed HLS approach considers a similar intermediate representation, and applies a similar technique to identify starting conditions at the instruction level. In [18], the authors propose an approach to support multimodal functional units, while minimizing the size of the resulting circuit. Multimodal FUs implement multiple types of operations, usually with different execution latencies. The approach still relies on the joint scheduling of mutually exclusive tasks. Thus, it still is static and does not aim at exploiting parallelism.

4.3 The Parallel Controller Architecture

The proposed HLS flow aims at generating accelerators that support dynamic scheduling through the construction of a parallel controller. In opposition to typical FSM-based approaches, the design methodology does not require the computation of a static schedule. The synthesized architecture is able to dynamically determine the execution order of the operations, according to the particular run-time conditions. This enables higher parallelism exploitation, because it potentially allows executing, every cycle, operations that are ready to run (i.e., that have all dependencies satisfied), independently from their effective execution delay and, more importantly, from the execution delay of the operations they depend on. This adaptive behavior is obtained by analyzing data and control dependencies in the specification, and by consequently identifying the *Activating Conditions* each operation is subject to. Activating conditions preserve execution correctness and identify when an operation is ready for execution, i.e., when all its dependencies have been satisfied. The flow combines the various Activating Conditions in logic functions, and then synthesizes them as hardwired control functions for each operation. The different Activating Conditions of each instruction are combined together through *join* and *or* operators, which have similar semantics of the boolean *and* and *or* operators. However, the *join* operator (Figure 4.5, 4.4) also considers a temporal element with respect to the boolean *and*.

```
join($in_1$, $in_2$, ..., $in_N$, $out$)
{
    $\forall i \in [1, N] state_i = in_i + state_i$

    $out = \prod_i state_i$

    $if(out)$
        $state = 0$
}
```

Figure 4.4: *N-inputs join module behavior: $state_i$ is high if the i-th token signal in_i has already been collected.*

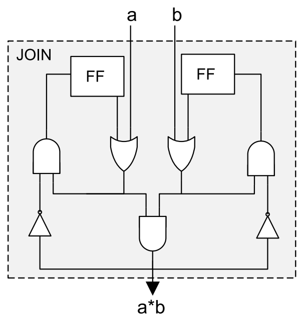


Figure 4.5: *2-inputs join module RTL schematic representation.*

The operands of the logic functions are token signal. They represent the completion of the execution of a previous operation, which identifies the satisfaction of a data dependency, or the outcome of a conditional operation, which allows verification of control dependencies. Figure 4.9 shows the dependence graph for function *call_loops* in Figure 5.2, annotated with the Activating Conditions. A dedicated module, named *Execution Manager* (EM), collects these tokens for each operation. Each EM:

Verifies when the execution of the operation can start This is obtained through the logic implementing the Activating Conditions. When the dependencies and resource constraints (as explained later) are satisfied, execution starts: if the operation is implemented through a functional unit, which requires an explicit start signal, the EM also generates the related token.

Manages execution temporization Three classes of operations are identified: fixed latency operations, zero delay operations and variable latency operations. The EM manages temporization of fixed latency operations through

```
RM( $req_1$ , $req_2$ , ... , $req_N$ ,
    $ack_1$ , $ack_2$ , ... , $ack_N$ )
{
    $ack_1$ = req_1

    $\forall i \in [2, N]$ $ack_i = req_i$ & !$ack_{i-1}$

}
```

Figure 4.6: *N-inputs Resource Manager behavior: requests are ordered according to their priority.*

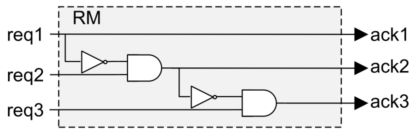


Figure 4.7: *3-inputs Resource Manager RTL schematic representation.*

simple counters. The EM considers zero delay operations (i.e., *chained* operations) completed at the same clock cycle they started in. Finally, the EM manages variable latency operations through explicit done signals. It considers a variable latency operation running, until it receives the done signal. The functional unit associated with the operation generates the done signal.

Sets the control signals Control signals are: selectors of steering logic and multi-modal FUs; start signals for FUs that require explicit activation; write enable signals for registers. At the opposite of FSM controllers, where the values of these signals depend on the current states, in the distributed architecture the EMs explicitly sets them.

Notifies execution completion The EM notifies execution completion through token signals. The EM associated with dependent operations collects the related token signals.

Checks for resource availability If an operation ready for execution is bound to a shared functional unit, the EM sends an execution request to a Resource Manager (RM). The RM is a lightweight arbiter associated with the shared resource and designed to manage concurrency (Figure 4.7, 4.6). If the resource is available, the RM replies with a notification that it accepted the request, so that execution can start. If multiple requests come at the same time, the RM chooses the one to accept, according to a priority ordering. If the shared resource is a variable latency unit, then the RM intercepts the done signal coming from the data-path, and forwards it to the EM associated with the operation currently running. RMs are implemented through simple combinational logic, thus they do not introduce additional clock cycles of delay. The resulting adaptive controller is composed of one EM for

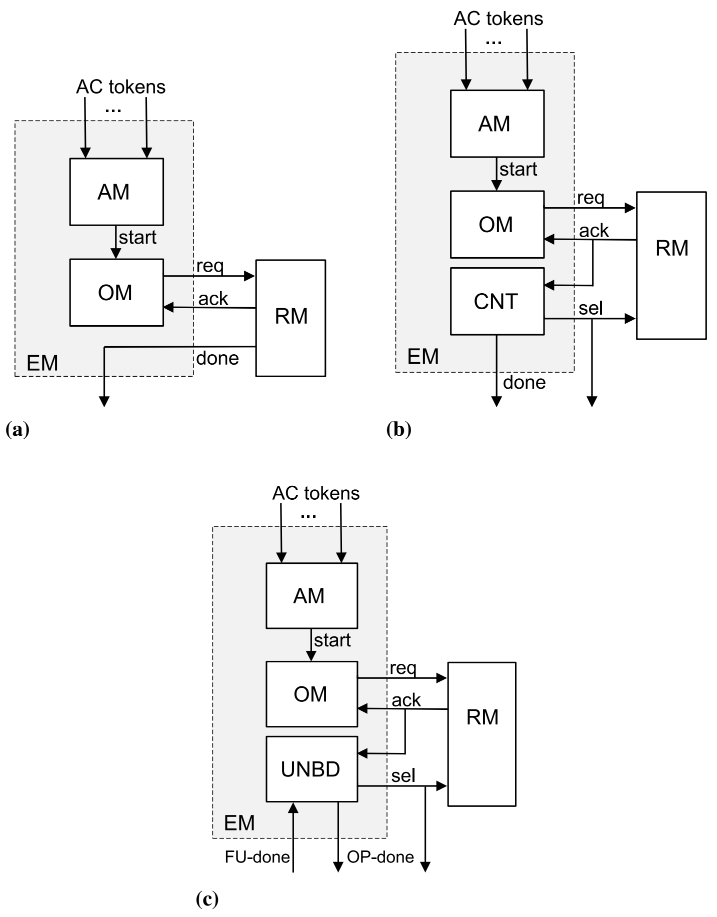


Figure 4.8: *Distributed Controller Modules: single-cycle(a), multi-cycle(b) and unbounded operations(c) Execution Managers (EM).*

each operation in the specification and of one RM for each shared resource. Thus, the complexity of the controller linearly grows with the number of operations.

Figure 4.8 shows schematic representations of the EMs associated with single cycle (Figure 4.8a), multi cycle (Figure 4.8b) and variable latency operations (4.8c). Each EM is composed of two modules: an Activation Manager (AM), which implements the Activating Conditions, and an Operation Manager (OM), which interfaces to RMs sending execution requests (*req* signals) until accepted. The OM sends execution requests one clock cycle after the AM activates it. Thus, it also manages temporization for single cycle operations. The RMs always notify resource availability within

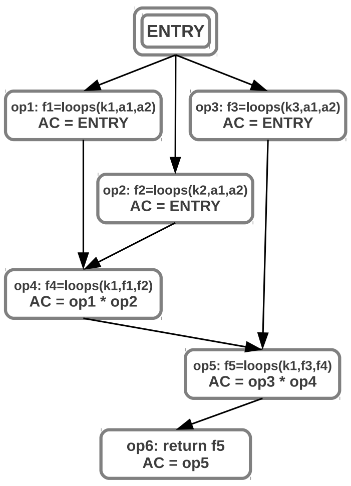


Figure 4.9: *Dependence graph for function* call_loops, *annotated with Activating Conditions.*

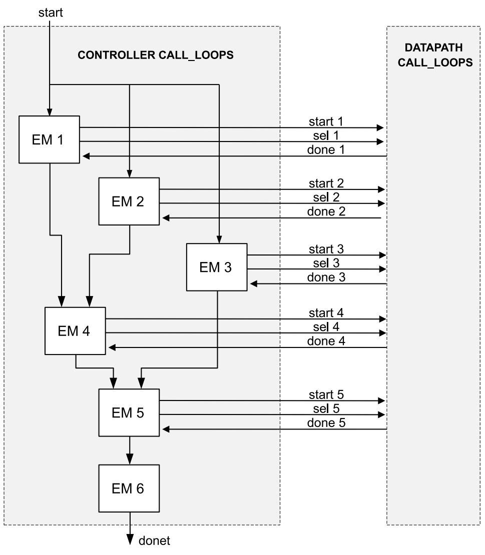


Figure 4.10: *Distributed controller architecture for function* call_loops.

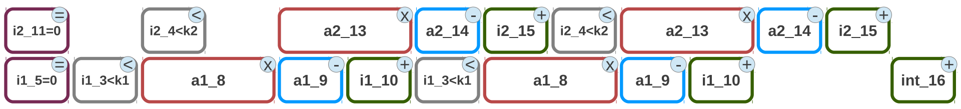


Figure 4.11: *Runtime behavior of adaptive controller: execution trace of function* loops *with* $k1 = k2 = 2$*, under resource constraint (one instance available for each kind of functional unit).*

the same cycle of the request. For multi cycle and unbounded operations, the EMs include CNT and UNBND modules, which presents similar behaviors. They are both activated when execution starts, and have the role of setting control signals and inhibiting the RM from accepting new requests, during the whole execution. The difference between the two modules is that the CNT directly states when execution completes (it is a counter), while the UNBD module collects done signals coming from the data-path. For chained operations, the EM degenerates in the AM.

The described architecture can independently manage operations and adaptively execute them as soon as possible. Figure 4.11 highlights the adaptive behavior of the controller, proposing an execution trace for function *loops* in Figure 5.2, characterized by two parallel loops. Notice that the number of iterations of both loops is statically unknown, thus most compilation processes will not perform neither unrolling nor loop collapsing,

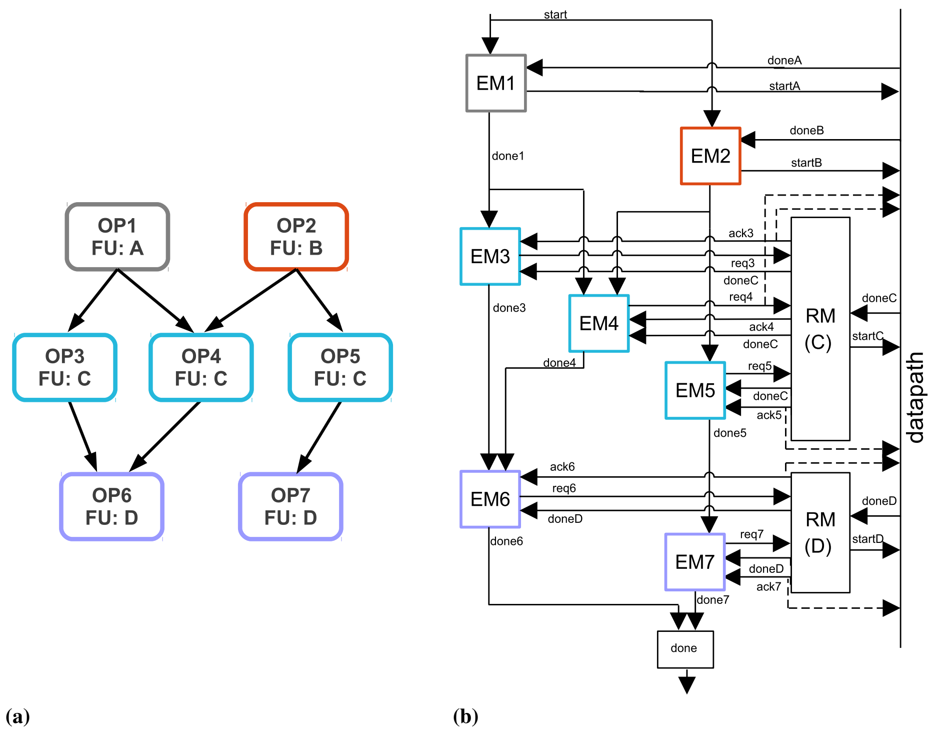


Figure 4.12: *Example specification (a) and corresponding parallel controller implementation. Operations 3, 4 and 5 share FU C, operations 6 and 7 share FU D.*

sequentializing the two portions of code. In this example, it is assumed that one resource for each kind of operations is available. All operations require one clock cycle, except for multiplications, which require 2 clock cycles. The architecture checks resource availability at runtime, executing in turn operations belonging to the two loops, as soon as shared resources become available. To obtain the same behavior, a static scheduling approach needs to merge the two loops. Instead, the proposed approach provides such behavior without any code transformation. Moreover, the as soon as possible execution is preserved also when introducing variability in operations delays, which most classical approaches fail to support without severe penalties in terms of controller complexity. For example, function *call_loops* presents concurrent function calls, whose latency is statically not known. The proposed approach allows running these function concurrently, with a low complexity architecture, as shown in Figure 4.10. It is remarked that obtaining the same behavior with a static scheduling requires the synthesis of a FSM characterized by 11 states and 34 transitions (Figure 4.3).

4.4 Adaptive Behavior

This section propose a simple illustrative example to highlight the adaptive behavior of the target accelerator design, able to rearrange execution order at runtime, even in the presence of VLOs. Figure 4.12a shows a simple specification (through its DFG) and Figure 4.12b the corresponding parallel controller architecture. Each operation is mapped on a variable latency functional unit: FUs C and D are shared among operations 3,4,5 and 6,7 respectively. The variability on execution latency and the concurrency on shared resources generate several feasible runtime schedules, which exploit the available parallelism. The parallel controller builds such schedules directly at runtime, as shown in Figure 4.13 through same examples. Operations 1 and 2 are associated to exclusively bound FUs, thus will al-

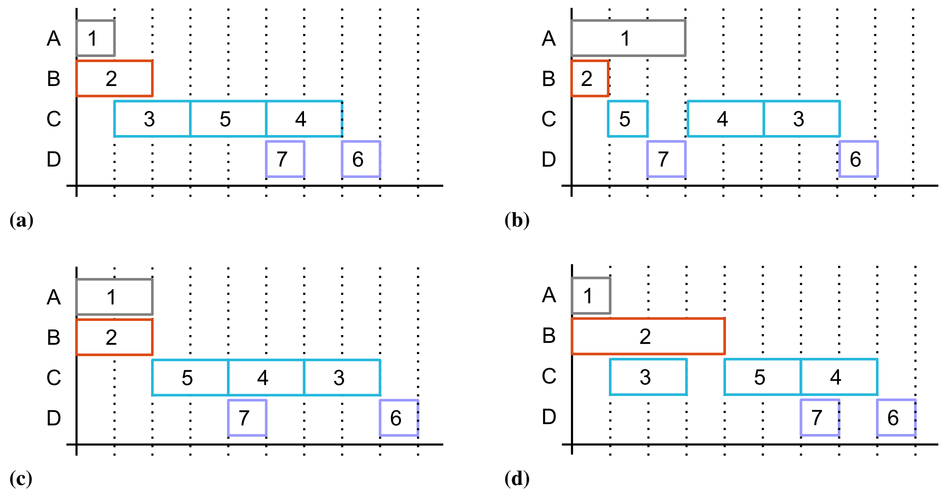


Figure 4.13: *Runtime schedules when considering dynamic variations on FU latencies. Assumed priority for RMs: op3<op4<op5; op6<op7.*

ways run concurrently. However, their latency, affect the execution order of the subsequent operations. If op1 completes firts, then op3 will execute before op4 and op5 (figure 4.13a). If op2 completes first instead, op5 will execute before the other operations mapped on FU C (figure 4.13b). If op1 and op2 complete execution in the same control step, then op5 will execute first, since it has been assigned the maximum priority (figure 4.13c).

4.5 Conclusions

This chapter presented a parallel controller architecture, for the design of adaptive accelerators featuring dynamic scheduling. Its modularity and regularity make suitable its adoption in HLS flows. The proposed architecture is able to exploit parallelism at different granularities, and provides natural support for variable latency operations, such as speculative operations, memory accesses and function calls. The next chapter will describe a complete C-based HLS flow for the automated generation of the proposed architecture, detailing the different algorithms designed to perform all the HLS tasks.

CHAPTER 5

High Level Synthesis of Adaptive Hardware Components

The parallel controller architecture, introduced in the previous chapter, allows the design of adaptive components featuring dynamic scheduling. The controller is composed of a set of interacting modules, each managing the execution of a particular operation or task, enabling as soon as possible execution once dependences are satisfied. This features provide efficient support for variable latency operation, poorly managed by statical approaches, and in general for coarse grained parallelism exploitation, allowing multiple execution flows to run concurrently. However, the automated synthesis of such components from a C-like specification introduces several challenges. To overcome some of these difficulties, previous approaches targeting parallel controllers considered particular specification languages, making them difficult to be adopted in practice. The generation of the controller itself is relatively trivial: what makes the synthesis process difficult is the absence of pre-computed schedule. In fact, most of the techniques proposed in literature for the data-path synthesis are based on the definition of an execution ordering among operations, and then address the HLS tasks under this assumption. In order to obtain a complete HLS flow, this work proposes novel algorithms for the automated synthesis of hardware components featuring dynamic scheduling [42] [46]. Such algorithms have been designed as general as possible, in order to possibly be applied on similar flows. For example, the liveness analysis algorithm [43] proposed in Section 5.4 can be adopted in any dynamic scheduling approach, since it enables the definition of conflict free bindings regardless of the runtime execution order. This chapter describes the resulting C-based HLS flow for the automated generation of the target architecture. The proposed techniques allowed the actual design of a complete HLS tool, developed extending the Bambu HLS framework. Section 5.1 overviews the proposed flow; Sections 5.2 describe the front-end steps, useful to support the synthesis process; Sections 5.3 and 5.4 describe the designed binding algorithms; Section 5.5 details the controller generation process; Section 5.6 explains how the designed flow supports complex code behaviors, which include Static Single Assignment (SSA) programs, nested loops and nested function calls. The input language coverage is one of the key contributions of this work: as afore mentioned, other approaches featuring parallel controllers or dynamic execution for parallelism exploitation, are not suitable for the HLS of C-code specifications, or reduce the domain of supported constructs. Section 5.7 summarizes the implementation effort, for the integration of the flow in the Bambu framework; Section 5.8 evaluates the effectiveness of the methodology on a set of common HLS benchmarks, on both performance and area. Section 5.9 concludes the chapter.

5.1 Proposed High Level Synthesis Flow

Figure 5.1 summarizes the proposed HLS flow. As in typical flows, it is possible to distinguish three different phases: front-end, synthesis and back-end. The front-end phase takes care of compiling the input specification: this allows exploiting compiler optimizations such as constant propagation and code motion techniques [95] [86]. The Front end also processes the intermediate code produced during compilation (Figure 5.2) to construct an Internal Representation (IR), which is then exploited to perform the HLS tasks. The synthesis process considers a graph IR that represents the program dependencies, i.e., data, control, and structural dependencies. Structural dependencies are introduced between memory accesses if they can access the same memory location, so to preserve the ordering between stores and loads. The produced IR also embeds control flow information, such as back-edges of loop constructs. This is required for the subsequent step, the identification of the Activating Conditions associated with each

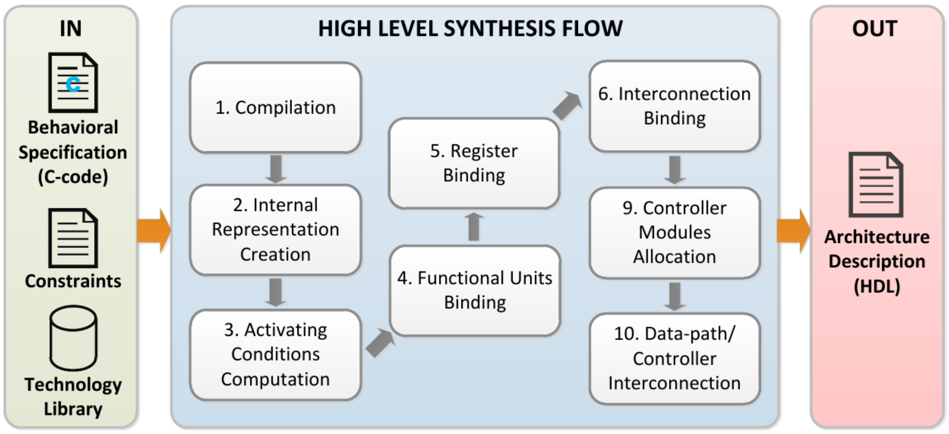

IN
Behavioral Specification (C-code)
Constraints
Technology Library
HIGH LEVEL SYNTHESIS FLOW
1. Compilation
2. Internal Representation Creation
3. Activating Conditions Computation
4. Functional Units Binding
5. Register Binding
6. Interconnection Binding
9. Controller Modules Allocation
10. Data-path/ Controller Interconnection
OUT
Architecture Description (HDL)

Figure 5.1: *Proposed High Level Synthesis flow.*

operation. The synthesis phase mainly consists of allocation and binding of FUs (modules), registers, and interconnections. Typical HLS techniques address these tasks starting from the computation of the scheduling of the operations. The next sections detail how FU and register binding tasks are addressed without scheduling information. Instead, interconnection binding is simply performed as a result of FU and register binding. Finally, the back-end phase produces the actual RTL implementation of the input specification.

```
int loops(int k1, int a1, int a2){
  int internal_16;
  int a2_14,a2_2,a2_13,a1_8,a1_9,a1_1;
  int i1_3,i1_5,i1_10,i2_4,i2_15,i2_11;
  i1_5=0;
  i2_11=0;
  BB_LABEL_4:
  a1_1 = gimple_phi(a1, a1_9);
  i1_3 = gimple_phi(i1_5, i1_10);
  if (i1_3 < k1){
    a1_8 = a1_1 * i1_3;
    a1_9 = a1_8 - k1;
    i1_10 = (int)(i1_3 + (1));
      goto BB_LABEL_4;
  }
  BB_LABEL_7:
  a2_2 = gimple_phi(a2, a2_14)
  i2_4 = gimple_phi(i2_11, i2_15)
  if (i2_4 < k1){
    a2_13 = a2_2 * i2_4;
    a2_14 = a2_13 - k1;
    i2_15 = (int)(i2_4 + (1));
      goto BB_LABEL_7;
  }
  internal_16 = (int)(a1_1+a2_2);
  return internal_16;
}

int call_loops(int k1, int k2, int k3,
               int a1, int a2)
{
  int f5_10, f2_6, f3_8, f1_4, f4_9;
  f1_4 = loops(k1, a1, a2);
  f2_6 = loops(k2, a1, a2);
  f3_8 = loops(k3, a1, a2);
  f4_9 = loops(k1, f1_4, f2_6);
  f5_10 = loops(k1, f4_9, f3_8);
  return f5_10;
}
```

Figure 5.2: *Example C specification, after the compilation step.*

5.2 Compilation and IR Generation

The first step of the front-end phase is the compilation process. In the designed flow, this task is performed interfacing with a software compiler (e.g. GCC), which restructures the input code through different optimizations. The code obtained after the optimization process, as shown in Figure 5.2, is written in Static Single Assignment (SSA) form. SSA improves several optimization techniques based on data-flow analysis, such as constant folding and dead code elimination [19]. The optimized code is then processed to generate an appropriate Internal Representation.

