Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2012, Article ID 156907, 30 pages
doi:10.1155/2012/156907

Review Article

Formal ESL Synthesis for Control-Intensive Applications

Michael F. Dossis

Department of Informatics and Computer Technology, School of Kastoria, Higher Technological Education Institute of Western Macedonia, Fourka Area, 52100 Kastoria, Greece

Correspondence should be addressed to Michael F. Dossis, mdossis@yahoo.gr

Received 6 February 2012; Accepted 14 April 2012

Academic Editor: Kamel Barkaoui

Copyright © 2012 Michael F. Dossis. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Due to the massive complexity of contemporary embedded applications and integrated systems, long effort has been invested in high-level synthesis (HLS) and electronic system level (ESL) methodologies to automatically produce correct implementations from high-level, abstract, and executable specifications written in program code. If the HLS transformations that are applied on the source code are formal, then the generated implementation is correct-by-construction. The focus in this work is on application-specific design, which can deliver optimal, and customized implementations, as opposed to platform or IP-based design, which is bound by the limits and constraints of the preexisting architecture. This work surveys and reviews past and current research in the area of ESL and HLS. Then, a prototype HLS compiler tool that has been developed by the author is presented, which utilizes compiler-generators and logic programming to turn the synthesis into a formal process. The scheduler PARCS and the formal compilation of the system are tested with a number of benchmarks and real-world applications. This demonstrates the usability and applicability of the presented method.

1. Introduction

During the last 3-4 decades, the advances on chip integration capability have increased the complexity of embedded and other custom VLSI systems to such a level that sometimes their spec-to-product development time exceeds even their product lifetime in the market. Because of this, and in combination with the high design cost and development effort required for the delivery of such products, they often even miss their market window. This problem generates competitive disadvantages for the relevant industries that design and develop these complex computing products. The current practice in the used design and engineering flows, for the development of such systems and applications, includes to a large extent approaches which are semimanual, ad-hoc, nonautomatically communicants from one level of the design flow to the next, and with a lot of design iterations caused by the discovery of functional and timing bugs, as well as specification to product requirements mismatches later in the development process. All of these issues have motivated industry and academia to invest in suitable methodologies and tools to achieve higher automation in the design of contemporary systems. Nowadays, a higher level of code abstraction is pursued as input to automated E-CAD tools. Furthermore, methodologies and tools such as high-level synthesis (HLS) and electronic system level (ESL) design entries employ established techniques, which are borrowed from the computer language program compilers and mature E-CAD tools and new algorithms such as advanced scheduling, loop unrolling, and code motion heuristics.

Even nowadays, the practiced flow for complex heterogeneous (hardware + software) systems and applications is still to a large extent an empirical process. Usually, engineers and engineering managers with different technical area skills are brought together at the same group, or even from a large number of different organizations of a consortium, and they are allocated engineering tasks which are scheduled and partitioned manually and by the most experienced engineers of the team. Even during the design process, the engineers of the team exchange information between them in a nonformal and ad-hoc way. Quite often, the system designers, the engineering managers and the other engineers of the team use a mix of manual methods and tools that are fundamentally incompatible with each other. Thus, the

design data need to be manually modified in order to be ported from one tool or engineering level of the design flow, to another. This of course prolongs the spec-to-product cycle and drastically increases the engineering effort which is required for the completion of the product, mainly due to the very fragmented design and development flow which is still in practice in industry. Therefore, academia and industry, for a long time, have been investigating formal and automatic transformation techniques to automatically convert design representations from a higher abstraction level to a lower level in the design process. The main contribution of this paper is formal and automatic hardware compiler system to deliver correct hardware implementations from high level, algorithmic, and directly executable program code system specifications.

The conventional approach in designing complex digital systems is the use of register-transfer level (RTL) coding in languages such as VHDL and Verilog. However, for designs that exceed an area of a hundred thousand logic gates, the use of RTL models for specification and design can result into years of design flow loops and verification simulations. Combined with the short lifetime of electronic products in the marker, this constitutes a great problem for the industry. Therefore, higher abstraction level and executable types of specifications are required to make the industry competitive. HLS started appearing as an attractive solution in the 1980s where simple HLS tools were mapping mostly linear (dataflow-oriented) applications into hardware netlists. However, the broad acceptance of HLS by the engineering community was prevented for a long time from the poor synthesis results from specifications that with hierarchy and complex (e.g., nested) control flow constructs in the specification program. The programming style of the specification code has an unavoidable impact on the quality of the synthesized system. This is deteriorated by models with hierarchical blocks, subprogram calls, as well as nested control constructs (e.g., if-then-else and while loops). The result of these models is that the complexity of the transformations which are required for the synthesis tasks (compilation, algorithmic transformations, scheduling, allocation, and binding) increases at an exponential rate, whereas the design size increases linearly.