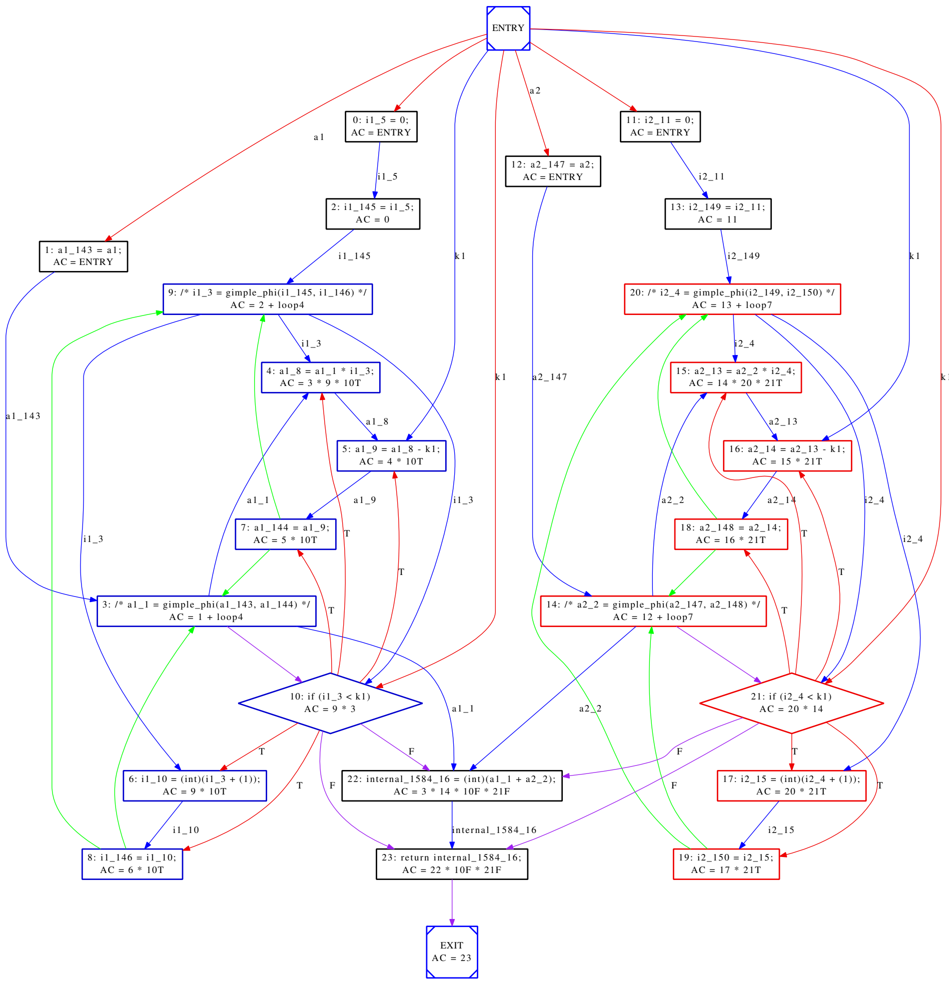


Figure 5.3: *Extended program Dependence Graph for function* call loops, *annotated with Activating Conditions. Blue edges denote data dependences, red edges control dependences, green edges backward control flow dependes, purple edges forward control flow dependencies.*

5.2.1 Extended Program Dependence Graph

The synthesis process considers as input an IR which highlights both data and control dependencies in the program: the *Extended Program Dependence Graph* (EPDG) [122]. The EPDG is a direct graph $G(V, E)$, where V denotes the set of operations in the program, and where each edge $e \in E$ represents a dependence between its source and target operations, thus defining a precedence relation on their execution order when both source and target must be executed. The EPDG *extends* the Program Dependence Graph: in addition to data and control dependencies, directly identified through typical flow analysis, it considers Control Flow Dependencies (CFDs). CFDs are introduced to provide useful information on properties which must be preserved during execution of loop constructs. Two main classes of CFDs exist:

1. **Backward CFDs**. Let $First(L)$ be the set of operations belonging to loop L, which can be executed first in the loop, and $Last(L)$ the set of operations which can be executed last. These sets can be easily computed through dependence analysis; in the general case, their cardinality can be greater than 1, especially for set $Last(L)$. In case of nested loops, these sets also consider operations belonging to inner and outer loops. Backward CFDs are defined as edges $e(i, j)$, with $i \in Last(L)$ and $j \in First(L)$: they are introduced to identify loop entry and exit points.

2. **Forward CFDs**. Given a loop-exit operation i for loop L, and operation $j \notin L$ control-independent with respect to i, a Inter-loop forward CFD is defined between i and j if there exist an operation $z \in L$ such that j is data-dependent on z. Such flow dependences are also inserted if j is a return statement, and after the definition of the other CFDs j is still not reachable from i.

Figure 5.3 shows the EPDG obtained for function *call_loops* in Figure 5.2.

5.2.2 Activating Conditions Computation

Once the EPDG is built, it is analyzed in order to identify, for each operation, the corresponding Activating Conditions. ACs identify which conditions must be satisfied at runtime to enable the execution of each operation. The first formulation of the concept of ACs is proposed in [81], considering only data and control dependences. [122] extends such formulation, considering also CFDs. In this flow, the latter is considered. Activating conditions are composed of 3 parts, and defined as:

$$AC(i) = AC_{data}(i)\ join\ AC_{control}(i)\ join\ AC_{control-flow}(i). \qquad (5.1)$$

Each component corresponds to a particular kind of dependencies, and it is in turn represented as a logic function. For example, the *control_flow* component allows to support loops, and usually occurs in the form $AC_{control_flow}(i) = init_i + loop_i$, where $init_i$ is the condition which enable the execution of i associated to the first iteration of the loop, and $loop_i$ identifies when the execution must be-repeated in subsequent iterations. The component $loop_i$

denotes the join of all the operation belonging to the set $Last$ of the given loop. Notice that this component occurs only for operations belonging to the set $First$, as shown in Figure 5.3.

5.3 Module Binding

In typical approaches, operations that execute concurrently, according to the pre-determined schedule, are not allowed to share the same hardware resource. This avoids resource conflicts. In the proposed flow, instead, RMs dynamically resolve resource conflicts at runtime. The approach avoids conflicts for any possible FU binding and for any possible runtime execution order, thus module binding does not influence the correctness of the accelerator's behavior. Binding is performed through a clique covering algorithm [178] on a weighted compatibility graph. The compatibility graph is built by identifying the operations that can profitably share the same hardware resource, analyzing the dependences graph. The algorithm assigns weights taking into account area/delay trade offs as a result of sharing; for example, FUs that demand significant area will be more likely shared. However, weights assignment is tuned with the aim of enhancing performance: for example, function calls which are not conflicting (thus they may run concurrently), are less likely to share hardware resources, even if sharing is profitable in terms of area. The flow also considers the cost of interconnections for introducing steering logic, both in terms of area and frequency. This is required to meet timing constraints. In case of multiple concurrent execution requests, RMs adopt the topological ordering of the operations sharing the FU on the IR graph as a priority ordering. This approach does not impose an execution order, because it does not introduce any structural dependence: given two operations bound to the same resource, if one of them is ready for execution before the other one, it starts execution first regardless of the assigned priority. This characteristic is highlighted in the execution traces proposed in Figure 4.11 and Figure 4.13.

5.4 Liveness Analysis and Register Binding

Common HLS approaches acquire liveness information assuming a partial ordering relation between operations, obtained by means of a static schedule [24] [32]. Liveness information is then captured in the form of a Conflict or Compatibility Graph, and register binding often addressed as a graph coloring problem [48] [169]. Such techniques, where register binding exploits the scheduling task results, are known as *post-scheduling* approaches. On the other side, in *pre-scheduling* approaches allocation is performed before scheduling: in this case minimizing the number of allocated registers may lead to the introduction of false dependencies in order to avoid resource conflicts. Despite of to the absence of a pre-computed schedule, proposed register binding methodologies for dynamic scheduling architectures are usually designed as *post-scheduling* approaches. Examples are given in [65] and [64], where the authors propose a distributed controller for managing Speculative Functional Units (SFUs) in HLS. SFUs exploits a predictor for the carry signal, thus execution order may vary according to the runtime prediction. Data-dependent operations will wait until the needed results are correctly computed. Figure 5.4(a) provides an example

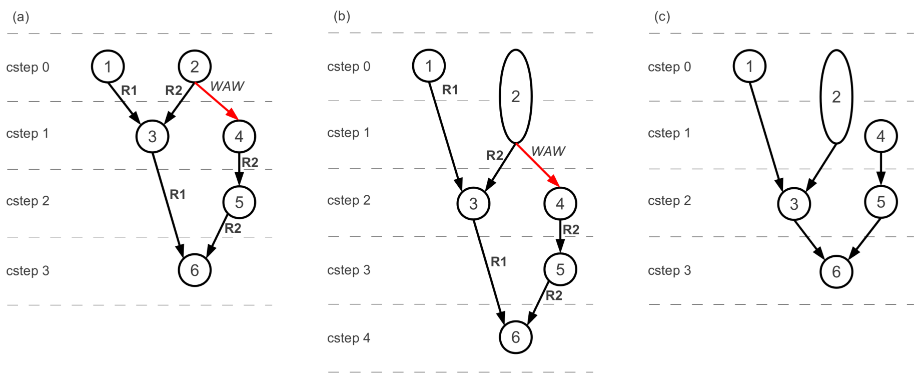


Figure 5.4: *Scheduled CDFG (a), runtime schedule in the case of misprediction for SFU 2 (b) and runtime schedule in the case of misprediction for SFU 2, without the resource constraint between nodes 2 and 4.*

scheduled Data Flow Graph (DFG): since the results of SFUs 2 and 3 are bound to the same register R2, a Write After Write dependency is introduced between them. If at runtime all the predictions will be right, the runtime schedule will reflect the static schedule (4 control steps). On the contrary, in the case of misprediction on SFU 2, the inserted dependency will delay the program execution (assuming a penalty of one control step) as shown in Figure 5.4(b). Figure 5.4(c) shows the resulting runtime schedule without introduced resource constraint. In order to reduce the impact of structural dependencies, in [65] and [64], the register binding based on the static schedule has been customized by means of a Least Recently Used Register binding policy, which tries to bind different registers for close in time operations. However this approach cannot provide a conflict free register assignment. The binding approach based on the liveness analysis proposed in this work instead, completely avoids any runtime resource conflict, associating operations that may be executed concurrently to different registers. A similar idea has been introduced in [156], where register allocation is performed through the coloring of a *parallelizable conflict graph* which avoids the insertion of false dependencies. The parallelizable conflict graph is built starting from a pre-computed conflict graph, adding edges between storage values defined by possibly concurrent operations. Such approach is heavily influenced by the initial conflict graph construction, which still requires a statically defined ordering relation among operations. Obviously this requirement induces severe restrictions to the number of feasible schedules: a schedule which violates the previously selected ordering may lead to incorrect execution. On the contrary, the register binding based on the proposed methodology guarantees correctness without performance loss for *any possible* runtime schedule.

5.4.1 Preliminary Notions and Definitions

Flow Graphs Denoting with V the set of vertices and with E the set of edges, a node $v \in V$ of a graph $G(V, E)$ has *out edges* that lead to *successor* nodes and *in-edges* that come from *predecessor* nodes. The set *out(v)* represents the set of *out edges*, while *in(v)* represents the set of *in edges*. Moreover $pred(v) \subset V$ denotes the set of predecessors of node v and $succ(v) \subset V$ the set of successors of v. Given $e \in E$, $source(e) \in V$ represents the source node of e, while $target(e) \in E$ represents the target node. A *directed path* $p = a \to b$ is a sequence of edges $e_0, e_1, ..., e_n$ such that $source(e_0) = a, source(e_1) = target(e_0), \dots , target(e_n) = b$.

Uses and Defs Representing with VAR the set of variables in a subprogram P, and with $G(V, E)$ a graph representation of P: the set $def(x) \subset V$ of a variable $x \in VAR$ is the set of graph nodes that define it; the set $use(x) \subset V$ of a variable $x \in VAR$ is the set of graph nodes that use it; the set $def(v) \subset VAR$ of a graph node $v \in V$ is the set of variables that it defines; the set $use(v) \subset VAR$ of a graph node $v \in V$ is the set of variables that it uses.

Liveness Analysis and Data-Flow Equations A variable $x \in VAR$ is said to be *alive on edge* $e \in E$ if $\exists$ a directed path p from e to a node $b \in use(x)$. A variable $x \in VAR$ is *live-in* at node $v \in V$ if $\exists e \in in(v)$ such that x is alive on e; similarly $x \in VAR$ is *live-out* at node $v \in V$ if $\exists e \in out(v)$ such that x is alive on e. *Live-in* and *live-out* sets capture liveness information useful to perform register allocation and binding. Such information is obtained from *use* and *def* through *Data-Flow equations*:

$$live\_in(v) = use(v) \cup (live\_out(v) \setminus def(v)) \tag{5.2}$$

$$live\_out(v) = \bigcup_{s \in succ(v)} live\_in(s) \tag{5.3}$$

Liveness analysis is performed iteratively solving the Data-Flow (DF) equations, until the least fixed point is reached [19].

5.4.2 Schedule-Independent Liveness Analysis

DF equations as formulated in 5.2 and 5.3 require a graph representation of the specification characterized by a single execution flow, which reflects a given execution order. As discussed in the previous sections, for dynamic scheduling architectures such an order is not statically computed. Thus in these cases, a single flow representation will not adequately represent the run-time execution order, and standard liveness analysis will produce inaccurate information. However, even if a static schedule is not available, it is always possible to partially define execution order according to the dependencies between operations. If operation a depends on operation b, then a must be executed after b. A convenient way to represent such relations is the EPDG, since it captures information on control dependencies, data dependencies and control flow. If there exist a path $p = x \to y$ between two nodes x and y of the EPDG, then y will be executed after x. Such a relation is not defined for all the possible pairs of nodes: if such a path does not exist, then it is not possible to state, simply analyzing the EPDG, which operation will be executed first; obviously in this case it is also not possible to establish if the two operations will be executed concurrently.

In Figure 5.6(a) the results obtained through DF liveness analysis on an example EPDG (Figure 5.5) are shown. Analyzing the live-in/live-out sets it results that variables $a1$ and $b1$ are never simultaneously alive. According to this result, these variables may be mapped on the same physical resource, since their life intervals do not overlap, not ensuring correctness for all the possible execution orders. For example, if operations 2 and 3 complete their execution in the same control step, then $a1$ and $b1$ cannot be stored in the same physical register. Notice that for nodes 2 and 3 it is not possible to compute an order relation. Such nodes can be denoted as *parallel* nodes because they may be executed concurrently. It is possible to define a binary relation $||$ among the nodes of the EPDG: given two nodes a and b, $a||b$ iff there is not a path $p = a \to b$ or $p = b \to a$, i.e. they are *parallel*. In order

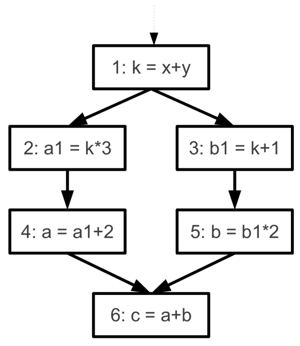


Figure 5.5: *Example Dependencies Graph.*

(a)	(b)
Live-out(6) = { }	Live_p(6) = { }
Live-in(6) = {a, b}	
Live-out(5) = {a, b}	Live_p(5) = {a, b, a1, k}
Live-in(5) = {a, b1}	
Live-out(4) = {a, b}	Live_p(4) = {a, b, b1, k}
Live-in(4) = {b, a1}	
Live-out(3) = {a, b1}	Live_p(3) = {a, b, a1, k}
Live-in(3) = {a, k}	
Live-out(2) = {b, a1}	Live_p(2) = {a, b, b1, k}
Live-in(2) = {b, k}	
Live-out(1) = {k, a, b}	Live_p(1) = { }
Live-in(1) = {x, y}	

Figure 5.6: *Liveness analysis results solving standard DF equations (a), $live_p$ sets(b), for the example DG in Figure 5.5.*

to guarantee correctness for liveness computation, even in the presence of parallel nodes, two sets are defined:

$$parallel(x) = \{y | \nexists \textit{ a path } p = x \to y \text{ and } \nexists \text{ a path } p = y \to x\} \quad (5.4)$$

i.e. the set of *parallel* nodes with respect to node *x*, and

$$live_p(n) = \bigcup_{s \in parallel(n)} live\_in(s) \cup live\_out(s)$$

Sets $live_p(n)$ are introduced in order to take in account interferences between variables alive at *parallel* nodes. The main reason for keeping these sets separated from $live\_in(n)$ and $live\_out(n)$ is that this will facilitate the construction of the *conflict graph* representing the interferences between storage values. The introduced sets are computed when the DF equations (as in 5.2 and 5.3) have been already solved. Figure 5.6(b) reports $live_p$ sets for the previous example. However, there is another issue to be considered, concerning *death* of variables. In the proposed example, variable k results *dead* on exit on both nodes 2 and 3, since there are no uses of k reachable from these nodes. Nevertheless, k cannot be considered dead until both 2 and 3 are completed; the equation defining $live_p$ is modified in order to consider this aspect:

$$live_p(n) = \bigcup_{s \in parallel(n)} live\_in(s) \cup live\_out(s) \cup dead(s,n) \qquad (5.5)$$

where

$$dead(s,n) = \{x | \nexists \text{ a path } n \to u, u \in use(x), x \in use(s)\},$$

i.e. $dead(s,n)$ is the subset of $use(s)$ of variables that are not live out at s, but are used on a parallel node. Applying these equations, the sets $live_p$ for nodes 2 and 3 now include variable k. Even if the computation of $live_p$ sets allows a conservative construction of a conflict graph, the DF equations, in their standard formulation, produce over-conservative sets. For instance, in the considered example variables a and b results alive on exit at node 1, even if they have not already been defined, i.e. there is not a definition of a and b reaching node 1. To avoid this issue, DF equations can be modified as follows:

$$live\_in(n) = use(n) \cup (out(n) \setminus def(n)) \qquad (5.6)$$

$$live\_out(n) = \bigcup_{s \in succ(n)} in'(s)\ , in'(s) \subset in(s) \qquad (5.7)$$

where

$$in'(s) = \{x | \exists \text{ a path } d \to n, d \in def(x)\}$$

i.e. $in(s)$ is the subset of $in(s)$ of variables whose definition reaches node n. The resulting sets, for the considered example, are shown in Figure 5.7. Summarizing, in the proposed approach liveness information is obtained iteratively solving the modified DF equations, as formulated in 5.6 and 5.7, and then computing $live_p$ sets as in 5.5, which characterize the relations between variables alive at parallel nodes.

Mutual Exclusiveness Parallel nodes have been described as nodes such that there is not a path between them in the EPDG. According to this definition, operations belonging to mutually exclusive branches will be detected as parallel. This can be avoided refining the set $parallel(x)$, defined in 5.4 as follows: if node x lays on the i-th branch of a n-ary branch node d, then

(a)	Live-out(6) = { }	(b)	Live_p(6) = { }
	Live-in(6) = {a, b}		
	Live-out(5) = {b}		Live_p(5) = {a, a1, k}
	Live-in(5) = {b1}		
	Live-out(4) = {a}		Live_p(4) = {b, b1, k}
	Live-in(4) = {a1}		
	Live-out(3) = {b1}		Live_p(3) = {a, a1, k}
	Live-in(3) = {k}		
	Live-out(2) = {a1}		Live_p(2) = {b, b1, k}
	Live-in(2) = {k}		
	Live-out(1) = {k}		Live-out_p(1) = { }
	Live-in(1) = {x, y}		

Figure 5.7: *Proposed liveness analysis results for the example DG in Figure 5.5.*

any node y, laying on the j-th branch of d ($j \neq i$), is removed from the set $parallel(x)$. Mutually exclusive nodes can be detected considering the control dependencies in the EPDG.

5.4.3 Conflict Graph Creation

Liveness analysis allows the definition of an interference relation among the storage values: two variables *interfere* if their life-intervals overlap, and thus they cannot be mapped on the same resource. Such a relation is usually represented by means of a *conflict graph*. Overlapping life-intervals may be detected looking at live-out sets. Given a node v in the EPDG, storage values belonging to $live\_out(v)$ *interfere* with each other, since the associated variables are simultaneously alive. Equations 5.6 and 5.7 provide liveness information about *depending operations*, i.e. operations whose execution order is well defined. Equation 5.5 instead, provides liveness information about *independent operations*, i.e. operations whose execution order is not statically defined. Storage values belonging to $live_p(v)$ *may not interfere each other*; on the contrary each storage value $x_p \in live_p(v)$ interfere with each storage value $x \in live\_out(v)$. This property avoids the insertion of unnecessary interferences. Consider as an example the EPDG in Figure 5.6: it results $2||3$, $2||5$, $4||3$, $4||5$. Set $live_p$ for node 2 contains both b and $b1$, but b and $b1$ will never be simultaneously alive. Thus, while storage values belonging to $live\_out(v)$ will be *cliqued* in the conflict graph, storage values belonging to $live\_out(v)_p$ will not. This is the reason for keeping these sets separated. In Figure 5.8 the resulting Conflict Graphs for the considered example are shown for both separated and unified $live\_out$ sets: in the latter case storage values k, a, $a1$, b, $b1$ are cliqued, and 5 registers are needed to hold the program variables, while in the first case 3 register are needed. Once the Conflict Graph is built, register binding may be addressed

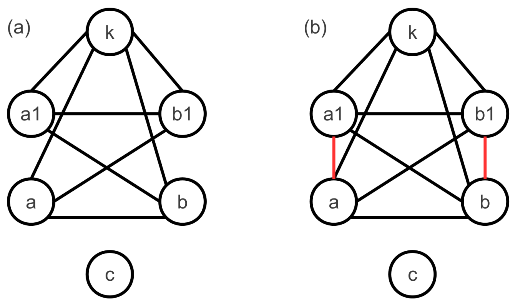


Figure 5.8: *Conflict Graph obtained for the previous example, keeping $live\_out$ and $live_p$ sets disjoint (a) and Conflict Graph obtained unifying $live\_out$ and $live_p$ sets (b).*

through standard techniques such as vertex coloring algorithms.

5.4.4 Algorithm Evaluation

In order to validate the described methodology, a register allocator based on the proposed liveness analysis has been implemented, and integrated in the *Bambu* HLS flow (see Section 2.4. Such allocator performs the register binding task by means of coloring an interference graph. The algorithm has been evaluated on a set of common HLS benchmarks: *crc32*, *ethernet*, *gcd* and *sha1* are from [4], *matmul* and *cftmdl* are from [120], while the others are from [170]. The synthesis targeted the parallel controller architecture. All the considered applications have been generated following two approaches: adopting the proposed register binding, and allocating a register for each storage value (*unique binding*). This comparison is motivated by the fact that unique binding always produces a conflict free binding, thus not impacting on performance regardless of the runtime schedule. For the Functional Units (FUs) instead, the same binding has been adopted for both the approaches, in order to isolate the effects of the register binding. FUs sharing is managed through dedicated logic, without introducing false dependencies imposing serializations. Memory access operations are not allowed to execute concurrently. The resulting designs have been simulated by means of ModelSim SE-64 6.6d [8]. For each benchmark, both the approaches led to the same execution latency (in terms of clock cycles): this result confirms the proposed methodology to produce a conflict-free binding.

Synthesis Results

The approach has been compared with three different register binding algorithms: unique binding, left edge and standard vertex coloring based binding.

Table 5.1: *High Level Synthesis results: Register Binding.*

	Conflict-Free		**Non Conflict-Free**			
Bench	**# Reg Prop.**	**# Reg Unique**	**# Reg Coloring**	**# Reg Left Edge**	**Gain vs Unique**	**Loss vs Col & LE**
crc32	13	19	7	7	31.6 %	85.7 %
ethernet	13	42	8	8	69.0 %	62.5 %
gcd	15	37	9	9	59.5 %	66.67 %
sha-1	32	161	15	15	80.1 %	113.3 %
cftmdl	39	72	14	14	45.8 %	178.5 %
chem	176	341	176	176	48.4 %	none
dir	65	121	63	63	46.3 %	3.1 %
lee	27	72	8	8	62.5 %	237.5 %
matmul	16	28	16	16	42.9 %	none
mcm	46	94	30	30	51.1 %	53.3 %
Total					55.2 %	27.7 %

Table 5.1 compares the number of allocated registers. Moreover the data-paths designs obtained through proposed algorithm, unique binding and vertex coloring, have been synthesized by means of Synopsys Design Compiler [11], using Nangate 45nm Open Cell Library [9]. Tables 5.2 and 5.3 report the obtained results, indicating non-combinational (*SEQ* columns), combinational (*CMB*), interconnection (*CON*) and overall (*TOT*) area costs in terms of library units. Percentage average gains adopting the proposed approach are also shown.

Comparison with unique binding

The provided High Level Synthesis results show that the proposed binding allocates 55.2% fewer registers, on average, when compared with the unique binding, that is the only other conflict free strategy. This result has been confirmed by the synthesis experiments, that reported a 55.8% average reduction in non-combinational area. In addition, despite the introduction of steering logic due to the register sharing it is also observed, an average reduction for both the combinational and interconnection parts: 0.48% and 6.92% respectively.

Comparison with non conflict-free approaches

In the conducted experiments, left edge and vertex coloring have been considered as non conflict-free approaches. In fact, they exploit standard liveness analysis, where the ordering relation for solving the DF equations is dictated by the scheduling task. As a result, on the contrary with respect to unique and proposed bindings, for these approaches parallelism exploitation may be limited by resource conflicts introduced to ensure correctness.

Table 5.2: *Design Compiler Synthesis Results: comparison against unique binding.*

	Conflict-Free								
	Proposed Approach				Unique Binding				
Bench	**SEQ**	**CMB**	**CON**	**TOT**	**SEQ**	**CMB**	**CON**	**TOT**	**Gain**
crc32	2189	5125	1716	9032	3262	5188	1903	10354	12.8%
ethernet	2189	5504	2040	9734	6837	5520	2747	15104	35.5%
gcd	2681	4899	2250	9831	6094	4796	2739	13630	27.8%
sha-1	5720	53394	14952	74067	28779	55310	19095	103184	28.2%
cftmdl	6971	32557	9681	49209	12870	32812	10675	56358	12.7%
chem	31460	486579	112141	630180	61133	487025	116870	665028	5.2%
dir	11619	156088	36421	204128	21629	156654	38171	216454	5.7%
lee	4826	30320	8447	43594	12870	30583	9763	53216	18.1%
matmul	2860	43901	10078	56839	5005	44026	10454	59486	4.45%
mcm	8222	89471	21846	119540	16803	90330	23491	130624	8.48%

Table 5.3: *Design Compiler Synthesis Results: Comparison Against Vertex Coloring.*

	Conflict-Free				Non Conflict-Free				
	Proposed Approach				Vertex Coloring				
Bench	**SEQ**	**CMB**	**CON**	**TOT**	**SEQ**	**CMB**	**CON**	**TOT**	**Gain**
crc32	2189	5125	1716	9032	1251	7880	2168	11300	20.1%
ethernet	2189	5504	2040	9734	1430	5874	2045	9350	-4.11 %
gcd	2681	4899	2250	9831	1609	4600	1991	8200	-19.9%
sha-1	5720	53394	14952	74067	2681	86527	21469	110678	33.1%
cftmdl	6971	32557	9681	49209	2502	31114	8505	42121	-16.83%
chem	31460	486579	112141	630180	31460	486817	112235	630512	0.05%
dir	11619	156088	36421	204128	11261	156444	36455	204160	0.02%
lee	4826	30320	8447	43594	1430	29774	7747	38951	11.9%
matmul	2860	43901	10078	56839	2860	44020	10122	57002	0.3%
mcm	8222	89471	21846	119540	5362	89343	21399	116104	-2.9%

In the considered settings, the static schedule for solving the DF equations has been computed through a list based scheduling approach [77]. Table 5.1 shows that left edge and vertex coloring lead to allocation of the same number of registers, that is the optimal one for the considered schedules. These more aggressive register sharing techniques allocate on average 21.7% less registers when compared with the proposed one, and 64.9% less registers with respect to unique binding. In two cases (*chem*, *matmul*) the proposed algorithm is able to obtain the same number of registers of left edge and vertex coloring. However it must be taken into account that, since these strategies do not consider the impact of interconnections [50] [94], massive sharing may not always lead to better results in area. This guess is confirmed by the synthesis results for the vertex coloring based data-paths, for which it is registered an impact in terms of combinational and interconnection costs: adopting the proposed approach provides an area reduction for both the components (3.67% and 2.03% on average respectively). The effects of the multiplexers allocation arise more clearly considering, for example, the results for the *sha-1* benchmark, where vertex-coloring allows a very aggressive register sharing. In this case it is reported an area reduction of 38.29% for the combinational part, and 30.35% for the interconnections. The greater number of allocated registers for results in a non-combinational area increase of 27.31%: nevertheless the overall results, including combinational and interconnection costs, demonstrate an overall average area reduction of 1.81%. In the two designs in which the same number of registers is used (*chem* and *matmul*), the area results are nearly the same for both the approaches. These results demonstrate that while providing a conflict free register binding, the proposed approach is not affected on average by area overheads.

5.5 Controller Generation

The high regularity of the parallel controller architecture facilitates the automated synthesis process. It allocates a RM for each shared resource, according to the module binding results. Then, it traverses the graph IR, instantiating an EM for each operation. OM, CNT and UNBD modules are allocated directly from the resource library; AMs, instead, are custom synthesized for each operation, according to its Activating Conditions. To simplify this process, ACs are encoded with the following simple grammar:

```
expr -> expr op expr
expr -> TOKEN_SIGNAL
op -> JOIN
op -> OR
```

where token signals and operators are terminals. Conditional operations require the allocation of an additional component, which tokenizes the output of the comparators. More in detail, this component samples the results of comparisons only when the corresponding operation is running, producing single token signals. The number of outputs of this component is equal to the number of branches of the conditional operations, each one denoting a specific control condition.

5.6 Support for Complex Behaviors

Nested Control Constructs In the typical FSM-based designs, the current state of the controller completely represents the state of the whole accelerator. At the opposite, in the proposed distributed architecture, all the states of each EM constitute the overall state of the accelerator. In a given control step, the state of an EM also accounts for the token signal already collected. These aspects introduce some challenges when, to guarantee correctness, it is required to reconstruct the global state of the accelerator. This happens, for example, in the case of nested control constructs. Consider a simple Activating Condition $AC(x) = data1 * cond1 * cond2$, with operation x belonging to a loop body. If, in a certain iteration, $cond1$ and $data1$ are satisfied and $cond2$ is not, in the next iteration x will start as soon as $cond2$ is satisfied, even if $cond1$ and $data1$ are not. Thus, the architecture detects all these situations to obtain a correct behavior. Every time a new iteration of a loop starts, it resets all the components that perform the join operations for the Activating Conditions to their initial state. In addition, in case of loop nests, it resets the modules that manage the inner loops, both when they terminate and the control returns to higher level loops, and when outer loops perform a new iteration.

SSA Programs During compilation, the input specification is translated into a Static Single Assignment (SSA) form. This facilitates and enhances both analysis and optimization steps. For example, SSA translation is usually able to reduce the life time of variables, allowing more aggressive register sharing. Once translated in SSA form, the code may present some particular operations, the *phi* functions. *Phi* functions are introduced if the original code assigns variables under different control conditions. An example can be found in Figure 5.2, which assigns variable $i1\_3$ both outside (line 5) and inside (line 13) the first loop, leading to the introduction of the *phi* function (line 9). The main issues in supporting SSA programs with a dynamic scheduling approach arise from the absence of the concept of transition, which is exploited to establish the value of each phi function. This limitation is overcome by analyzing the dependence graph. Two kind of phi operands are identified: operands whose assignment is subject to control dependencies, and operands assigned inside a loop body. In the first case, the results of phi functions are written as soon as the assignment is performed. In the second case, if the phi occurs in the loop itself, then the result is written when the loop body completes a given iteration; otherwise it is managed as in the previous case. As previously mentioned, the presence of phi functions also affects the register binding. The liveness analyzer, as proposed in [43] and described in Section 5.4, has been tuned as follows: given an operation $x = phi(x1, x2)$, x is considered defined in both $x1$ and $x2$ definition points. This does not increase the number of conflicts, because phi functions usually are defined with operations in mutual exclusion, which in turn are treated as non conflicting.

Hierarchical Designs In the proposed HLS flow function calls are considered as variable latency operations, mapped on custom Functional Units. The flow can automatically generate such FUs from the specification, or include them as custom components in the input component library. Another option is to flatten the design by exploiting function inlining in the compilation process. However preserving the hierarchical structure of the specification is often more profitable, since modularity presents three main advantages. The first is that the resulting architecture can also extract parallelism at the function call level. The second is that it reduces the complexity of the final architecture, by partitioning the design. Finally, it greatly enhances reusability: each function is synthesized once, even if called several times.

5.7 Implementation Details: Integration in the Bambu Framework

As mentioned in Section 2.4, from the software design point of view Bambu is characterized by an extreme modularity, implementing each HLS algorithm separately: the overall synthesis process is obtained as a composition of such algorithms. The resulting flow acts on different IRs depending on the synthesis stage, and on the selected algorithms. Such selection may be tuned through configuration files or command-line options, provided by the user. The software modularity of Bambu has mitigated the development effort for the integration of the methodologies and techniques described in this thesis, allowing the exploitation of the architecture independent components of the flow, such as the back-end steps. When targeting the proposed architecture, the tool automatically selects the proper algorithms for its generation. The front end has been enriched, allowing the construction of the Extended Program Dependence Graph, obtained analyzing the PDG, produced in the baseline flow, and applying the rules introduced in Section 5.2.1. A subsequent front-end step processes the EPDG , and computes the activating conditions. The following synthesis algorithms mainly adopts the EPDG as the IR describing the specification. As described before, also the liveness analysis algorithm exploits the EPDG as IR. The module binding technique has been implemented specializing the weight assignment routines; then, once obtained compatibility/conflict graphs, both register and module binding are addressed exploiting the graph covering/coloring algorithms provided by Bambu. As mentioned in Section 5.5, the controller generation routine has required the integration of the controller modules in the resource library: such modules have been described in Verilog. During the synthesis process, the target architecture is described through a hypergraph IR, built incrementally during the synthesis flow. The adoption of a common IR for describing both the FSM and the Parallel Controller architectures has allowed to adopt the same netlist generation algorithms for both the approaches. The developed flow preserves the wide language coverage provided by the release version of Bambu, offering complete support for most of the C language features (2.4).

5.8 Experimental Evaluation

In addition to the example proposed in Figure 4.1, the proposed flow has been evaluated on a set of common HLS benchmarks: Test1 from [118], BarcodeReader from [142], Bcnt, Blit and Des from the Powerstone suite [164] and Adpcm-decode/encode and GSM from the CHstone suite [89].

Table 5.4: *Execution latencies (number of clock cycles) targeting adaptive and FSM-based accelerators.*

Benchmark	Proposed, #CC	FSM-based, #CC	Speed-Up
Loops	804	1607	2
Call-loops	2413	9637	3.99
Test1	52	108	2.08
BarcodeReader	1812	2846	1.57
Bcnt	3360	5156	1.53
Blit	56188	72280	1.29
Des	209194	274688	1.32
Adpcm-decode	806	990	1.23
Adpcm-encode	844	1038	1.23
GSM	22073	37213	1.69
Average Speed-Up			***1.79***

Area and latency of the resulting designs have been compared against the release version (0.9.0) of *Bambu*, which is a representative of typical FSM-based flows. In its default settings, it adopts a LIST-based algorithm for the scheduling task and performs register binding by computing liveness analysis through a non iterative SSA data-flow algorithm [29]. Module

Table 5.5: *Synthesis results: number of required Flip Flop(FF), LUT and FF/LUT pairs targeting adaptive and FSM-based accelerators.*

	Proposed Approach			***FSM-based approach***			
Benchmark	**# FF**	**LUTS**	**PAIRS**	**# FF**	**LUTS**	**PAIRS**	**Area Overhead**
Loops	431	458	458	243	268	276	1.66
Call-loops	1378	1447	1515	371	427	514	2.95
Test1	765	628	745	377	384	400	1.86
BarcodeReader	1131	909	1126	436	503	537	2.09
Bcnt	1777	1520	2085	877	918	1211	1.72
Blit	2515	2603	2807	1832	1715	1964	1.43
Des	7777	4920	7950	5441	3919	5509	1.44
Adpcm-decode	8482	7573	9882	5693	6215	7492	1.32
Adpcm-encode	8939	8426	10829	5920	7343	8726	1.24
GSM	13412	14508	17236	9754	12036	13775	1.25
Average Area Overhead (LUT/FF pairs)							***1.69***

binding is addressed through clique covering. Two considerations motivate this choice: first, statically scheduled designs, based on the construction of a centralized FSM, represent the dominant solution in HLS; second, alternative flows featuring concurrent execution of multiple flows, require particular descriptions of the input specifications, such as Petri nets. This is in contrast with the general trend of adopting programming languages (i.e. C, C++) as specification languages. For both the approaches, the target operating frequency has been set to 200 MHz. Execution latency has been evaluated by simulating the designs through Xilinx ISIM. Table 5.4 shows the simulation results. For every benchmark it is reported a significant speed-up, which varies according to the characteristics of each specification. For example for function *loops*, which presents two parallel loops, it is reported a performance speed up close to 2x, since the proposed approach allows to overlap their execution (as shown in Figure 4.11). A similar behavior characterizes benchmark *Test1*, which includes two parallel loops embedding nested conditionals. The proposed technique is able to exploit parallelism across basic block boundaries, providing a speed up greater than 2x. The maximum performance gain has been experienced when synthesizing function *call_loops*: the approach takes advantage of coarse grained parallelism, allowing the multiple function calls to run in parallel. Such characteristic is also highlighted by simulation results for benchmark GSM, which presents parallelism at function call level and shows a speed-up of 1.69x. When limiting coarse grained parallelism, obtained speedups are relatively lower. This is the case, for example, of the ADPCM benchmarks, which mostly present parallelism in form of ILP. Area has been evaluated by synthesizing the designs with Xilinx ISE ver 14.4., targeting a Zynq xc7z020 device (package clg484). All the synthesis options were left to their default values. Table 5.5 summarizes the synthesis results, reporting the number of allocated Flip Flop (FF) Registers, LUT and FF/LUT pairs for both the parallel controller and the FSM-based flow. The table shows that the area overhead for the proposed flow ranges from 1.24x to 2.95x, when comparing the number of required FF/LUT pairs. Similarly to the latency, the area overhead is directly related to the available parallelism degree in the application. This behavior depends on the binding strategies. The register binding approach produces a conflict free binding, so parallel operations do not share resources. Thus, more registers are allocated as more parallelism is available in the specifications. Similarly, the module allocation heuristic, which focuses on increasing performance, allocates more resources as more operations can run concurrently. In fact the maximum overhead is paid when synthesizing *call_loops*: to run parallel calls concurrently, the flow needs to allocate multiple instances of modules implementing the *loops* function. Such redundancy obviously increases the overall accelerator area, but allows increasing the performance of almost 4 times. A statically scheduled approach cannot provide the same speedup within the same resource budget, while meeting timing constraints. Once again, managing multiple flows with a statical schedule requires the construction of complex product FSMs, which results in unfeasible complexity for highly parallel applications. On average, the area overheads result reasonable considering the speed ups obtained. For example, for the most area demanding application, the GSM benchmark, the dynamically scheduled accelerator design, with an area overhead of only 1.25x , provides a speed up of 1.69x. Furthermore, the accelerator design still requires only a small fraction (27%) of the resources available on the target device.

5.9 Conclusions

This chapter presented an automated flow for the generation of adaptive accelerators. The proposed flow is able to exploit parallelism at different granularities, including control constructs and function calls. The flow also supports variable latency operations. In contrast with typical synthesis flow, the generated architectures feature a distributed controller, supporting dynamic scheduling. When compared to conventional designs, the dynamically scheduled accelerators achieve significant speedups, causing only limited area overheads with respect to the performance improvements obtained. However, the reported overheads identify opportunities for further improvements. Currently, the designed binding strategies focus mainly on performance: future works will investigate and refine such binding techniques, exploring different performance/area trade-offs. Among the different offered features, the proposed approach appears very promising for coarse grained parallelism exploitation, especially in the form of TLP. Next chapters will propose hardware components and techniques to fully take advantage of this characteristic.

CHAPTER 6

The Memory Interface Controller

Previous chapters have described an adaptive accelerator design, based on the definition of a parallel controller, together with a HLS flow for its automated synthesis. The parallel controller architecture improves coarse grained parallelism exploitation, allowing multiple flows to run concurrently and managing each of them independently. This is a desirable feature, since several applications expose parallelism at the task level (TLP), where each thread may be characterized by different runtime latencies. In this domain, exploiting TLP is the key component for improving performance. Hardware synthesis for TLP usually is based on the replication of computing resources. Tasks/threads are designed as custom hardware components, and then multiple instances of such modules are allocated in the final design. The binding strategy adopted in the described HLS flow for concurrent function calls, is an example of this common practice. This approach is supported by the constant improvements in silicon technology, which progressively mitigate the pressure on the design flows of area constraints. However, not all the resources can be straightforwardly replicated: this is the case of memory resources. Memory in fact, especially for parallel applications, is usually shared among the multiple tasks, thus allowing their parallel execution requires to manage concurrency on the shared resources.

The memory bottleneck can considerably degrade performance, especially for memory bound applications where the computation component is not enough to hide the memory latency. Among these, *irregular* applications represent the most affected class: they are memory bound, have poor locality, and usually perform several fine-grained memory accesses with unpredictable access patterns. Their acceleration through custom hardware components seems very promising, since general purpose systems fail in overcoming the memory bottleneck. Solution based on caching require the adoption of coherency protocols, and provide limited benefits, if any, in the absence of locality. More suitable architectural approaches are mostly based on memory distribution and or partitioning. These techniques allow multiple operations to access the memory at the same time, but introduce additional challenges:

1. memory addresses usually are not known statically, thus it is required to identify the targeted memory locations at runtime;
2. tasks may access the memory in parallel, thus it is needed to manage synchronization between them;
3. structural conflicts on shared memory resources have to be avoided.

Partial solutions addressing these issues have mainly focused on efficient implementations of specific applications, such as graph exploration algorithms, with strict assumptions on the target architectural model, including the memory system. This lack of generality makes such techniques difficult to apply on both RTL and HLS synthesis flows. Focusing on the memory component, current design methodologies lack in efficient support of memory hierarchies and sufficient abstraction of external memories, constraining the design process in several ways. This chapter describes a design methodology which aims at alleviating this gap, while addressing the above mentioned challenges, through the definition of an adaptive Memory Interface Controller (MIC) [45] [44]. The MIC introduces an abstraction layer between hardware accelerators design and external memory structure; it allows fine grained parallelism exploitation on memory accesses, automatically manages concurrency on shared resources, and supports atomic memory operations for synchronization. The MIC has been designed as a custom, parameterizable IP, thus suitable for easy adoption in both RTL and HLS flows. Since, as mentioned before, irregular applications represent the class of algorithms most affected by the memory bottleneck, this work mainly focus on them. However, the proposed design techniques can be profitably exploited for parallel applications in general, and are suitable for adoption in HLS, as demonstrated in the next Chapter. The remainder of this Chapter is organized as follows. Section 6.1 introduces the proposed methodology, highlighting some characteristics of irregular applications and providing an overview on existing approaches for their acceleration. Such techniques are detailed in Section 6.2. Section 6.3 remarks some challenges introduced by irregular applications, through the Breadth First Search (BFS) case study. Section 6.4 describes an accelerator design template, adopted as target architectural model for the synthesis process, and highlights the advantages of adopting the parallel controller architecture for its implementation. Section 6.5 details the MIC design, which allows maximizing bandwidth utilization and supporting atomic memory operations. Section 6.6 presents the experimental evaluation, providing an exploration of the design space in terms of spatial parallelism (number of concurrent kernels) and number of memories for the BFS case study. Finally, Section 6.7 concludes the Chapter.

6.1 Motivation

Semantic databases, social network analysis, data mining, bioinformatics, language understanding, pattern recognition and, in general, knowledge discovery are new, emerging irregular applications. They feature irregular data structures such as graph, unbalanced trees or unstructured grids, which employ pointers or linked lists [191]. These data structures provide a large amount of inherent dynamic parallelism, because the application can potentially spawn new tasks for each explored element. However, they also present very poor spatial and temporal locality, because any element can point to any other element, leading to substantially unpredictable, fine-grained, memory accesses. In addition, they usually are large, but difficult to partition without generating load unbalance, and often require synchronization at the level of the single element to coordinate accesses among a multitude of tasks. Irregular applications are mostly memory bandwidth bound, and the key in maximizing their performance is maximizing bandwidth utilization in presence of fine-grained memory accesses. Modern multicore processors, which exploit complex and large caches, mainly rely on data locality and regular computations to achieve high performance. Therefore, they usually execute irregular applications poorly. Multithreaded architectures, which focus on tolerating latencies by switching to other threads while performing memory accesses rather than reducing latencies through caches, usually provide higher performance with this class of applications. Lately, several systems targeting irregular applica-

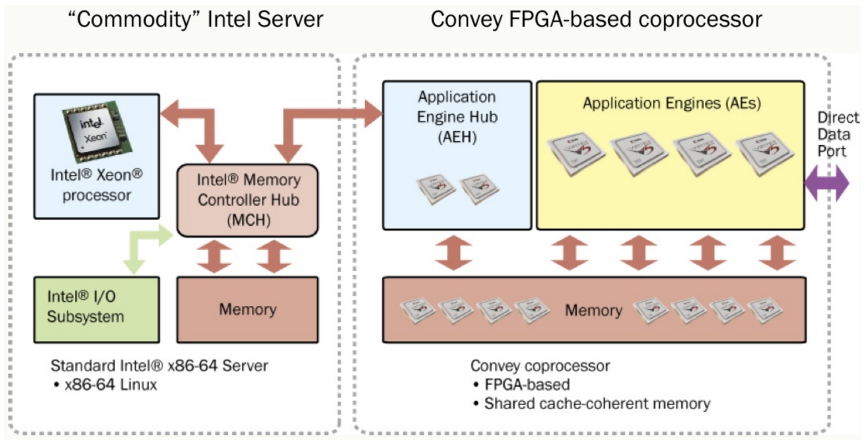

"Commodity" Intel Server
Convey FPGA-based coprocessor
Intel® Xeon® processor
Intel® Memory Controller Hub (MCH)
Intel® I/O Subsystem
Memory
Standard Intel® x86-64 Server
• x86-64 Linux
Application Engine Hub (AEH)
Application Engines (AEs)
Direct Data Port
Memory
Convey coprocessor
• FPGA-based
• Shared cache-coherent memory

Figure 6.1: *Schematic representation of Convey HC-1 platform.*

tions that employ hybrid architectures have appeared. Hybrid architectures integrate both general purpose processors and reconfigurable logic, such as Field Programmable Gate Arrays (FPGAs), to accelerate some specific workloads. Solutions like the Convey HC-1 or the HC-2 include custom personalities (hand-designed accelerators) for some irregular algorithms, such as Breadth First Search (BFS) [52]. The Convey MX-100 [53] implements a full custom manycore, multithreaded processor architecture on the FPGA, together with an OpenMP programming environment (CHOMP), to speed up irregular applications. These platforms integrate complex, custom memory controllers with multiple distributed or banked memories, which support many concurrent fine-grained parallel memory requests, high bandwidths and large memory sizes. Figure 6.1 shows the schematic representation of Convey HC-1 platform, which is an example of such platforms. Its co-processor board includes four user programmable Virtex5 LX 330s FPGAs, called Application Engines (AEs), connected to eight Memory Controllers through a 2,5 Gbyte/s link. Thus each AE can reach the peak bandwith of 20 Gbyte/s when accessing the eight MCs concurrently. These approaches demonstrated promising speed ups with respect to commodity systems, providing alternative, smaller scale, solutions than fully custom systems for irregular applications such as the Cray XMT multithreaded supercomputer [72]. However, they still are custom designs, with hand-developed accelerators or even processors, loaded on the reconfigurable logic. Modifying them means rewriting the Register Transfer Level (RTL) code, the software runtimes and the related interfaces towards the general purpose processors. It is difficult to adapt them to specific or new applications, providing a better utilization of the reconfigurable logic depending on the requirements, or to modify them so to support completely new requirements. On the other end of the spectrum, High Level Synthesis tools appear very promising for hybrid architectures [129]: the application developers can decide to offload some of the kernels (usually, the ones that mostly constrain the application performance) to the reconfigurable logic, and let the tool generate all the RTL code to synthesize. However current HLS paradigms do not consider many of the issues in irregular applications [64,65,88]. They adopt very restrictive abstractions for memory, usually considering a single ported memory, sequentializing all the accesses. In addition, they provide poor (if any) support for synchronization directives. In fact, synchronization is provided through the interaction with off-the-shelf soft processors or custom schedulers, which manage the execution on the hardware modules. This interaction may considerably slow-down execution latencies when compared to full custom hardware executions. This chapter introduces an adaptive Memory Interface Controller (MIC) which features complete concurrency and synchronization management on the memory resources. The MIC dynamically maps memory operations across multiple, distributed and/or multi-ported memories, such as those available in hybrid systems. The accesses routing, towards a particular memory port, is performed at runtime in order to efficiently support unpredictable memory access patterns, which are typical in irregular applications. Concurrency is managed through a lightweight arbitration scheme, which avoid any structural conflict on shared resources, and which does not introduce any delay penalty. Since accesses routing and resource availability checks both occur at runtime, the MIC is able to issue several memory operations at the same time, provided that they do not target the same memory locations, thus improving the system memory bandwidth utilization. Synchronization management is provided through embedded implementations of atomic memory operations such as fetch-add and compare-swap, commonly adopted in parallel programming as synchronization directives. The MIC structure is general, facilitating its integration on different target platforms and possibly its customization. Moreover, its actual implementation can be tuned through parameters, varying for example bitwidths, number of memory accesses which can be concurrently managed and the number of targeted memory banks. This characteristic also facilitates design space exploration tasks. The MIC can be easily integrated in custom hand-written designs, but can also be automatically allocated in typical HLS flows.

6.2 Related Work

In the last few years, several approaches to accelerate irregular kernels, such as graph traversal, with hybrid architectures and reconfigurable devices have appeared. They mainly exploit hand-tuned hardware accelerators. The most important examples are the Breadth First Search (BFS) personalities for the Convey HC systems [52], and the new Convey MX system, which couples a multithreaded custom processor on the reconfigurable logic with an OpenMP programming environment (CHOMP - Convey Hybrid OpenMP) [53], providing significant speed ups for graph exploration kernels [54]. Betkaoui *et al.* [27] present a reconfigurable hardware methodology for efficient parallel processing of large-scale graph exploration problems. The authors introduce an architecture for the Convey HC-1. The proposed solution, demonstrated on the BFS algorithm, decouples computation and communication while keeping multiple memory requests in flight at any given time, taking advantage of the hardware capabilities of the FPGAs and of the parallel memory subsystem. The authors design the architecture by hand, and increase parallelism by replicating the basic BFS kernel. They make the consideration that a HLS synthesis flow could only generate the kernel itself. The proposed approach, instead, can be completely integrated in a HLS framework. In [55], Cong *et al* present an implementation of the fluid registration algorithm on a Convey HC-1 multi-FPGA platform. The authors implement the algorithm through a HLS tool, with additional source-code level optimizations including fixed- point conversion, tiling, prefetching, data-reuse, and streaming across modules using a ghost zone (time-tiling) approach. The paper suggests that further steps are required to fully support the features of a hybrid architecture in an automatic tool. In [182] the authors show how ROCCC 2.0, a HLS tool, can support the Convey HC-1 platform. The paper shows how Dynamic Time Warping and Viola-Jones algorithms are converted from C specification to a Hardware Description Language (HDL) specification, targeting the Convey system. However, the approach still requires to perform optimizations on the code to fully support the platform, and does not introduce new solutions to support irregular applications. To the best of our knowledge, current hardware synthesis methodologies do not address the issues of irregular, and more in general, memory bound applications, in their entirety. These include: abundant task level parallelism, unpredictable and parallel memory accesses, fine grain synchronization through atomic memory operations. There are, however, some approaches that look at supporting some of these features. [88] discusses how to extend ROCCC to support irregular

```
void application_template(){
    //code block

    for(id=init; id<NUM_it; id=id+1 ){
       kernel(id, data);
    }
    //code block
}
```

Figure 6.2: *Application Template*

applications. The authors introduce multithreading to tolerate long memory access latencies, and describe how they customized the ROCC compiler to generate concurrent hardware threads and to support customized state information for each dynamically generated thread. However, they do not address atomic memory operations, and they test the toolchain only on a very simple pointer chasing example. In [18], the authors propose an approach to support multimodal functional units, while minimizing the size of the resulting circuit. Multimodal FUs implement multiple types of operations, usually with different execution latencies. The MIC, with support for atomic operations, can be considered as a special multimodal FU. However, the approach proposed in the paper does not look at increasing parallelism exploitation. It uses a static scheduling technique, which jointly schedules on the same multimodal FUs mutually exclusive tasks.

6.3 Accelerating Memory Intensive and Irregular Applications

Irregular applications typically expose coarse grain parallelism, usually located in loops. The general template provided in Figure 6.2 can map most irregular applications, such as graph problems [27]. There are several challenges when mapping these applications to hardware with automated synthesis flows. First, hardware acceleration generally relies on fine grained parallelism exploitation: ILP inside each kernel is usually limited. Furthermore, the kernels are mostly memory bound, because the largest part of parallel operations are memory accesses. As a result, hardware design methodologies focused on ILP extraction provide limited speed ups. Multi-ported, multi-banked or distributed memories can mitigate this issue by allowing multiple concurrent memory accesses. However, the memory access patterns are irregular. Thus, statically binding a memory operation to a hardware resource is not possible. In turn, this makes concurrent execution of memory operations non trivial. Synchronization issues among different kernels are additional sources of complexity. In fact, different concurrent

```
int bfs(unisgned *offsets, unisgned *edges){
    // n = number of total vertices
    // e = number of total edges
    unisgned *q       = malloc((1+n)*sizeof(unisgned));
    unisgned *qnext   = malloc((1+n)*sizeof(unisgned));
    unisgned  *map     = malloc(n*sizeof(unisgned ));

    init(offset, edges, q, qnext, map);
    unisgned q_size=q[0];
    while (q_size!=0) {
        int vid;
        for (iterId = 1; iterId<q_size; iterId++) {
            unisgned v=q[iterId];
            int i;
    // bfs kernel
            for (i=offsets[v]; i<offset[v+1]; i++) {
                unisgned neighbor=edges[i];
                if (atomic_CAS(map[neighbor], 0, 1) {
                    qnext[atomic_FA(qnext[0])]=neighbor;
                }
            }
    // end bfs kernel
        }
        unisgned *qtmp = q;
        q=qnext;
        qnext=qtmp;
        q_size=q[0];
        qnext[0]=0;
    }
    free(offsets); free(edges); free(q); free(q_next); free(map);
}
```

Figure 6.3: *Queue-based BFS implementation*

kernels share the same memory resources, so a way to preserve consistency is required.

The graph Breadth First Search (BFS) algorithm, a typical irregular application kernel, presents all the previously mentioned aspects. Figure 6.3 shows a C implementation, with atomic constructs added where necessary, of a queue-based BFS. The algorithm works on a graph represented in the Compressed Sparse Row (CSR) format. In the implementation, q and $qnext$ respectively are the queues of vertices to explore in the current and in the next iteration. The first location of each queue stores the number of elements in the queue itself. *Map* is the array that the algorithm employs, at each iteration, to mark the vertices that it has already explored. This implementation can be easily mapped on the general template of Figure 6.2, identifying kernels which may execute concurrently (lines 17-24) . Each one of the kernels iterates over the out edges of a given vertex: when it traverses a new neighbor, the neighbor is marked in the map as visited and

```
void parallel_application_template(){
    //code block

    for(id=init; id<NUM_it; id=id+N){
        kernel(id, data);
        if(id+1<NUM_it)
            kernel(id+1, data);
        if(id+2<NUM_it)
            kernel(id1, data);
        ...
        if(id+N-1<NUM_it)
            kernel(id+N-1, data);

    }
    //code block
}
```

Figure 6.4: *Parallelized Application Template*

added to the next iteration queue. If multiple kernels run concurrently, they perform write accesses to a shared data structure. Consequently, there must be a way to to synchronize the accesses. In software parallel programming, atomic operations, such as compare-and-swap and fetch-and-add (as shown in Figure 6.3), provides synchronization. To overcome the highlighted issues and to achieve significant speed ups, this work proposes a hardware accelerator design, suitable for the synthesis of irregular applications, based on the definition of an adaptive Memory Interface Controller. The MIC:

- allows concurrent execution of memory operations on multiple memory banks, also with an irregular access pattern;
- guarantees memory consistency when multiple kernels concurrently access shared data structures; this is achieved by implementing atomic operations through dedicated hardware modules;
- improves coarse grained parallelism exploitation, allowing multiple kernels to run concurrently without the need of inter-processes communication.

The methodology exploits coarse grained parallelism through "spatial' multithreading (see Section 6.4), i.e. by replicating multiple times the same kernel. The introduction of the MIC enables support of concurrent memory accesses to different memory banks. The MIC dynamically steers memory access requests to the proper memory ports, while managing concurrency among them. The MIC also provides atomic operations, which are considered as a special type of memory operations.

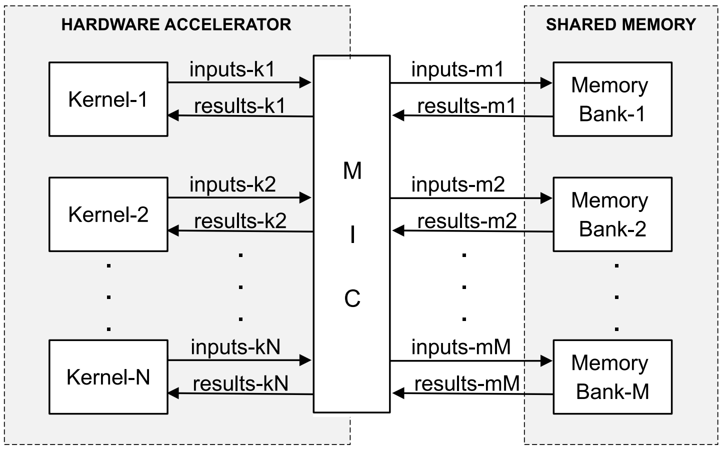


Figure 6.5: *Accelerator design template schematic representation.*

6.4 Accelerator Design Template

In this work we consider an accelerator structure, shown in Figure 6.5, which maps on hardware the application template proposed in Figure 6.4. Such specification template is obtained modifying the general one (Figure 6.2) through partial unrolling, with an unrolling factor of N. This better exposes task level parallelism, and facilitates the implementation process. Replicated kernel function calls are executed only within the bounds of the outer loop. In the general case, checks for the bounds are explicit because the number of loop iterations may not be statically known. For example, in the BFS, it varies dynamically as new vertices are explored. If the upper bound is static, then it is not necessary to insert checks. In the proposed design each kernel call is modeled, from the caller module perspective, as a common variable latency operation, and it is implemented as a stand-alone processing unit. If a kernel function performs memory operations on shared memories, it forwards execution requests to the caller. In case of nested calls, forwarding is recursive, until the top level is reached. At the top level, the MIC manages the memory accesses. Since multiple memory operations may target the same memory locations at the same time, resulting in structural conflicts, memory accesses are also modeled as variable latency operations. In fact, it is impossible to establish at design time if and when a collision will occur. The introduction of the MIC adds an abstraction layer between the accelerator and the memory structure, which decouples the problems of designing the two components. Varying for example the number of available memory banks, or the scrambling function used to distribute data, have no impact on the accelerator implementation. In fact the accelerator can be implemented as an independent module: the designer or the synthesis tool may ignore mutual interferences between different kernels, or in general, memory accesses, since synchronization and concurrency are managed by the MIC. This abstraction also facilitates the design process: for example operation scheduling may be addressed independently for each kernel, without performing difficult inter-functional analysis. In addition it allows trivial kernel replication. All these aspects improve modules reusability and the efficiency of Design Space Exploration tasks, while reducing their complexity. This is an important characteristic, since several design choices affect the system performance: in addition to the memory structure, other design choices which may alter the quality of results are the number of allocated kernel instances, and the number of concurrent memory operations which the MIC should manage. HLS has a significant role in DSE, since it usually allows the generation of several alternative design solutions, with different performance trade/offs. As demonstrated in Section 6.6, and detailed in Chapter 7, accelerators modeled through the proposed design template can be successfully generated through HLS frameworks.

6.4.1 Exploitation of the Parallel Controller Architecture

The parallel controller architecture represent a good candidate for implementing the high level accelerator design template shown in Figure 6.5, since it efficiently manages multiple concurrent execution flows. In addition, it also allows parallelism exploitation of variable latency operations: regardless to the memory structure, concurrency makes memory operations VLOs, even if an isolated memory access has a fixed latency. Variability may grow further for complex memory structures, for example due to network communications latencies. Allowing kernels to issue multiple memory operations is a desired feature for exploiting the available bandwidth. Such parallelism is completely exploited by the parallel controller architecture. Centralized approaches instead can cope with very limited degrees of parallelism. For example, if such degree is 8, a FSM implementation requires hundreds of states and thousands of transitions. These considerations apply also at the task granularity level, where the parallelism is associated to the number of kernels. However, the design template do not impose a particular architectural model for the kernels. If they expose limited parallelism at the level of memory operations, they can be designed as FSM-based accelerators. In this case, memory bandwidth utilization may

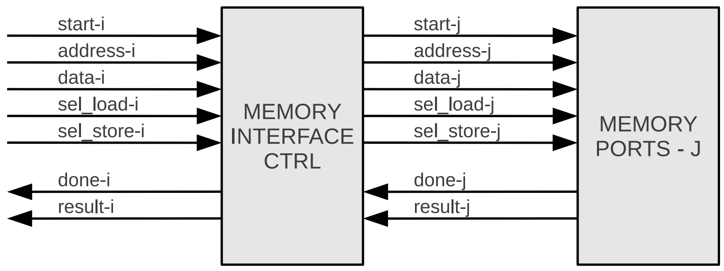


Figure 6.6: *Top Level Memory Interface Controller Structure.*

be improved increasing the number of allocated kernels. Chapter 7 will analyze in depth these aspects.

6.5 Memory Interface Controller Design

The MIC has been designed with the objectives of dynamically addressing irregular memory accesses to the corresponding memory port, while managing their concurrency. Figure 6.6 shows the basic structure of the MIC. Basically, the MIC takes in input memory access requests from N ports, which have an address, a data and an operation type (load/store) line. It routes requests towards one of the M output ports by evaluating their addresses. It serves a request as soon as the corresponding port is available. In a similar way, it routes back M done signals (which notify termination of an operation) and the results (in case of loads) to the requesting operation. The memory is composed of M different and independent banks, and each output port accesses one bank (Figure 6.5). Each memory bank has non-overlapping addresses. This is equivalent to having M different distributed memories. Figure 6.7 provides a schematic view of the controller structure. The MIC associates each input operation i to a Control Element (CE) and a module (PI) that analyzes the input address to establish the destination port. This is obtained embedding in the PI design, the hardware implementation of the scrambling function used to distribute data on the multiple memory partitions. For each output j, it is allocated a Resource Manager (RM), which has the role of managing concurrency. Each CE intercepts execution requests and forwards them to the RMs, until they are accepted (Section 4.3. A Port Index signal produced by the PI, working as selector of the steering logic (connection 1), allows performing the routing of the requests. Once a RM accepts a request, it sends back an ack signal to the corresponding CE, disabling it, and to the UNBD module, which is responsible of setting the selectors while the operation is running. These

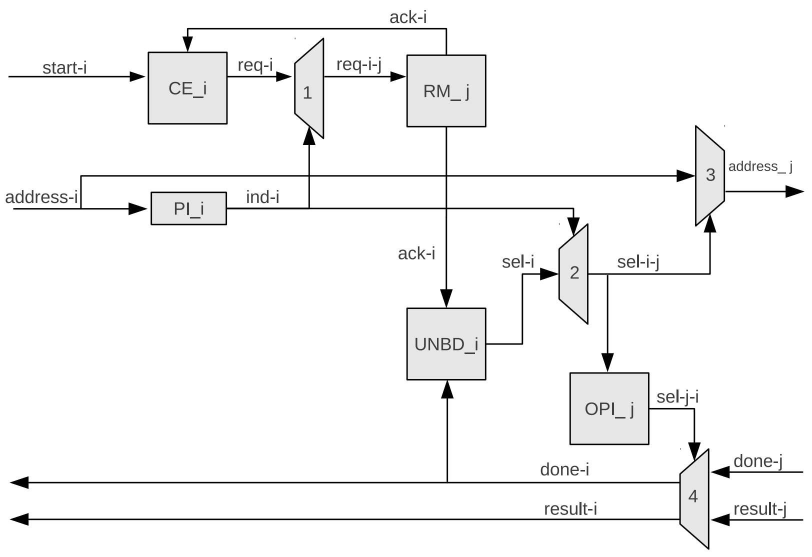


Figure 6.7: *Memory Interface Controller schematic representation.*

signals, according to the output of PIs (connection 2), drive the steering logic that feeds the memory ports (connection 3). Figure 6.7 only shows the connections and logic for the address line of an input port. All the other input lines follow a similar approach, duplicating the steering logic and interconnections. Similarly, the MIC routes done signals and results coming from memory (read accesses), according to the requesting input port. In this case, OPeration Index (OPI) modules provide the selectors for the interconnections. OPIs identify the input port requesting the memory access. The design of the MIC, thanks to its modularity and regularity, is not constrained by any particular characteristic of the ports (number or bitsize of inputs/outputs).

Table 6.1 shows a high-level characterization of resources requirements and complexity of the MIC, when varying the number of concurrent memory operations (N), the number of memory banks(M) and the number/size of memory ports(n). Number and size of memory ports is not considered as a parameter, since it is imposed by the targeted memory structure. We can classify the allocated components as elements which generate routing signals, and steering logic. The first have limited complexity: only the Resource Manager complexity increases with the number of input operations. However, it is a lightweight module, which requires only N-1 two bit or

Table 6.1: *Resource requirements and complexity for a MIC managing N input operations towards M memory banks.*

Resource	Number	Complexity
CE	N	constant
RM	M	f(N)
UNBD	N	constant
PI	N	constant
OPI	M	f(N)
conn1,3,4	M	f(N,M)
conn2	N	f(N,M)

gates to be implemented. The main impact on area requirements of the MIC, is due to the steering logic, in particular connection objects 3 and 4, since they route signals of higher size (e.g. address, data, results), and both their number and complexity grows with N and M.

Atomic operations Atomic operations indivisibly perform a sequence of operations (read and write) on a given memory location, ensuring that its content is not modified by other operations during their execution. In the considered design, we obtain an atomic behavior by delegating management of atomic operations to the MIC. When the MIC accepts the execution request of an atomic operation, it exclusively binds the associated memory port to the operation, until its completion. The MIC manages atomic execution through dedicated hardware. For example, fetch-and-add operations read the value at the specified address, add the provided operand to the previously read value, and then store the result in the same memory location. They return the old value they read. The MIC implements this operation as follows. First, it performs the load operation. When the MIC receives the done signal coming from the memory, it intercepts the signal, buffers the loaded value, and calculates the sum. Then, it stores the result of the sum into the memory. The subsequent done signal corresponds to the completion of the whole atomic operation, thus it returns the buffered value. The MIC implements other atomic operations following the same approach. The MIC includes dedicated hardware to manage atomic operations for each memory port, thus allowing concurrent execution of an atomic operation per memory bank.

Table 6.2: *Area evaluation of Memory Interface Controllers.*

	M=2		M=4		M=8		M=16	
N	**FF**	**LUT**	**FF**	**LUT**	**FF**	**LUT**	**FF**	**LUT**
2	4	216	4	344	4	721	4	1434
4	8	308	8	496	8	975	8	1894
8	16	623	16	997	16	1958	16	3905
16	32	1257	32	2061	32	4168	32	8030
32	64	2891	64	4875	64	10005	64	19147
2	76	384	148	609	292	1249	581	2434
4	80	570	152	838	296	1668	584	3278
8	88	1012	160	1482	304	2939	592	5771
16	104	1908	176	2857	320	5598	609	11194
32	136	4114	208	6348	352	12474	643	24726

6.6 Experimental Evaluation

The MIC has been designed, and implemented in Verilog, in two different versions. The first only considers load and store operations, without support for atomic operations. It takes in input memory addresses, input data for store operations, selectors for identifying the operation type, and a start signal. It produces in output done signals and, in case of load operations, a result. The second version of the module extends the previous one by providing support to fetch-and-add and compare-and-swap atomic operations. There are additional selectors in input, and dedicated output lines for providing the results. The same lines used for input data of store operations provide the operands. Both the MIC implementations are customizable, indicating number of input operations and number of memory banks as parameters.

As a preliminary step, we evaluated the area of the designed modules, varying both the number of inputs N (associated with the maximum number of parallel memory operations) and the number of outputs M (associated with the number of memory banks). The designs have been synthesized with Xilinx ISE ver 14.4, targeting a Virtex 6 xc6vlx75t-3ff784 device. Table 6.2 summarizes the synthesis results for both versions of the controller, in terms of allocated Flip Flop (FF) registers, and LUT. The first 5 rows refer to the interface controller supporting only loads and stores, the last five rows, instead, to the design that also supports fetch-and-add and compare-and-swap. The size of the address, data and result lines, have been set to 32 bits. The number of required resources increases almost linearly with both the number of inputs and number of outputs. For the first design, the

Table 6.3: *Simulation results; input graph: 5000 nodes, average degree: 10.*

	M = 4			M = 8		
kernels	**2cc**	**5cc**	**10cc**	**2cc**	**5cc**	**10cc**
1	306156	524376	888076	306156	524376	888076
4	113210	221920	403351	107696	201030	357887
5	98963	200221	370933	91536	176243	318709
6	88638	184015	346154	80379	156811	287159
7	81891	174452	330933	72959	143184	266953
8	77218	165959	317174	66055	132809	246044

number of FFs is negligible, because the modules allocated for managing operations issue and temporization only require 2 FFs. The second design also needs FFs (32 for each line) for buffering the fetch-and-add results. For the second design it is also reported an increase in the number of LUTs, due to the additional logic required to implement the atomic operations.

The proposed approach has been evaluated by exploring area and performance when synthesizing the BFS algorithms with different parameters. The choice is motivated by the fact that the BFS is considered one of the most typical irregular application kernel. Different implementations, varying the number of allocated kernels and the number of available memory ports, have been compared. Such designs have been automatically generated with the HLS flow described in Chapter 4, and manually modified introducing the MIC in the obtained implementations. Each kernel performs six memory accesses, and two atomic operations, i.e., one fetch-and-add and one compare-and-swap. Since the kernels require access to the whole memory, the MIC manages the memory accesses at the top level. According to the program dependences, each kernel can issue up to two concurrent memory operations. For this reason, two input ports of the MIC are reserved to each kernel. All the accelerators have been synthesized targeting an operating frequency of 100 MHz, and all were able to meet the timing constraint. Performances have been evaluated, in terms of execution latency, while also varying the size of the input graph and the latency model of the memory operations (2, 5 and 10 clock cycles per operation).

Tables 6.3,6.4,6.5 report the simulation results (number of clock cyles) for the different data sets, characterized as follows:

- Table 6.3: 5000 nodes, average out degree for node: 10, overall number of edges: 22767;
- Table 6.4: 5000 nodes, average out degree for node: 20, overall number of edges: 47597;

Table 6.4: *Simulation results; input graph: 5000 nodes, average degree: 20.*

	M = 4			**M = 8**		
kernels	**2cc**	**5cc**	**10cc**	**2cc**	**5cc**	**10cc**
1	754146	1253058	2084578	754146	1253058	2084578
4	264826	505388	919541	253693	463188	823023
5	226582	454496	837274	213431	398821	717946
6	203785	417353	785667	187255	355475	646308
7	187423	397995	748200	167535	322705	590834
8	174418	380939	720322	151475	296591	545312

Table 6.5: *Simulation results; input graph: 5000 nodes, average degree: 30.*

	M = 4			**M = 8**		
kernels	**2cc**	**5cc**	**10cc**	**2cc**	**5cc**	**10cc**
1	1176994	1932928	3192818	1176994	1932928	3192818
4	405913	766858	1393948	391490	706281	1244692
5	350162	691134	1271945	329344	616505	1096587
6	311400	636499	1192199	286875	540074	977461
7	284709	600454	1132529	256882	491327	891643
8	265618	576155	1091994	231492	449192	821210

- Table 6.5: 5000 nodes, average out degree for node: 30, overall number of edges: 72887.

To preserve the irregularity, graphs have been randomly generated. All the results refer to the execution of the complete BFS algorithm. First, the specification has been synthesized allocating only one kernel function, and interfacing it to a single bank memory. In these settings, parallelism exploitation is strictly bound to ILP. We compared the obtained latencies for serial execution (1 kernel), and then progressively increasing the number of allocated kernels (from 4 to 8), targeting a 4-bank and a 8-bank memory architecture. For all the experiments, we verified speed-ups when increasing the number of kernels. To demonstrate the effectiveness of the proposed approach in parallelizing irregular memory accesses, it must also been considered the relative speed-ups when varying the memory latency. Regardless of the number of kernels, it is reported a significant speedup when increasing the number of memory banks. The gains are higher with high latency memories, because in such a case the execution latency is dominated by the memory accesses. This is a valuable result, especially when considering that the speed up increases when increasing the number of kernels, thus providing higher concurrency on the memory resources.

Table 6.6 summarizes the area requirements (number of FF and LUT

Table 6.6: *Area evaluation of the generated designs.*

	1ker/M=1	**4ker**	**5ker**	**6ker**	**7ker**	**8ker**
FF,M=4	942	2466	3066	3415	4124	4563
LUT,M=4	1141	4981	6200	7218	8378	8911
FF,M=8	942	2611	3210	3559	4268	4706
LUT,M=8	1141	6367	7912	9305	10935	11586

slices) of the generated accelerators. It is remarked that the single kernel implementation interfaces with a single banked memory. Thus, it is not affected by the area overhead of the MIC. Two aspects mainly determine the area utilization. The first one is associated with the number of allocated kernels: each kernel requires 433 FF and 488 LUT slices. The second component is the cost of the MIC: increasing the number of kernels slightly increases the cost of the controller, because two 2 additional ports per kernel are added to the MIC.

6.7 Conclusions

This chapter presented a synthesis approach targeting memory intensive and irregular applications, based on the implementation of an adaptive Memory Interface Controller. Irregular applications present unpredictable and fine-grained (single word) memory accesses, exploit large data sets, are mostly memory bandwidth limited and have high synchronization intensity. The MIC supports distributed and multi-ported memories, which provide very high bandwidths for fine-grained memory operations; since it manages both synchronization and concurrency among the memory resources, it allows multiple accesses to execute at the same time, and introduces an abstraction layer which facilitates the design of custom accelerators. It also provides support for atomic memory operations. The approach has been evaluated on the Breadth First Search (BFS) algorithm, exploring several trade-offs in terms of number of kernels and number of memories, showing how the MIC allows exploiting the parallelism available in the algorithm, maximizing concurrency of memory operations, and thus improving the system bandwidth utilization. The methodology has been designed in order to be compatible with different specific target architectures or synthesis flows (e.g. RTL vs HLS). Next Chapter will show how the HLS flow proposed in Chapter 4 has been extended, taking advantage of the MIC, for supporting and improving the synthesis of generic annotated parallel applications.

CHAPTER 7

Support for Annotated Parallel Specifications

While most of the techniques proposed in literature focused on ILP to improve performance, recent efforts in hardware design methodologies are aiming at exploit parallelism at coarser granularaties. However, the synthesis of Task Level Parallel applications introduces new challenges:

1. C-based specification are inherently serial, making it difficult to identify TLP;
2. the generated hardware should be able to manage multiple execution flows;
3. tasks may share memories, thus requiring ways to manage concurrency;
4. parallel execution may involve synchronization among tasks (e.g., when they share program variables).

The techniques described in Chapter 5 and 6 partially solve theses issues. In the proposed HLS flow parallelism identification occurs through front-end analysis: dependencies between operations are examined, allowing the identification of concurrent portions of code. However, most static analysis techniques are still not able to provide an accurate characterization of the program structure. The main limitation when processing C-like code arise from the analysis of memory accesses. Most compilers adopt (over)conservative approaches, serializing memory operation whenever there is no evidence that they target different locations. For modular designs, this often leads to serialization of function calls. The second challenge, related to the management of concurrent flows, is completely addressed by the designed parallel controller architecture. The last 2 issues have been considered through the definition of the Memory Controller Interface. However, in the general case, it is required user intervention: for example, there is not any automated technique for the identification of task-synchronization points, since these may depend on the *behavior* of the input program, and in addition, this in general would require, as for parallelism identification, sophisticated alias analysis. A promising approach for mitigating these issues is the adoption of parallel programming paradigms (e.g. OpenMP, CUDA) for specification. Parallel code, usually instrumented through pragmas, directly expose the available TLP, and allows the programmer to explicitly indicate synchronization points. This does not represent a limitation for the users, since parallel programming has massively been adopted in software development. This chapters investigates the opportunities offered by these paradigms in HLS, and proposes a refined HLS flow for their exploitation [41] [47]. The flow takes advantage of different architectural solutions, i.e. both FSM and parallel controllers, and exploits a restructured design of the MIC, tuned to perfectly suit with a modular synthesis process. The Chapter is organized as follows. Section 7.1 highlights the advantages of considering parallel programming paradigms, and motivates the adoption of both static and dynamic controllers in the target architecture design. Section 7.2 describes recent research effort in the definition of synthesis methodologies for TLP. Section 7.3 illustrates the refined HLS flow, highlighting how the synthesis process proposed in Chapter 5 has been extended. Section 7.4 analyzes how the Memory Interface Controller design, presented in previous chapter, has been tuned to be effectively exploited in the proposed flow. Section 7.5 evaluates the methodology on a set of parallel applications, instrumented through OpenMP pragmas. Section 7.6 concludes this chapter, suggesting further research directions.

7.1 Motivation

In the last two decades, researchers spent most of the efforts in exploiting Instruction Level Parallelism (ILP). Hardware implementations that exploit ILP can provide significantly higher performance than their software counterparts. However, ILP is substantially limited in most applications [187] and, as in microprocessor architecture design, further efforts to increase its exploitation bring diminishing returns. For this reason novel HLS approaches are now looking at exploiting coarse grained parallelism, mainly in the form of Task Level Parallelism (TLP). However, extracting TLP from a inherently serial specification is non trivial. The front-end analysis presented in Section 5.2 potentially allows coarse grained parallelism identification, as demonstrated through experimental results. However it is able to detect only a portion of the available parallelism, since it is constrained by the dependences identified during the compilation. More in detail, structural dependencies are often introduced to preserve memory consistency. Alias analysis thus plays a significant role: most software compilers adopt very conservative approaches even for relatively trivial situations. For example, if multiple functions write in the memory on different locations, they are often serialized even if they are completely independent, and if it is possible to determine statically that the addressed locations do not overlap. This aspect represent one of the main factors mitigating parallelism identification. For these reasons, as discussed before, early works investigating TLP adopted particular specification languages such as Petri nets or process networks, but so reducing their applicability and effectiveness in HLS. To overcome this limitation, new generation HLS tools should support common parallel programming paradigms, for several reasons:

- they are widespread in software development;
- they still enable software execution, thus facilitating HW/SW co-design, partitioning, and DSE;
- they directly expose TLP, identifying portions of code which may run in parallel;
- they provide explicit synchronization directives, facilitating the synthesis;
- even if they expose the programmer to lower details associated to synchronization and concurrency, most of them hide architectural details.

Among the multitude of such models currently available, [56] suggests to chose device-neutral APIs, such as OpenMP. For example, several approaches considered CUDA as specification languages; while still able to expose TLP and synchronization, it has been originally designed to model applications mapped onto NVIDIA graphics processing units (GPUs), and thus includes many GPU specific features which are not suitable, for example, for FPGAs. For this reason this work targets OpenMP-like APIs. The techniques presented in the previous chapters have been refined and extended, leading to the definition of an improved and more flexible HLS flow. In addition to the features already described, the proposed flow offers:

- support for annotated parallel specification, with the aim of exploiting both ILP (within a task) and TLP (among multiple tasks);
- synthesis of modular, hierarchical components, with different design approaches depending on the specification characteristics: behaviors characterized by ILP only are synthesized as statically scheduled accelerators, while in presence of TLP the flow generates dynamically scheduled architectures, based on the adaptive parallel controller.
- a Hierarchical Memory Interface Controller (HMIC), based on the MIC design, which better matches with the characteristics of the particular proposed flow, while preserving all the features offered by the centralized implementation of the MIC.

7.2 Related Work

Several solutions for exploiting TLP involve the automated synthesis of kernels on custom hardware, but require the introduction of custom schedulers or processors for coordinating tasks. For example, the approach presented in [93] maps tasks, identified by partitioning the input behavior, onto custom Processing Units (PU) and manages their execution through a top-level controller. [38] similarly manages task execution through a *Control Processor* (CP), which represents the top layer of a Multi-Level Computing Architecture (MLCA) [107]. The lower level of the design is a set of PUs: they may be custom accelerators or soft-cores. The CP has the main role of scheduling and mapping tasks onto PUs, considering a top-level control program that consists of *task instructions*. Compared with these hybrid solutions, the proposed approach generates custom accelerators able to exploit TLP without the intervention of external control units or soft-cores. This avoids (high) latencies due to communication and synchronization. In addition, thanks to modularity, the flow allows TLP exploitation at any level of the call structure of the input program: each task may, in turn, include multiple parallel sub-tasks. Recent work in HLS proposes synthesis flows that consider parallel programming for the input description. Among the most well know parallel APIs or languages we can find: CUDA, OpenMP, OpenCL, pthreads [22] [56]. OpenMP has been among the first APIs identified as suitable for HLS. In [68], the authors draw preliminary guidelines on the synthesis of OpenMP programs, analyzing pragmas' semantics and indicating which of them can be successfully exploited in hardware synthesis. [34] proposes extensions to the set of OpenMP pragmas that could be used for designing hardware accelerators. [121] discusses a HLS flow that translates OpenMP programs in synthesizable Handel-C or VHDL. However, the flow imposes severe restrictions on the input specifications: for example, it does not support global variables. Moreover, it does not consider ILP exploitation: each kernel executes serially, and only performs one operation at a time. The LegUp framework [7] provides both a typical HLS flow, for the synthesis of hardware accelerators, and a MLCA flow. The latter allows the automatic generation of designs that couple a MIPS processor and custom PUs. This approach enables the concurrent execution of parallel kernels, identified from OpenMP and pthreads specifications. The processors is responsible for managing concurrency and synchronization among the tasks. [143] and [144] describe techniques to automatically synthesize CUDA kernels on FPGAs. In [143], the authors propose a framework that translates a CUDA program into a parallel-c specification. The Autopilot HLS tool then performs the actual synthesis of the generated C code. [144] introduces a HLS framework for mapping CUDA kernels onto hardware accelerators while considering different granularities of parallelism. [58] proposes a synthesis approach that targets OpenCL applications. These applications are composed of a host C/C++ program and of several OpenCL kernels. The approach compiles and executes the host program on x86 processors, while it synthesizes and download the OpenCL kernels on a FPGA device. An API binds the host function calls to the hardware kernels. Host and FPGA accelerators communicate through PCI-E.

7.3 Refined High Level Synthesis Flow

The proposed flow allows the generation of hardware accelerators starting from a parallel C specification. It takes advantage of emerging programming models that explicitly express concurrency, for example through pragma annotations, such as OpenMP [59]. One of the main motivation

```
void application_template(){
  //code block A
  #pragma parallel
  for(id = init; id < NUM_it; id = id+1)
     kernel(id, data);
  //code block B
}
```

Figure 7.1: *Application template*

```
void application_template_PU(){
  //code block A
  for(it= init; it < NUM_it % NUM_ACC; it = it+1)
     kernel(it, data);

  for(; it < NUM_it; it= it + NUM_ACC)
     #pragma parallel
     for(id=0; id < NUM_ACC; id = id+1)
        kernel(id, data);
  //code block B
 }
```

Figure 7.2: *Application template for Partial Unrolling*

is that the emergence of massively parallel architectures in the last decade made parallel programming techniques more common. In addition, there now exists several source-to-source compilers (e.g., [60]) that can (partially) identify parallelism in a specification and automatically emit annotated code. However, currently they cannot emit synchronization directives. As discussed before, the framework described in this thesis is already able to automatically extract portions of the available TLP, even in the absence of parallelization directives; as most software compilers, it treats pragmas as *hints* to guide the HLS process, and especially dependency analysis, improving the final QoR.

Figure 7.1 shows an example of parallel code template, characterized by the presence of a parallel loop. Without loss of generality, bodies of parallel loops are modeled as function calls: this is coherent with most software compilation approaches, which perform this code transformation during the compilation process. The approach also allows exploiting loop unrolling, avoiding an excessive growth of the number of operations, since loop bodies are embedded in function calls. This is a crucial aspect, because the tool can exploit loop unrolling to expose parallelism at the function call level. Although complete unrolling is not always applicable nor profitable, like when the number of iterations is not statically computable or when the number of iterations is too high, simple code transformations allow *partial* unrolling. Figure 7.2 shows a possible transformation applied to the application template in Figure 7.1, enabling unrolling of factor NUM_ACC. The transformation makes the bounds of the innermost loop, starting at line 7, constant values, enabling its parallelization also when NUM_it is unknown at compile time.

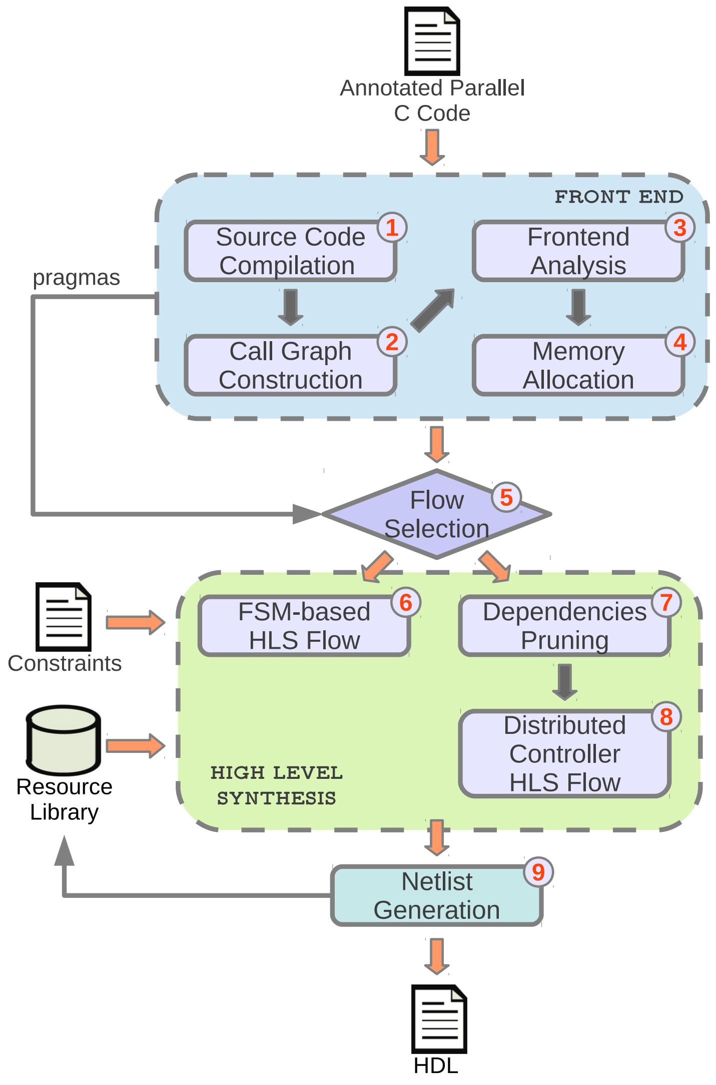


Figure 7.3: *Proposed High Level Synthesis Flow.*

Figure 7.3 summarizes the refined HLS flow. We can coarsely divide it in two phases: the front-end phase and the synthesis phase. The main differences with respect to the baseline approach presented in Chapter 4 refer to the target architecture design. The flow generates hierarchical modules, according to the specification call structure, and implements each module following two approaches: a FSM based approach for components characterized by ILP only, and the parallel controller approach, if multiple flow may run concurrently. For simple *single-flow* behaviors indeed, FSM-based approaches generate efficient implementations, since able to extract ILP while exploiting static analysis to optimize the generated data-path. In these settings the parallel controller design usually cannot provide substantial benefits, and typically is slightly more area demanding. On the contrary, for TLP specifications, the parallel controller design allows multiple flows to execute at the same time, and it is much more efficient. Thus the flow tries to take advantage of both the techniques. A dedicated step in the HLS flow establishes which approach should be followed for the synthesis of a given function (Section 7.3.2). The other differentiating component is the memory interface design. The flow embeds in the final architecture a hierarchical memory interface controller, obtained restructuring the MIC in Chapter 6 to best adapt to the implemented HLS tool.

```
#define NUM_ACCELS 4
#define ARRAY_SIZE 10000
#define OPS_PER_ACCEL ARRAY_SIZE/NUM_ACCELS

int array[ARRAY_SIZE]={...};
int output[NUM_ACCELS];

void add (int id, int startidx, int endidx){
  int sum=0;
  int i;
  for (i = startidx; i < endidx; i++)
    sum += array[i];
  output[id] = sum;
}

void main (){
  int i;
  #pragma omp parallel for num_threads(NUM_ACCELS)
  for (i = 0; i < NUM_ACCELS; i++) {
    add(i, i*OPS_PER_ACCEL, (i+1)*OPS_PER_ACCEL);
}
```

Figure 7.4: *Example annotated parallel specification*

```
int array[10000] = {...};
int output[4];
void add(int accelnum, int startidx, int endidx)
{
  unsigned int internal_11522_13;
  int i_9;
  int i_17;
  int sum_16;
  int sum_8;
  int* temp_addrR_144;
  int sum_15;
  unsigned int TMR_idx_143;
  int internal_11527_7;
  i_17 = startidx;
  if (startidx < endidx)
  {
    BB_LABEL_3:
    sum_15 = gimple_phi(sum_8, 0);
    i_17 = gimple_phi(i_9, startidx);
    internal_11522_13 = (unsigned int) (i_17);
    TMR_idx_143 = internal_11522_13 << (2u);
    temp_addrR_144 = (int*)(array+TMR_idx_143);
    internal_11527_7 = *((int*)(temp_addrR_144));
    sum_8 = (int)(sum_15 + internal_11527_7);
    i_9 = (int)(i_17 + (1));
    if (i_9 != endidx)
      goto BB_LABEL_3;
  }
  sum_16 = gimple_phi(sum_8, 0);
  (output)[accelnum] = sum_16;
  return ;
}
void main()
{
  add(0, 0, 2500);
  add(1, 2500, 5000);
  add(2, 5000, 7500);
  add(3, 7500, 10000);
  return ;
}
```

Figure 7.5: *Intermediate code produced during the compilation process.*

7.3.1 Front-end

The first step of the front-end phase consists in the source code compilation, which involves compiler optimizations and code transformations. Figure 7.4 shows an example of parallel specification. The compilation process produces optimized intermediate code, as shown in Figure 7.5. The most important optimizations applied include unrolling and constant propagation. These further facilitate the identification and subsequent synthesis of the concurrent tasks (see *main* function in Figure 7.5). Step 2 is the construction of a Call Graph (CG). All the subsequent steps of the flow act on a single function at a time. Step 3 acts as in the baseline flow, analyzing control and data dependencies, and then building graph IRs of the program (including the EPDG). Step 3 also includes an analysis of the memory operations, which in turn allows inserting structural dependencies (to guarantee ordering of the memory accesses, if needed), or identifying local data which can be bound on local memories allocated on the target device (Step 4).

7.3.2 Flow Selection

The designed tool is able to generate hardware modules following both FSM and Parallel Controller based approaches. The first is a typical HLS flow, which statically computes an operation schedule, according to resource and timing constraints, as briefly described in Section 7.3.4. The second approach implements the parallel controller architecture, supporting dynamic scheduling and TLP exploitation. For each function, step 5 establishes which one of the two flows will perform the synthesis. The selection is driven by information acquired through the analysis of the CG and of the annotations in the source code. The flow selection process marks as *parallel* the code occurring under a *parallel section*. If the current function calls other tasks identified as parallel in the CG (i.e. exposes TLP), then the flow implements it with the distributed controller approach. A Design Space Exploration (DSE) engine may perform autonomously the flow selection, also for non-annotated specifications. For example, it may detect parallelism at the level of concurrent memory accesses or unbounded operations in loop. These can be modeled as multiple execution flows, making the distributed controller approach more profitable. However, the definition of such a DSE engine is not in the scope of this thesis, and it is postponed to future work.

7.3.3 Synthesis

The synthesis phase produces a hardware implementation of the input specification, which is described through a graph based IR. It takes in input synthesis constraints, which include timing and resource constraints, and a resource library. The resource library contains the description of all the available modules, indicating the provided functionality, the actual HDL description, and performance metrics such as area and combinational delay, which affect the synthesis algorithms. The module description includes additional tuning parameters, such as the number of pipeline stages needed to reach a particular target frequency and to meet timing constraints. When the

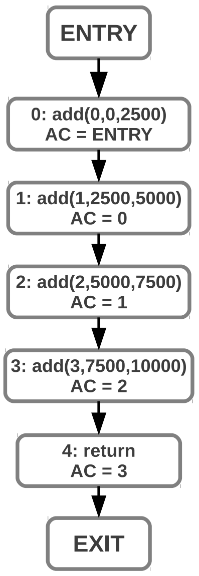


Figure 7.6: *EPDG with structural dependencies.*

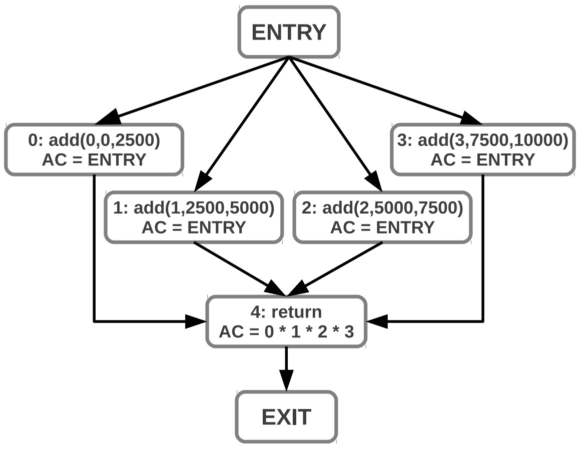


Figure 7.7: *EPDG after dependences pruning.*

flow synthesizes a function, it updates the library with the module description, allowing the execution of a single synthesis process when multiple calls of the same function occur.

7.3.4 FSM-based synthesis flow

The FSM-based flow produces statically scheduled hardware modules, managed through a FSM controller. It exploits the synthesis algorithms offered in the release version of Bambu, briefly described in Section 2.4. Summarizing: the flow performs scheduling through a LIST-based algorithm [148], which is constrained by resource availability; module binding is addressed through a clique covering algorithm [178] on a weighted compatibility graph. It builds the compatibility graph by identifying the operations that can potentially share the same hardware resource and assigns weights taking into account area/delay trade-offs as a result of sharing. Finally, the flow treats register binding as a coloring problem on a conflict graph: two storage values conflict if their life intervals overlap. The flow determines this information through a Non-Iterative SSA liveness analysis algorithms [30].

7.3.5 Parallel Controller synthesis flow

The front end may introduce *structural dependencies* to guarantee memory consistency, especially in presence of accesses to shared memories. This conservative approach may severely limit parallelism exploitation. This issue is mitigated exploiting the information provided by the annotations in the source code, which allows pruning the unnecessary structural dependencies (step 7). Functions marked as parallel are assumed to be safely executed concurrently. If not so, the application developer should insert synchronization directives, as usual in parallel shared memory programming approaches. Figures 7.6 and 7.7 compare initial and pruned EPDGs for the main function in Figure 7.5, annotated with the Activating Conditions (ACs) of the operations. The identification of ACs occurs after the EPDG pruning step. The data-path synthesis algorithms adopts the pruned IR as input, and proceeds as described in Chapter 4: the synthesis algorithms at this point do not longer require to consider the pragmas in the specification, since all the information on parallelism is embedded in the IR.

7.4 The Distributed Memory Interface

One of the main challenges in exploiting TLP resides in managing concurrency when accessing shared memories. This challenge is addressed through the definition of a Hierarchical Memory Interface (HMI), which exploits a restructured design of the MIC presented in previous chapter. The hierarchical implementation of the MIC still avoids any structural conflict on shared resources, while supporting atomic operations to enable synchronized accesses. Hierarchy is exploited to preserve the design modularity.

Figure 7.8 shows the schematic representation of the HMI for the example proposed in 7.4. Each synthesized function has a proper Memory Interface (MI), which provides the following ports:

1. **sel_store**: write access request;
2. **sel_load**: read access request;
3. **addr**: the memory address;
4. **w_data**: data to write;
5. **r_data**: data loaded;
6. **ready**: notifies completion of a memory access.

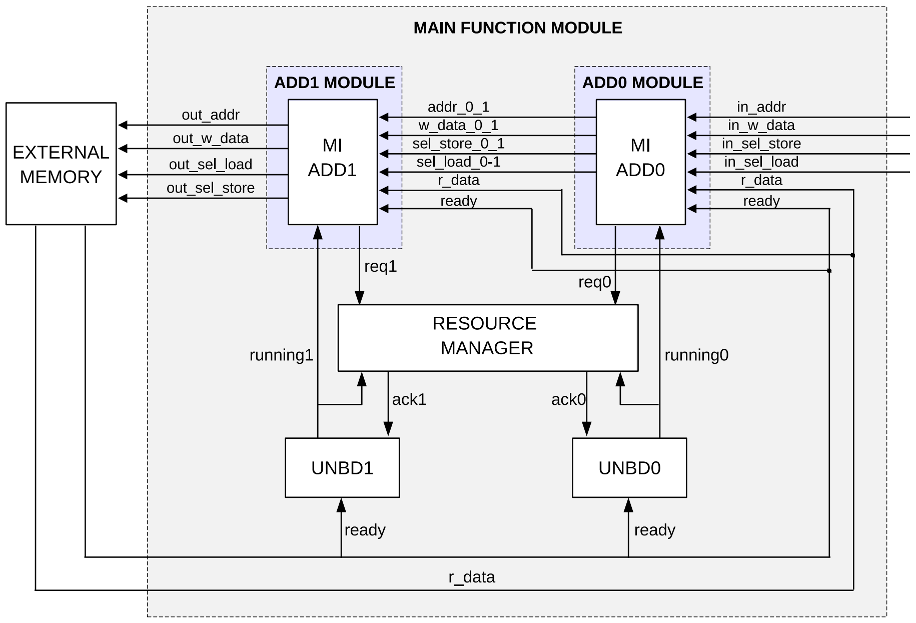
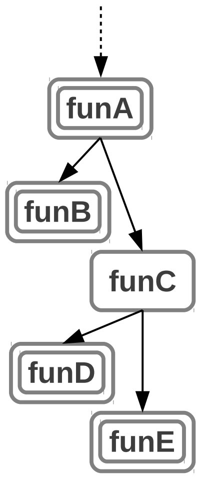


Figure 7.8: *Memory interface structure.*

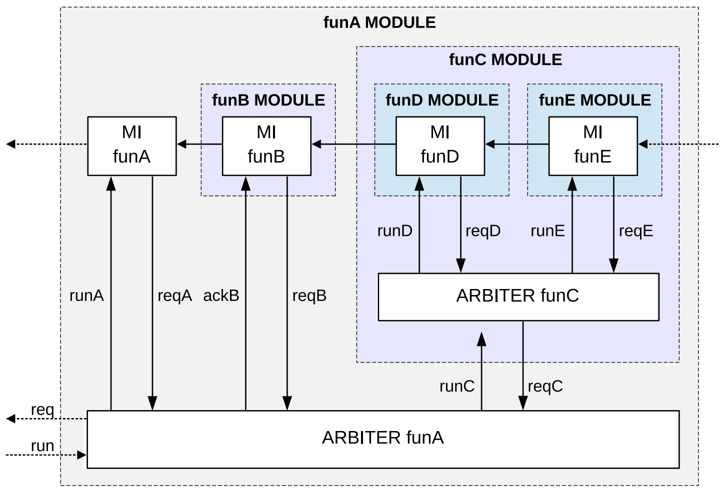
Multiple MIs are chained, and they can perform only one memory operation at a time.

Signal propagation starts from the leaves of the CG. First, it interfaces function modules at the same level in the CG. Then, it crosses the hierarchy, until reaching the top level function module. The top level function module is the only one that directly interfaces with the memory. The propagation scheme requires that only one function module at a time sets *sel_store* and *sel_load* signals, which identify memory access requests. Standard approaches can ensure this behavior through scheduling, which guarantees performing only one operation at a time. However, this often degenerates in the serialization of function calls. This issue is avoided by integrating additional control logic in the HMI that exploits the RM and the UNBD modules, previously presented as building blocks for the parallel controller architecture, and also adopted for the implementation of the centralized MIC.

RMs intercept memory access requests (*req*). If a RM accepts a request, it notifies a dedicated UNBD module, associated with the MI of each function. For example, in Figure 7.8, the module *UNBD1* is associated with the MI of module *ADD1*, while *UNBD2* is associated with the MI of *ADD2*. In this example, the caller (i.e., the main function) does not directly access memory, hence it is not part of the arbitration scheme.



(a)



(b)

Figure 7.9: *Example Call Graph (a) and associated memory interface structure (b). The framed nodes in the CG are associated with functions that directly perform memory accesses;* arbiters include RMs and UNBD modules.

Figure 7.9a shows an example of CG: framed nodes denote functions (*funA*, *funB*, *funD*, *funE*), which directly access the shared memory. Figure 7.9b shows the associated HMIs. Function *funC* does not perform any memory access, but the called functions (*funD*, *funE*) do. For this reason, funC is involved in the management of memory concurrency at the caller level (funA). This greatly enhances the modularity of the generated design.

Atomic operations In parallel programming, thread synchronization is usually achieved through atomic memory operations, which access shared memory locations in mutual exclusion. The HMI design provides natural support for atomic operations, without additional overheads. Atomic memory operations may include a set of operations. The flow synthesizes atomic memory operations as function calls. However, to achieve atomicity, the memory interface must reject any other memory access request during their execution. This behavior is obtained by slightly modifying the interconnection structure among MIs and control logic. In the basic design (Figure 7.8), when the interface accepts a memory operation, it activates the corresponding *UNBD* module. The *UNBD* module locks the shared resource, until the *Ready* signals notifies the completion of the memory access. For atomic memory operations, the *Ready* signal is substituted with the *Done* signal, which notifies the completion of the whole function. As a result, the UNBD module does not release the lock, thus ensuring atomicity. This approach provides two main advantages. First, it is general: it allows extending the set of atomic operations, because they are synthesized through the HLS flow itself. Second, it is independent from the target platform: the accelerator design completely embeds concurrency management, without relying on or being constrained by external hardware. Since the set of atomic memory operation is limited, it is also possible to implement them as hand-tuned custom IPs to include in the resource library.

Distributed and multi-ported memories In parallel applications, memory bandwidth can easily become a bottleneck for the execution latency, especially for memory bound and irregular applications. In fact, if concurrent kernels execute subsequent memory accesses, with few (if any) computations between them, then the kernels will be stalled for most of the application execution latency, waiting for resource availability. Increasing the memory bandwidth may reduce this issue: this is typically obtained by adding further memory banks (distributed or partitioned memories with multiple ports) as the constraints (pins, power, etc.) allow. In these conditions, multiple tasks can concurrently access distributed or partitioned memories, provided that they do not address the same partition. Distributed and multi-ported memories are supported by extending the memory interface as shown in Figure 7.10. Given N available memory partitions, the interface provides N input and output ports for each data (e.g., address, data to write) or control (e.g., LOAD/STORE selectors) signal, each associated with a particular memory partition. The interface manages the concurrency on the N memory ports through N dedicated arbiters. When a function module forwards a memory access request, the interface is responsible for identifying the memory partition to address, and for routing accordingly the request to the proper arbiter, which checks for resource availability. If the memory interface accepts the request, it forwards data and control signals to the proper output ports through the steering logic. The memory interface selects the partition by analyzing the memory address in input. In our current flow, the memory interface identifies the proper line by filtering $log_2(N)$ bits from the address (*line selector* module). It is not considered a fixed tag size (which must consider an upper bound for N) to reduce as much as possible the cost of the steering logic.

Differences between hierarchical and centralized MIC implementations There are few differences between the hierarchical and the Centralized MIC (CMIC, Chapter 6) designs. The HMIC, is characterized by a finer granularity: arbiters, which are completely embedded in the centralized MIC, appear in the hardware description of synthesized modules at each level of the call graph. The resulting structure is regular, efficient, and relatively easy to be allocated through an automated tool. On the other hand, it requires a complete integration in the HLS flow, for example for binding interconnections. The CMIC instead, is easier to be directly integrated on existing RTL and HLS synthesis processes, since it is designed as an IP allocated at the top level. The other key difference refers to the management and implementation of atomic operations. The CMIC embeds their associated modules, in the HMIC they are external components synthesized directly by the HLS tool, or implemented as custom library components. Even in this case, a tighter integration in the HLS flow is required.

7.5 Experimental Evaluation

The described techniques have been implemented further extending the Bambu framework (Section 2.4). The resulting flow has been evaluated on a set of parallel applications, instrumented with OpenMP pragmas [7]. All the applications are representative of the general template presented in Figure 7.1, which exposes coarse grained parallelism. *ADD* features 5 concurrent kernels, while all the other applications present a parallelism degree of 6. All the programs, except *PERFECTHASH*, have a very similar 2-level hierarchy call structure. *PERFECTHASH*, instead, employs nested function calls. Area and latency of the resulting designs have been compared against the release versions of *Bambu* (0.9.0) and *LegUp* (ver 3.0) [37]. Both are state-of-the-art HLS frameworks, implementing single execution

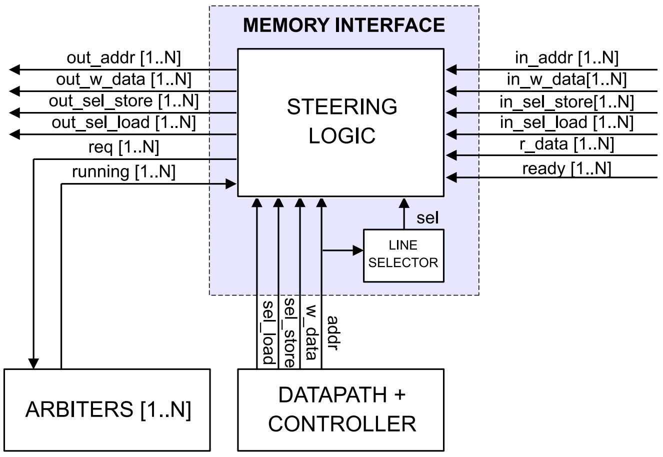


Figure 7.10: *Memory interface for distributed and multi ported memories.*

flow approaches based on the construction of centralized FSMs. Both are freely available on the Internet. The standard flow of Bambu 0.9 provides a baseline for the proposed approach, and allows understanding the performance benefits and area overheads only due to parallel execution. The use of Bambu enables a better evaluation of the proposed methodology for the HLS of parallel specification, independently from the framework design or implementation details such as the front-end compiler, the technology library and others. For Bambu, code has been compiled with GCC at the *O1* optimization level, which enables constant propagation and loop unrolling. The target frequency has been left to its default value of 100 MHz. The shared data is bound to external memories, and the latency model for the memory accesses is of 2 clock cycles for loads and of 1 clock cycle for stores. Nevertheless, the number of read operations is predominant in all the synthesized applications. The approach has also been compared against LegUp, because it has demonstrated to provide comparable or better QoR with respect to industrial tools [37]. The LegUp framework also offers support for parallel execution, which exploits both pthreads primitives and OpenMP pragmas. However, while LegUp synthesizes concurrent kernels as custom accelerators, it delegates the management of the generated accelerators themselves to a soft-core processor. This introduces significant overheads for both area and performance with respect to fully custom solutions, making the approaches hardly comparable. In these settings, the synthesized applications need on average more clock cycles to complete execution, while operating at lower (more than 10x lower for some benchmarks) frequencies when compared to fully custom accelerators. Thus, for a fair comparison, this feature has not been enabled. The applications have been synthesized with LegUp using the default parameters. LegUp exploits LLVM as the front-end, and considers *O3* as optimization level. The synthesis targets an operating frequency of 100 MHz. Global variables and arrays are mapped to BRAMs (Block RAMs), directly integrated in the target reconfigurable device. There are three main aspects, which may affect both area and performance, to carefully take into consideration when comparing the designs produced by LegUP and Bambu:

- the different targeted memories, i.e. local BRAMs for LegUp, external memories for Bambu;
- the front-end compilers, i.e. LLVM for LegUp, GCC for Bambu;
- the target devices: LegUp is optimized for Altera devices, while Bambu produces designs that are synthesizable on both Xilinx and Altera FPGAs.

To highlight the impact of memory accesses on the performance in parallel execution, each application has been synthesized through two accelerator designs with the proposed flow: one that interfaces with a single memory bank, and another that interfaces with multiple memory banks. Designs with multiple memory banks interface with 4 independently accessible banks.

7.5.1 Performance Evaluation

Execution latencies have been evaluated by simulating the designs with Mentor ModelSim. Table 7.1 shows the simulation results, reporting the execution latencies in terms of clock cycles for the accelerators synthesized through Legup, Bambu, and the proposed flow. The proposed approach provides a performance gain for all the benchmarks. However, for designs with a single memory bank, the gain is extremely variable and depends on the application characteristics. With this setup, *HYSTOGRAM* and *LOS* provide the highest speedup with respect to the conventional Bambu flow and LegUp. These two benchmarks present parallel kernels that perform both computation and memory accesses. The parallel execution of the kernels provides performance gains that are, however, reduced by the lack of concurrency on memory operations. This issue heavily impacts memory bound applications, such as *MATRIXMULT* and *MATRIXTRANS*: their parallel kernels are characterized by subsequent memory access, and few arithmetic operations. For *MATRIXMULT*, it is reported a limited speedup with

Table 7.1: *Simulation results: execution latencies in terms of clock cycles, and speedups for the proposed flow, interfacing single and multi ported memories (4 channels). All the designs target an operating frequency of 100MHz.*

				Speedup			***Speedup***	
Benchmark	**LegUp**	**Bambu**	**Proposed**	**vs LegUp**	**vs Bambu**	**Proposed,4ch**	**vs LegUp**	**vs Bambu**
ADD	38,009	30,027	22,019	1.73	1.36	6,024	9.46	4.98
DOT PRODUCT	29,011	36,031	25,335	1.15	1.42	6,041	5.71	5.96
HISTOGRAM	281,182	245,507	89,584	3.14	2.74	43,570	9.68	5.63
LOS	188,297	236,286	82,345	2.77	2.29	42,036	5.43	5.62
MATRIXMUL	14,592	18,302	15,262	0.96	1.2	5,590	2.93	3.27
MATRIXTRANS	36,881	55,340	45,089	0.82	1.23	15,566	3.55	3.56
PERFECTHASH	264,020	336,143	214,419	1.23	1.57	86,918	4.56	3.87
Average speedup				***1.62***	***1.77***		***4.29***	***4.7***

respect to Bambu (1.2) and no performance gain with respect to LegUp. The single memory bank prevents kernels to run concurrently, since there is not enough computation to perform while waiting for the availability of the memory. The results highlight the importance of designing HLS flows that exploit both task-level and memory-level (data-level) parallelism. The latter is obtained by taking advantage of memory structures that allow executing multiple memory operations in parallel. In fact, when targeting a memory composed of 4 memory banks, the proposed flow provides significantly improved performance. For most of the applications, it is obtained a speedup close to the number of kernels (i.e., 5 for *ADD*, 6 for *DOTPRODUCT*, *HISTOGRAM* and *LOS*). This result demonstrates that the generated accelerators can exploit all the available parallelism, saturating the memory bandwidth. The linear speedup also demonstrates that the memory interface for multiple memory banks does not introduce any delay. Accelerators with multiple memory banks for *MATRIXMULT*, *MATRIXTRANS* and *PERFECTHASH* show a slight reduction in the speedups with respect to the other applications. The reasons are hot-spotting on the memory resources and memory access patterns, in particular for *PERFECTHASH*. In fact, when multiple memory operations address the same memory bank, they cannot run in parallel. *PERFECTHASH* has an irregular memory access pattern that generates a higher number of structural conflicts with respect to the other benchmarks, which instead access the memory with more regularly strided patterns. *MATRIXMULT* provides the lowest performance gain also when interfacing to a memory composed of 4 banks. We also tried to synthesize this application using 8 memory partitions, and obtained an execution latency of 3850 cycles, with a speedup of 1.45 with respect to the 4-bank design. On average, when targeting the multi-ported memory, the flow provides a speedup of 4.7 with respect to accelerators generated by Bambu with a single execution flow. This is a valuable result, since the maximum speedup reachable is 5.86 in ideal settings, i.e., assuming that all the computation is bound to parallel kernels and that there are only non-conflicting memory accesses. Instead, in the considered benchmarks, even if most of the computation is performed by the parallel tasks, there still is a serial fraction of the code. In addition, as demonstrated by the simulation results, conflicts on memory resources may degrade performance.

7.5.2 Area Evaluation

Area requirements of the accelerators have been evaluated by synthesizing the designs with Altera Quartus version 11.1 SP2. The target device is an

Table 7.2: *Synthesis results: number of Flip Flop (FF) registers and Logic Elements (LEs) required by the generated designs. Area overhead ratios consider the number of required LEs.*

Benchmark	***LegUp***		***Bambu***		***Proposed***		***Area Overhead Ratio***		***Proposed,4ch***		***Area Overhead Ratio***	
	#FF	**#LE**	**#FF**	**#LE**	**#FF**	**#LE**	**vs LegUp**	**vs Bambu**	**#FF**	**#LE**	**vs LegUp**	**vs Bambu**
ADD	992	1364	379	546	1007	1431	1.05	2.62	1022	2462	1.8	4.51
DOT PRODCUT	1212	1841	521	704	1793	2461	1.34	3.5	1811	3714	2.02	5.28
HISTOGRAM	1681	2811	1902	2658	5580	8666	3.08	3.26	5604	10514	3.74	3.96
LOS	1809	3128	1214	1999	5908	9947	3.18	4.98	5962	12283	3.93	6.14
MATRIXMULT	1577	3028	761	1259	2229	3464	1.14	2.75	2334	5601	1.85	4.45
MATRIXTRANS	1645	3079	609	763	2337	3284	1.07	4.3	2354	5531	1.8	7.25
PERFECTHASH	483	911	620	755	2381	3095	3.4	4.1	2453	5108	5.61	6.77
	Average Area Overhead Ratio (LEs)						***2.04***	***3.64***			***2.96***	***5.48***
	Average Performance Gain/Area Overhead ratio						***0.79***	***0.49***			***1.45***	***0.86***

Altera Cyclone II EP2C70F896C6 FPGA platform. All the designs generated by Bambu met the frequency constraint of 100 MHz when mapped onto the target device. Three designs (LOS, MATRIXMULT and MATRIXTRANS) generated by LegUp achieved a maximum frequency, on average, close to 73 MHz, thus violating the timing constraint. Table 7.2 summarizes the synthesis results, reporting the number of allocated Flip Flop (FF) registers and Logic Elements (LEs). Area overheads are computed only considering the number of LEs, because they embed information on both register and logic utilization. Indeed, on Cyclone II FPGAs, each LE includes a 4-input look-up table and a register. Area evaluation is mainly focused on Bambu, because the designs are more similar from the architectural point of view. As expected, the area requirements for the accelerators featuring parallel execution increase with the number and the complexity of the replicated kernels. *ADD*, which only includes 5 parallel kernels of limited complexity, presents the smallest area overhead ratio (2.62). Kernel replication increases the overall area of a factor less than 3. The reason is that only the kernel modules are replicated, thus impacting just on a fraction of the overall area demands. For the other benchmarks, our flow allocates 6 concurrent kernels. In this case, the resulting accelerators are 3.8 times larger in average. The area requirements increase when generating designs that interface with 4 memory banks. The reason is the introduction of steering logic to dynamically route memory operations to the proper memory bank. The flow must also replicate, for each memory bank, the arbiters for managing concurrency on the shared memories. However, they have very low area requirements and, thus, do not significantly impact on the overall area demands. In fact, both RM and UNBD modules require a number of combinational logic cells close to the number of inputs. In addition, UNBD modules require a number of logic cell registers equal to the number of memory banks. In average, the proposed flow generates accelerators 3.64 times larger for a single memory bank and 5.48 times larger for multiple memory banks with respect to Bambu. This allows understanding the cost of the memory interface structure. However, performance gains are, comparatively, much higher than the area overheads. Table 7.2 also shows the area overhead/performance ratios, which are useful to evaluate trade-offs between costs and benefits. For designs with a single memory bank, the average ratio is 0.49. The ratio increases to 0.86 for designs with multiple memory banks. The metric confirms that, when considering multiple memory banks, the additional area costs have much higher returns in terms of performance improvement.

7.6 Conclusions

This chapter has presented a methodology for the HLS of parallel C specifications. The designed framework generates hardware accelerators that support concurrent execution flows, enabling TLP exploitation. TLP is exploited through the proposed adaptive controller, which can independently manage operations and/or tasks. The framework identifies the parallel tasks through annotations in the source code, and takes advantage of compiler optimizations. It manages concurrency between tasks on shared memories by synthesizing a hierarchical memory interface, which features custom control logic that avoid structural conflicts. The hierarchical memory interface supports multi-ported memories, enabling concurrent memory accesses, and provides support for atomic memory operations. The implemented HLS tool generates significantly faster accelerators, with respect to more conventional approaches, with an average speedup of 4.7. For some applications, the speedups provided are close to linearity. The replication of modules implementing the parallel tasks, and the insertion of steering logic for the dynamic management of the memory resources, introduce limited area overheads (average area increase of 5.48 times) with respect to the performance improvements. The designed framework has demonstrated encouraging experimental results. However, some opportunities for improvement have been identified. Currently, OpenMP pragmas are not considered in the compilation process, but treated as simple markers in the input code. Preliminary work in this direction has provided support for *parallel sections* pragmas, but a wider coverage of the OpenMP syntax is needed. The possibility of directly compiling OpenMP code will enable further optimization in the front-end phase. Another aspect that will be investigated is the integration of a DSE engine in the flow selection step. Currently the framework allows the expert user to select, at each hierarchy level, the architecture to target, through configuration files. Techniques for the analysis of the EPDG for the identification of coarse grained parallelism have already been developed, but not still integrated in a DSE step. This will provide better QoR for complex applications.

CHAPTER 8

Conclusions

The increasing complexity of hardware systems design has encouraged the design community to raise the abstraction level of synthesis methodologies above RTL. The Electronic System Level (ESL) design automation has been identified as the main component for boosting the semiconductor industry in the next generations: in these settings, High Level Synthesis plays a central role, allowing the automatic synthesis of hardware components from their behavioral description, usually provided as C-code. Research efforts and constant improvements in HLS have made modern tools suitable for adoption in industry, shortening the design cycle, reducing the development costs, and providing Quality of Results often comparable with hand-written designs. However, as highlighted by the analysis of the state of the art, most of the proposed approaches are affected by common limitations:

- they usually target architectural solutions based on the definition of a centralized finite state machine, without properly exploring design alternatives. On the other hand, techniques which consider promising architectural models, such as parallel controllers, are constrained by the adoption of particular specification languages;

- they focus on instruction level parallelism exploitation for improving performance, and often ignore coarser granularities;
- they lack efficient support and abstraction for external and shared memories;
- they provide limited support for parallel specifications, and the considered models for supporting Task Level Parallelism often require the introduction of soft cores or custom schedulers in the resulting designs, impacting on both area and performance.

This thesis work has considered all these aspects, proposing the design of novel hardware components and synthesis methodologies to mitigate the highlighted issues. Next section summarizes the main novel contributions.

8.1 Novel Contributions

8.1.1 The Parallel Controller Design

Chapter 4 presented a parallel controller architecture, as an alternative to the typical centralized FSM model, commonly adopted in HLS. The controller design consists of a set of interacting modules, communicating through a lightweight token-based schema. Each element is associated with a particular operation or task, and is responsible for their execution. The controller elements verify at runtime when resource and dependency constraints are satisfied, enabling as soon as possible execution. The resulting accelerators are thus adaptive, and feature dynamic scheduling. In opposition to the FSM model, which is inherently serial, the adaptive architecture is able to manage multiple execution flows, improving parallelism exploitation at different granularities. It also provides natural support for variable latency operations, such as speculative operations, memory accesses and function calls.

8.1.2 High Level Synthesis of Adaptive Hardware Components

Chapter 5 described a complete HLS flow targeting the proposed adaptive accelerators. The flow takes advantage of both the architectural model and a front-end analysis phase, which allows identifying most of the parallelism available in the input specification. The absence of a pre-computed scheduling has required the definition of novel algorithms to perform the various steps of the HLS flow. They have been designed as general as possible, making their adoption suitable also for other dynamic scheduling techniques. Among them, the proposed liveness analysis algorithm allows the definition of conflict free bindings regardless of the runtime execution order: its formal definition makes its adoption not limited to hardware synthesis methodologies. The synthesis approach has been evaluated comparing the dynamically scheduled accelerators to conventional designs, on a set of typical HLS benchmarks: experimental results reported significant speedups, causing only limited area overheads with respect to the obtained performance improvements.

8.1.3 Memory Interface Controller

Chapter 6 introduced a synthesis approach based on the implementation of an adaptive Memory Interface Controller (MIC). The MIC, designed as a custom parameterizable IP, introduces an abstraction layer between hardware accelerators design and external memory structure. It supports distributed and multi-ported memories, which provide very high bandwidths for fine-grained memory operations; since it manages both synchronization and concurrency among the memory resources, it allows multiple accesses to execute at the same time, and introduces an abstraction layer which facilitates the design of custom accelerators. It also provides support for atomic memory operations. Its adoption results particularly profitable for the synthesis of memory intensive algorithms, and among them, in particular for irregular applications. In fact, irregular applications present unpredictable and fine-grained (single word) memory accesses, exploit large data sets, are mostly memory bandwidth limited and have high synchronization intensity. The approach has been evaluated on the Breadth First Search (BFS) algorithm, exploring several trade-offs in terms of number of kernels and number of memories, showing how the MIC allows exploiting the parallelism available in the algorithm, maximizing concurrency of memory operations, and thus improving the system bandwidth utilization. To prove the suitability of the design for HLS, the BFS kernels have been automatically generated through the designed HLS flow: this has also highlighted the capability of the proposed approach to parallelize memory operations, modeled as variable latency operations. However, the MIC has been designed to be compatible with different specific target architectures or synthesis flows (e.g. RTL vs HLS).

8.1.4 Support for Annotated Parallel Specifications

Chapter 7 presented a methodology for the HLS of annotated parallel C specifications, taking advantage of (and improving) the already mentioned proposed techniques. The resulting HLS framework generates hardware accelerators that support concurrent execution flows, enabling TLP exploitation. As opposed to other approaches, the generated components do not require the instantiation of external custom schedulers or processors for managing the concurrent execution, definitively improving performance and area utilization. The generated designs are hierarchical, reflecting the call structure of the specifications. According to the code characteristics, the tool can choose to implement each function following both the parallel controller approach, for TLP exploitation, or a more conventional FSM based flow, for serial tasks. The framework identifies the parallel tasks through pragma annotations in the source code, and takes advantage of compiler optimizations such as constant propagation and loop unrolling, which improve TLP identification. It manages concurrency between tasks on shared memories by synthesizing a restructured hierarchical implementation of the MIC, which embeds custom control logic that avoid structural conflicts. As its centralized counterpart, the hierarchical memory interface supports multi-ported memories, enabling concurrent memory accesses, and provides support for atomic memory operations. The implemented HLS tool generates significantly faster accelerators, with respect to more conventional approaches. For some applications, the speedups provided are close to linearity. The replication of modules implementing the parallel tasks, and the insertion of steering logic for the dynamic management of the memory resources, introduce limited area overheads with respect to the performance improvements.

8.2 Further Development

The proposed techniques provide unique features, such as the capability of managing multiple execution flows and the support for variable latency operations; in addition, the experimental evaluation have shown encouraging results. However, some aspects need further improvements and investigations. First, when working at instruction level granularity, the designed HLS flow generates accelerators with area overheads, which are acceptable compared to the obtained performance improvements, but still considerable. For this reason, the proposed binding strategies should be refined and improved. Additional effort is also required to improve the identification of coarse grained parallelism, since it is constrained by, often over conservative, dependences analysis. The framework is able to generated accelerators coupling both FSM and parallel controllers, depending on the code characteristics: automatic code partitioning techniques may be designed to identify, within a given function, serial portions of code which may run concurrently, regardless of the call structure: this will improve TLP identification and exploitation, increasing the QoR for complex applications. Focusing on the support for parallel specifications, the flow can be improved through a complete support of OpenMP directives, currently considered just as markers in the source code, and not involved in the compilation process.

8.3 Future Research Directions

The techniques proposed in this thesis may provide significant benefits in several application domains, and among them, especially in the emerging field of High Performance Reconfigurable Computing (HPRC). HPRC applications are parallel, often irregular, and usually process large amounts (Terabytes) of data. HPRC is quickly being supported by novel hybrid architectures, such as the Convey platforms briefly introduced in Chapter 6, or the recently announced IBM Power8 system [6]. Hybrid architectures couple commodity processors, reconfigurable devices, and large high-bandwidth memory systems, shared among the various components. They promise good performance through hardware acceleration, and equally important, reduced power consumption. Both these aspects may have tremendous impacts, since operative costs of HPC systems are comparable or higher, sometimes much higher, than the costs of the actual hardware. Hybrid platforms already proved their potentiality on several applications, such as graph exploration algorithms. However, their exploitation still require great development effort, for the design, implementation and integration of the custom accelerators. While acceptable in specific domains, this represent a strong limitation in others. For example, financial and risk management algorithms, which can profitably be accelerated by means of custom components, vary on a monthly basis. Knowledge discovery and social network analysis, may require the definition of tens or hundreds of different routines for processing the data. It is clear that HLS may have a great impact in HPRC, and the techniques proposed in this work, for parallelism exploitation and memory management, could represent a significant contribution in this research direction.

Bibliography

[1] Bambu: A Free Framework for the High-Level Synthesis of Complex Applications. http://panda.dei.polimi.it/?page _id=31.

[2] Bdti high-level synthesis tool certification program results, http://edac.org/initiatives/committees/mss.

[3] Bdti high-level synthesis tool certification program results, http://www.bdti.com/resources/benchmarkresults/hlstcp.

[4] C to verilog - automatic circuit design. http://c-to-verilog.com/howtos.html.

[5] Cadence c-to-silicon white paper, 2008 available: http://www.cadence.com/rl/resources/ technical_papers/c_tosilicon_tp.pdf.

[6] Hot Chips, Power8 discussed. https://www.ibm.com/developerworks/community/blogs/fe313521-2e95-46f2-817d-44a4f27eba32/entry/hot_chips_power8_discussed.

[7] Legup official site: http://legup.eecg.utoronto.ca/.

[8] Modelsim - advanced simulation and debugging. http://www.model.com.

[9] Nangate open cell library. http://www.nangate.org.

[10] Panda framework official site. http://panda.dei.polimi.it/.

[11] Synopsys design compiler. http://www.synopsys.com.

[12] Synopsys synphony, [online] available: http://www.synopsys.com/systems/blockdesign/ hls/-pages/default.aspx.

[13] A. Bardsley. *"Implementing Balsa Handshake Circuits"*. PhD thesis, Univerity of Manchester, 2000.

[14] A. Crews and F. Brewer. "Controller Optimization for Protocol Intensive Applications". *Proc. of EURO-DAC*, 1996.

[15] A. Seawright and F. Brewer. "Clairvoyant: A Synthesis System for Production-Based Specification". *IEEE Trans. on VLSI*, pages 172–185, June 1994.

[16] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690, dec 1974.

Bibliography

[17] Alfred V. Aho. *"Algorithms for finding patterns in strings"*, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.

[18] C. Andriamisaina, P. Coussy, E. Casseau, and C. Chavet. High-level synthesis for designing multimode architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(11):1736–1749, 2010.

[19] Andrew W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, 1997.

[20] A.R. Karlin, H.W. Trickey, and J.D. Ullman. "Experience with a Regular Expression Compiler". *ICCD*, pages 656–665, 1983.

[21] B. Akers Sheldon. "Binary Decision Diagrams". *IEEE Transactions on Computers*, pages 509–516, June 1978.

[22] David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Queue*, 11(2):40:40–40:52, feb 2013.

[23] M.R. Barbacci, Department of Computer Science, Carnegie-Mellon University Electrical Engineering, G.E. Barnes, R.G.G. Cattell, and D.P. Siewiorek. *The ISPS Computer Description Language: The Symbolic Manipulation of Computer Descriptions*. Carnegie-Mellon University, Computer Science Department, 1978.

[24] Rami Beidas and Jianwen Zhu. Scalable interprocedural register allocation for high level synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC ’05, pages 511–516, New York, NY, USA, 2005. ACM.

[25] R.A. Bergamaschi, R.A. O’Connor, L. Stok, M.Z. Moricz, S. Prakash, A. Kuehlmann, and D. S. Rao. High-level synthesis in an industrial environment. *IBM Journal of Research and Development*, 39(1.2):131–148, 1995.

[26] Berkeley Design Technology Incorporation. "FPGAs for DSP: An Independent Perspective". *Xilinx Workshop, Embedded Systems Conference*, April 3, 2007.

[27] B. Betkaoui, Yu Wang, D.B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15, 2012.

[28] J. Biesenack, M. Koster, A. Langmaier, S. Ledeux, S. Marz, M. Payer, M. Pilsl, S. Rumler, H. Soukup, N. Wehn, and P. Duzy. The siemens high-level synthesis system callas. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 1(3):244–253, 1993.

[29] Benoit Boissinot, Florian Brandner, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello. A non-iterative data-flow algorithm for computing liveness sets in strict ssa programs. In *Programming Languages and Systems*, pages 137–154. Springer, 2011.

[30] Benoit Boissinot, Florian Brandner, Alain Darte, Benoît Dupont de Dinechin, and Fabrice Rastello. A non-iterative data-flow algorithm for computing liveness sets in strict ssa programs. In *Programming Languages and Systems*, pages 137–154. Springer, 2011.

[31] Thomas Bollaert. "Catapult Synthesis: A Practical Introduction to Interactive C Synthesis". In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 29–52. Springer Netherlands, 2008. 10.1007/978-1-4020-8588-8_3.

[32] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh. Optimal register sharing for high-level synthesis of ssa form programs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(5):772 – 779, may 2006.

[33] Inc (BDTI By the staff of Berkeley Design Technology. "An Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs". Technical report, Berkeley Design Technology, Inc, 2010.

[34] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. OpenMP extensions for FPGA accelerators. In *SAMOS IX: International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 17–24, 2009.

[35] R. Camposano. Design process model in the yorktown silicon compiler. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 489–494, 1988.

[36] R. Camposano and W. Wolf. *"High-Level VLSI Synthesis"*. Kluwer (now Springer), Dordrecht, 1991.

[37] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *FPGA '11: the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–36, 2011.

[38] D. Capalija and T.S. Abdelrahman. An Architecture for Exploiting Coarse-grain Parallelism on FPGAs. In *FPT 2009: International Conference on Field-Programmable Technology*, pages 285–291, 2009.

[39] Carl A. Petri. "Communication with automata". *DTIC Research Report*, 1966.

[40] V.G. Castellana and F. Ferrandi. Speeding-up memory intensive applications through adaptive hardware accelerators. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1415–1416, Nov 2012.

[41] Vito Giovanni Castellana and Fabrizio Ferrandi. Applications acceleration through adaptive hardware components. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 2274–2277, Washington, DC, USA, 2013. IEEE Computer Society.

[42] Vito Giovanni Castellana and Fabrizio Ferrandi. An automated flow for the high level synthesis of coarse grained parallel applications. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 294–301, Dec 2013.

[43] Vito Giovanni Castellana and Fabrizio Ferrandi. Scheduling Independent Liveness Analysis for Register Binding in High-Level Synthesis. In *DATE 2013: Design, Automation and Test in Europe*, pages 1571–1574, 2013.

[44] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. A synthesis approach for mapping irregular applications on reconfigurable architectures. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Technical Program Posters*, 2013.

[45] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. An Adaptive Memory Interface Controller for Improving Bandwidth Utilization of Hybrid and Reconfigurable Systems. In *DATE 2014: Design, Automation and Test in Europe*, 2014.

[46] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. C-based high level synthesis of adaptive hardware components. In *DATE 2014: Design, Automation and Test in Europe, PhD Forum*, 2014.

[47] Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi. High level synthesis of memory bound and irregular parallel applications with bambu. In *International Workshop on Electronic System-Level Design towards Heterogeneous Computing*, 2014.

[48] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[49] A.P. Chandrakasan, M. Potkonjak, J. Rabaey, and R.W. Brodersen. Hyper-lp: a system for power minimization using architectural transformations. In *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, pages 300–303, 1992.

[50] Deming Chen and Jason Cong. Register binding and port assignment for multiplexer optimization. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 68–73, Piscataway, NJ, USA, 2004. IEEE Press.

[51] Chris Papachristou and Yusuf Alzazeri. "A method of distributed controller design for RTL circuits". *Proc. of Design Automation and Test in Europe*, 1999.

[52] Convey Computer. Convey computer doubles graph500 performance, develops new graph personality. available at http://www.conveycomputer.com/files/2413/5095/9078/sc11_graph500_release.final.pdf.

[53] Convey Computer. Convey MX Series. Architectural Overview. available at http://www.conveycomputer.com.

[54] Convey Computer. New Convey MX Demonstrates Leading Power/Performance on Graph 500 Benchmark. available at http://www.conveycomputer.com/files/9613/5284/0776/sc12_graph500_release_final.pdf.

[55] J. Cong, Muhuan Huang, and Yi Zou. Accelerating fluid registration algorithm on multi-fpga platforms. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 50–57, 2011.

[56] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[57] J. Cortadella and W. Reisig. *Applications and Theory of Petri Nets 2004: 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*. Number v. 25 in Lecture Notes in Computer Science. Springer, 2004.

[58] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From OpenCL to High-Performance Hardware on FPGAs. In *FPL '12: 22nd International Conference on Field Programmable Logic and Applications*, pages 531–534, 2012.

[59] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Science Engineering*, 5(1):46–55, 1998.

[60] C. Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *IEEE Computer*, 42(12):36–42, 2009.

[61] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with flopoco. *Design Test of Computers, IEEE*, 28(4):18–27, 2011.

[62] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The olympus synthesis system. *Design Test of Computers, IEEE*, 7(5):37–53, 1990.

[63] G. De Micheli and D.C. Ku. Hercules-a system for high-level synthesis. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 483–488, 1988.

[64] A.A. Del Barrio, S.O. Memik, M.C. Molina, J.M. Mendias, and R. Hermida. A distributed controller for managing speculative functional units in high level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(3):350 –363, march 2011.

[65] A.A. Del Barrio, M.C. Molina, J.M. Mendias, R. Hermida, and S.O. Memik. Using speculative functional units in high level synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1779 –1784, march 2010.

[66] M.K. Dhodhi, F.H. Hielscher, R.H. Storer, and J. Bhasker. "Datapath Synthesis Using a Problem-Space Genetic Algorithm". *IEEE Transac- tions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, August 1995.

[67] S.W. Director, A.C. Parker, D.P. Siewiorek, and Jr. Thomas, D. A design methodology and computer aids for digital vlsi systems. *Circuits and Systems, IEEE Transactions on*, 28(7):634–645, 1981.

[68] P. Dziurzanski and V. Beletskyy. Defining Synthesizable OpenMP Directives and Clauses. In Marian Bubak, GeertDick Albada, PeterM.A. Sloot, and Jack Dongarra, editors, *ICCS 2004: Computational Science*, volume 3038 of *Lecture Notes in Computer Science*, pages 398–407. Springer Berlin Heidelberg, 2004.

[69] E. J. McCluskey. *"Introduction to the Theory of Switching Circuits"*, chapter 2. McGraw-Hill, 1965.

[70] T. El-Ghazawi, E. El-Araby, Miaoqing Huang, K. Gaj, V. Kindratenko, and D. Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, 2008.

[71] John P. Elliott. *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[72] John Feo, David Harper, Simon Kahan, and Petr Konecny. ELDORADO. In *CF ’05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM Press.

[73] Fredrick J. Hill and Gerald R. Peterson. *"Introduction to the Theory of Switching Circuits"*. McGraw-Hill, 1965.

[74] G. Berry and G. Gonthier. "The ESTEREL synchronous programming language: design, semantics, implementation". *Science of Computer Programming*, 19(2):87–152, 1992.

[75] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. "Wavesched: a novel scheduling technique for control-flow intensive designs". *IEEE Trans. on CAD*, 18(5):505–523, May 1999.

[76] D. Gajski et al. *"High Level Synthesis: An Introduction to Chip and System Design"*. Kluwer (now Springer), 1992.

[77] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[78] Geiger and Muller-Wipperfurth. "FSM decomposition revisited: algebraic structure theory applied to MCNC benchmark FSMs". *28th ACM/IEEE Design Automation Conference (DAC ’91)*, pages 182–185, 1991.

[79] W. Geurts et al. *"Accelerator Data-Path Synthesis for High-throughput Signal Processing Applications"*. Kluwer Academic Publishers, 1996.

[80] A. Girault, Bilung Lee, and E.A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, Jun 1999.

[81] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):166–178, 1992.

Bibliography

[82] J. Granacki, D. Knapp, and A. Parker. The adam advanced design automation system: Overview, planner and natural language interface. In *Design Automation, 1985. 22nd Conference on*, pages 727–730, 1985.

[83] Zhi Guo, Betul Buyukkurt, John Cortes, Abhishek Mitra, and Walild Najjar. A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming*, 36(5):493–520, 2008.

[84] Rajesh K. Gupta and Forrest Brewer. *"A Retrospective in High-Level Synthesis"*. Springer, 2008, chapter 2.

[85] S. Gupta, R.K. Gupta, N.D. Dutt, and A. Nicolau. *"SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits"*. Kluwer, Dordrecht, 2004.

[86] Sumit Gupta, Rajesh Kumar Gupta, Nikil D. Dutt, and Alexandru Nicolau. Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 9(4):441–470, October 2004.

[87] H. P. Zeiger. *"Loop-free synthesis of finite-state machines"*. PhD thesis, MIT, Department of Electrical Engineering, Cambrige, Mass., September 1961.

[88] Robert J. Halstead, Jason Villarreal, and Walid Najjar. Exploring irregular memory accesses on fpgas. In *Proceedings of the first workshop on Irregular applications: architectures and algorithms*, IAAA ’11, pages 31–34, 2011.

[89] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195, 2008.

[90] A. Hemani, B. Karlsson, M. Fredriksson, K. Nordqvist, and B. Fjellborg. Application of high-level synthesis in an industrial project. In *VLSI Design, 1994., Proceedings of the Seventh International Conference on*, pages 5–10, 1994.

[91] C.Y. Hitchcock and D.E. Thomas. "A method of automatic data path synthesis". *Design Automation Conference*, 1983.

[92] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, aug 1978.

[93] Chao Huang, S. Ravi, A. Raghunathan, and N.K. Jha. Generation of Heterogeneous Distributed Architectures for Memory-Intensive Applications Through High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(11):1191–1204, 2007.

[94] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data path allocation based on bipartite weighted matching. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC ’90, pages 499–504, New York, NY, USA, 1990. ACM.

[95] Q. Huang, R Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *FCCM 2013: the 21th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 89–96, 2013.

[96] J. Eppling and D. Thomas. "Multiple Controller Synthesis for Reducing Control Path Delays". *SRC TECHCON*, Sept. 1993.

[97] J. Hartmanis. "Symbolic analysis of a decomposition of information processing". *In Inform. Control*, June 1960.

[98] J. Hartmanis and R. E. Stearns. *"Algebraic Structure Theory of Sequential Machines"*. Prentice-Hall, Englewood CIiffs, N. J., 1966.

[99] R. Jain, A. Parker, and N. Park. Module selection for pipelined synthesis. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 542–547, 1988.

[100] Rajiv Jain, K. Kücükcakar, M. J. Mlinar, and A. C. Parker. Experience with adam synthesis system. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, DAC '89, pages 56–61, New York, NY, USA, 1989. ACM.

[101] James Lyle Peterson. *"Petri Net Theory and the Modeling of Systems"*. Prentice Hall, 1981.

[102] João M. Fernandes, M. Adamski, and A.J. Proença. "VHDL generation from hierarchical petri net specifications of parallel controllers". *Proc. of IEEE Computer and Digital Techniques*, 1997.

[103] John E. Hopcroft and Jeffrey D. Ullman. *"Introduction to Automata Theory, Languages, and Computation"*. Addison-Wesley Publishing, 1979.

[104] Jörg Desel and Gabriel Juhás. *"What Is a Petri Net? Informal Answers for the Informed Reader"*, pages 1–25. Hartmut Ehrig et al., 2001.

[105] K. Bilinski, E. Dagless, and J. Mirkowski. " Synchronous parallel controller synthesis from behavioural multiple-process VHDL description". *Proc. of European Design Automation Conference*, Sept. 1996.

[106] K. Thulasiraman and M. N. S. Swamy. *"Acyclic Directed Graphs"*, chapter 5, Section 7. John Wiley and Son, 1992.

[107] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. Abdelrahman. A Multilevel Computing Architecture for Embedded Multimedia Applications. *IEEE Micro*, 24(3):56–66, 2004.

[108] V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, and M. Sivaraman. Pico: automatically designing custom computers. *Computer*, 35(9):39–47, 2002.

[109] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[110] D.W. Knapp. *"Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler"*. Prentice-Hall, Englewood Cliff, NJ, 1996.

[111] T.J. Kowalski and D.E. Thomas. "The VLSI design automation assistant: what's in a knowledge base". *Design Automation Conference*, 1985.

[112] Krzysztof Biliński, E.L. Dagless, J.M. Saul, and M. Adamski. "Parallel controller synthesis from a Petri net specification". *Proc. of European Design Automation Conference*, 1994.

[113] D. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 59 –64, jun 1990.

[114] David Ku and Giovanni De Micheli. Hardwarec-a language for hardware design version 2.0. *Computer*, 1990.

[115] David C. Ku and Giovanni De Micheli. Constrained resource sharing and conflict resolution in hebe. *Integration, the {VLSI} Journal*, 12(2):131 – 165, 1991.

[116] K. Kucukcakar, Chih-Tung Chen, Jie Gong, W. Philipsen, and T.E. Tkacik. Matisse: an architectural design tool for commodity ics. *Design Test of Computers, IEEE*, 15(2):22–33, 1998.

[117] F.J. Kurdahi and A.C. Parker. Real: A program for register allocation. In *Design Automation, 1987. 24th Conference on*, pages 210–215, 1987.

[118] G. Lakshminarayana, K.S. Khouri, and N.K. Jha. Wavesched: A Novel Scheduling Technique for Control-Flow Intensive Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(5):505–523, 1999.

[119] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. "Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions". *Design Automation Conference*, 1998.

[120] Chunho Lee, M. Potkonjak, and W.H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330 –335, dec 1997.

[121] Y. Y. Leow, C. Y. Ng, and W.F. Wong. Generating Hardware from OpenMP Programs. In *FPT 2006: IEEE International Conference on Field Programmable Technology*, pages 73–80, 2006.

[122] Silvia Lovergine and Fabrizio Ferrandi. Instructions activating conditions for hardware-based auto-scheduling. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 253–256, New York, NY, USA, 2012. ACM.

[123] M. Yoeli. "The cascade decomposition of sequential machines". *In IRE Trans. Electronic Computers*, April 1961.

[124] Hugo De Man. "Cathedral-II: A Silicon Compiler for Digital Signal Processing". *IEEE Design and Test*, vol. 3, no. 6, 1986, pp. 13-25.

[125] Grant Martin and Gary Smith. "High-Level Synthesis: Past, Present, and Future". *IEEE Design & Test of Computers*, pages 18–25, December 2009, vol. 26(4).

[126] P. Marwedel. The mimola design system: Tools for the design of digital processors. In *Design Automation, 1984. 21st Conference on*, pages 587–593, 1984.

[127] Michael C. McFarland, Alice C. Parker, and Raul Camposano. "Tutorial on High-level Synthesis". In *Conference, in Proc. of 25th ACM/IEEE Design Automaton*, pages 330–336, 1988.

[128] Michael Meredith. "A Look Inside Behavioral Synthesis". *EE Times*, aug 2004.

[129] B. Meyer, J. Schumacher, C. Plessl, and J. Forstner. Convey vector personalities - fpga acceleration with an openmp-like programming effort? In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 189–196, 2012.

[130] G. De Micheli. *"Synthesis and Optimization of Digital Circuits"*. McGraw-Hill, New York, 1994.

[131] Milind Girkar and Constantine D. Polychronopoulos. "The hierarchical task graph as a universal intermediate representation". *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[132] M.O. Rabin and D. Scott. "Finite Automata and their Decision Problems". *IBM Journal of Research and Development*, pages 115–125, 1959.

[133] J. Moreira. "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors". *PhD. thesis, Univ. of Illinois at Urbana-Champaign*, 1995.

[134] Silva M. Mueller. On the scheduling of variable latency functional units. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '99, pages 148–154, New York, NY, USA, 1999. ACM.

[135] Nancy Wu, Gary Smith EDA. "ESL Synthesis: Tips for Implementing a Viable ESL-Synthesis Flow". *EDN Magazine*, July 30, 2010.

[136] Stephen Neuendorffer and Kees Vissers. Streaming systems in fpgas. In Mladen Bereković, Nikitas Dimopoulos, and Stephan Wong, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5114 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin Heidelberg, 2008.

[137] Noam Chomsky. "Three Models for the Description of Language". *IRE Transactions on Information Theory*, pages 113–123, 1956.

[138] Ntsibane Ntlatlapa. "Verified High-Level Synthesis Front-End and Simulator Using Dependence Flow Graphs". Technical report, Auburn University (Auburn, Alabama, USA), 1999.

[139] Ntsibane Ntlatlapa. "High-Level Synthesis using Dependence Flow Graphs as the Intermediate Form". Technical report, Auburn University (Auburn, Alabama, USA), January 20, 1998.

[140] P. Ashar, S. Devadas, and R. Newton. "Optimum and heuristic algorithms for an approach to finite state machine decomposition". *IEEE Trans. on CAD*, 10:296–310, 1991.

[141] P. Coussy , C. Chavet , P. Bomel , D. Heller , E. Senn and E. Martin P. Coussy and A. Morawiec. *"High-Level Synthesis: From Algorithm to Digital Circuits"*. Springer, 2008.

[142] Preeti R Panda and Nikil D Dutt. 1995 high level synthesis design repository. In *Proceedings of the 8th international symposium on System synthesis*, pages 170–174. ACM, 1995.

[143] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs. In *SASP ’09: IEEE 7th Symposium on Application Specific Processors*, pages 35–42, 2009.

[144] A. Papakonstantinou, Yun Liang, J.A. Stratton, K. Gururaj, Deming Chen, W.-M.W. Hwu, and J. Cong. Multilevel Granularity Parallelism Synthesis on FPGAs. In *FCCM 2011: IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 178–185, 2011.

[145] N. Park and A. Parker. Sehwa: A program for synthesis of pipelines. In *Papers on Twenty-five years of electronic design automation*, 25 years of DAC, pages 595–601, New York, NY, USA, 1988. ACM.

[146] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, and J. Kim. The cmu design automation system - an example of automated data path design. In *Design Automation, 1979. 16th Conference on*, pages 73–80, 1979.

[147] A.C. Parker, J. Pizarro, and M. Mlinar. "MAHA: A program for datapath synthesis". *Proc. 23rd IEEE/ACM Design Automation Conference*, pp. 461-466, Las Vegas NV, June 1986.

[148] P. G. Paulin and J. P. Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *DAC ’89: the 26th ACM/IEEE Design Automation Conference*, pages 1–6, 1989.

[149] P.G. Paulin and J.P. Knight. "Scheduling and binding algorithms for high-level synthesis". *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

[150] P.G. Paulin, J.P. Knight, and E.F. Girczyc. Hal: A multi-paradigm approach to automatic data path synthesis. In *Design Automation, 1986. 23rd Conference on*, pages 263–270, 1986.

[151] Pierre G. Paulin and J.P. Knight. "Force-Directed Scheduling for the Behavioral Synthesis of ASICs". *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 1989, vol. 8, no. 6, pp. 661-679.

[152] Pierre G. Paulin and J.P. Knight. "Force-Directed Scheduling in Automatic Data Path Synthesis". *Proc. of the 24th Design Automation Conference*, pages 195–202, June 1987.

[153] O. Penalba, J.M. Mendias, and R. Hermida. "Maximizing conditional reuse by pre-synthesis transformations". *Design Automation and Test in Europe*, 2002.

[154] C. Pilato, V.G. Castellana, S. Lovergine, and F. Ferrandi. A runtime adaptive controller for supporting hardware components with variable latency. In *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pages 153–160, June 2011.

[155] M. Pinedo. *"Scheduling Theory, Algorithms and Systems"*. Prentice Hall, 1995.

[156] Shlomit S. Pinter. Register allocation with instruction scheduling. *SIGPLAN Not.*, 28:248–257, June 1993.

[157] J.M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *Design Test of Computers, IEEE*, 8(2):40–51, 1991.

[158] I. Radivojevic and F. Brewer. "A New Symbolic Technique for Con- trol-Dependent Scheduling". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 1, January 1996.

[159] V. Raghunathan, S. Ravi, and G. Lakshminarayana. Integrating variable-latency components into high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(10):1105–1117, 2000.

[160] M. Rim, Y. Fann, and R. Jain. "Global Scheduling with Code-Motions for High-Level Synthesis Applications". *IEEE Transactions on VLSI Systems*, vol. 3, no. 3, pp. 379-392, September 1995.

[161] C. Rust, A. Rettberg, and K. Gossens. From high-level petri nets to systemc. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, volume 2, pages 1032–1038 vol.2, 2003.

[162] R.W. Floyd and J.D. Ullman. "The Compilation of Regular Expressions into Integrated Circuits". *Jo. ACM*, 1982.

[163] Soujanna Sarkar, Shashank Dabral, Praveen K. Tiwari, and Raj S. Mitra. "Lessons and Experiences with High-Level Synthesis". *IEEE Design & Test*, Vol. 26 , Issue 4, pp 34-45, July 2009.

[164] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the low-power m core architecture. In *Power Driven Microarchitecture Workshop*, pages 145–150. Citeseer, 1998.

[165] S.Devadas and R. Newton. "Decomposition and Factorization of Sequential Finite State Machines". *IEEE Trans. on CAD*, 8:1206–1217, 1988.

[166] Richard Sharp and Alan Mycroft. Soft scheduling for hardware. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 57–72, London, UK, 2001. Springer-Verlag.

[167] Michael Sipser. *"Introduction to the Theory of Computation"*. PWS Publishing Co., Boston 1997.

[168] V. Sklyarov and I. Skliarova. Design and implementation of parallel hierarchical finite state machines. In *Communications and Electronics, 2008. ICCE 2008. Second International Conference on*, pages 33 –38, june 2008.

[169] D.L. Springer and D.E. Thomas. Exploiting the special structure of conflict and compatibility graphs in high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(7):843 –856, jul 1994.

[170] M.B. Srivastava and M. Potkonjak. Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(1):2 –19, march 1995.

[171] Leon Stok. Data path synthesis. *Integr. VLSI J.*, 18(1):1–71, dec 1994.

[172] Taylor L. Booth. *"Sequential Machines and Automata Theory"*. John Wiley and Sons, 1967.

[173] Taylor L. Booth. *"Digital Networks and Computer Systems"*. John Wiley and Sons, 1971.

[174] A.H. Timmer and J.A.G. Jess. "Exact Scheduling Strategies based on Bipartite Graph Matching". *Proceedings of the European Design & Test Conference*, pp. 42-47, 1995.

[175] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, (11):25–33, 1967.

[176] J.L. Tripp, M.B. Gokhale, and K.D. Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007.

[177] J.L. Tripp, K.D. Peterson, C. Ahrens, J.D. Poznanovic, and M.B. Gokhale. Trident: an fpga compiler framework for floating-point algorithms. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 317–322, 2005.

[178] Chia-Jeng Tseng and D.P. Siewiorek. Automated Synthesis of Data Paths in Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(3):379–395, 1986.

[179] C.J. Tseng and D.P. Siewiorek. "Automated synthesis of data paths in digital systems". *Proceedings of the 20th Design Automation Conference*, July 1986.

[180] Frank Vahid. What is hardware/software partitioning? *SIGDA Newsl.*, 39(6):1–1, jun 2009.

[181] Diederik Verkest, Joachim Kunkel, and Frank Schirrmeister. "System Level Design Using C++". *Proc. of Design, automation & Test in Europe*, Paris, France, pp. 74-83, 2000.

[182] J. Villarreal, A. Park, R. Atadero, W. A. Najjar, and G. Edwards. Programming the convey HC-1 with ROCCC 2.0. In *CARL 2010: The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

[183] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, 2010.

[184] K. Wakabayashi and T. Yoshimura. "A resource sharing and control synthesis method for conditional branches". *IEEE International Conference on Computer-Aided Design*, 1989.

[185] Kazutoshi Wakabayashi. Cyber: High level synthesis system from software into asic. In Raul Camposano and Wayne Wolf, editors, *High-Level VLSI Synthesis*, volume 136 of *The Springer International Series in Engineering and Computer Science*, pages 127–151. Springer US, 1991.

[186] Kazutoshi Wakabayashi. C-based behavioral synthesis and verification analysis on industrial design examples. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 344–348, Piscataway, NJ, USA, 2004. IEEE Press.

[187] David W. Wall. Limits of Instruction-Level Parallelism. In *ASPLOS IV: the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, 1991.

[188] Weidong Wang, A. Raghunathan, N.K. Jha, and S. Dey. High-Level Synthesis of Multi-Process Behavioral Descriptions. In *16th International Conference on VLSI Design*, pages 467–473, 2003.

[189] Tomonori Yamashita, Kazuhiro Matsuzaki, Takeshi Toyoyama, Masaoki Satoh, Akifumi Adachi, and Fusako Sugawara. "Using high-level synthesis for FPGA development. Applying a C-based design methodology using Catapult C Synthesis at FUJITSU". Technical report, Design Platform Development Center Corporate Product Technology Unit, FUJITSU LIMITED.

[190] F.F. Yassa, J.R. Jasica, R.I. Hartley, and S.E. Noujaim. A silicon compiler for digital signal processing: Methodology, implementation, and applications. *Proceedings of the IEEE*, 75(9):1272–1282, 1987.

[191] Katherine A. Yelick. Programming models for irregular applications. *SIGPLAN Not.*, 28:28–31, January 1993.

Bibliography

[192] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A Platform-Based ESL Synthesis System. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008.