Usually the input code (such as ANSI-C or ADA) to HLS tool is first transformed into a control/data flow graph (CDFG) by a front-end compilation stage. This involves a number of compiler-like optimizations such as code motion, dead code elimination, constant propagation, common subexpression elimination, loop unrolling hardware-oriented optimizations such as minimization of syntactic variances, retiming, and code transformations which are based on the associativity and commutativity properties of some operators, in order to deliver simpler expressions. Then, various synthesis transformations are applied on the CDFG to generate the final implementation. The most important HLS tasks of this process are scheduling, allocation, and binding. Scheduling makes an as-much-as-possible optimal order of the operations in a number of control steps or states. Optimization at this stage includes making as many operations as possible parallel, so as to achieve shorter execution times of the generated implementation. Allocation and binding assign operations onto functional units, and variables and data structures onto registers, wires, or memory positions, which are available from an implementation library.

A number of commercial HLS tools exist nowadays, which often impose their own extensions or restrictions on the programming language code that they accept as input, as well as various shortcuts and heuristics on the HLS tasks that they execute. Such tools are the CatapultC by Mentor Graphics, the Cynthesizer by Forte Design Systems, the Impulse CoDeveloper by Impulse Accelerated Technologies, the Synfony HLS by Synopsys, the C-to-silicon by Cadence, the C to Verilog Compiler by C-to-Verilog, the AutoPilot by AutoESL, the PICO by Synfora, and the CyberWorkBench by NEC System Technologies Ltd. The analysis of these tools is not the purpose of this work, but most of them are suitable for linear, dataflow-dominated (e.g., stream-based) applications, such as pipelined DSP and image filtering.

An important aspect of the HLS tools is whether their transformation tasks (e.g., within the scheduler) are based on formal techniques. The latter would guarantee that the produced hardware implementations are correct-by-construction. This means that by definition of the formal process, the functionality of the implementation matches the functionality of the behavioral specification model (the source code). In this way, the design will need to be verified only at the behavioral level, without spending hours or days (or even weeks for complex designs) of simulations of the generated register-transfer level (RTL), or even worse of the netlists generated by a subsequent RTL synthesis of the implementations. The behavioral code can be verified by building a module that produces test vectors and reads the results and this verification can be realized with simple compilation and execution with the host compiler of the language (e.g., GNU C compiler and linker). This type of behavioral verification is orders of magnitude faster than RTL or even more than gate-netlist simulations.

Moreover, the hardware/software codesign approach, which is followed by the author's work, allows to model the whole embedded (or other) digital system in ADA (currently a C front-end is being developed as well), and coverified at this level using standard compile and execute techniques with the host ADA compiler. This also enables the building of the system under test as well as the testbench code to be developed in the same format, which enforces functional verification and debugging at the earliest steps of the product design and development flow. In this way, and by using standard and formal techniques, late reiterations in the development flow are avoided and thus valuable project time is saved, so as to focus to more important, system level design decisions such as the target architecture and tradeoffs between hardware and software implementations of the various system parts. For the system compilation, a formal IKBS methodology is used, by combining compiler-compiler and logic programming techniques, borrowed from areas such as formal compilation and artificial intelligence.

The codesign techniques of the author's work produce superior hardware module performance and their results are

more adaptable to different host architectures, as compared with traditional-platform-based and IP-based approaches. This is due to the fact that platform-based design makes a lot of system and interface assumptions about the target architecture which are often found out that they are not true, and, therefore, the delivered parts are not compatible and cannot be plugged into the target system. Even in the best case, the codesign results are found to be suboptimal due to mismatching performances of the core and the interface functionality of the delivered modules. The same apply for the IP-based design, plus the time spent to understand the IP's function and interfaces, and build proper test procedures for IP verification flows when the given IP is plugged in the target architecture.

This paper presents the formal IKBS methodology, as well as the usability and benefits of it in the prototype hardware compilation system. Section 2 discusses related work. After a review of existing intermediate formats, the author's intermediate predicate format is analyzed in Section 3. The hardware compilation design flow and in particular the loading of the IPF database in the IKBS engine are explained in Section 4. Section 5 summarizes the inference logic rules of the IKBS engine of the back-end phase of the prototype behavioral synthesizer. In Section 6, the mechanism of the formal high-level synthesis transformations is presented. Section 7 outlines the structure and logic of the PARCS optimizing scheduler which is part of the back-end compiler rules. Section 8 explains the available options for target microarchitecture generation and the communication of the accelerators with their computing environment. Section 9 outlines the general execution environment for the results of the hardware/software codesign methodology of this paper. Sections 10 and 11 discuss experimental results, draw useful conclusions, and propose future work.

2. Background and Review of ESL Methodologies

2.1. The Scheduling Task. The scheduling problem covers two major categories: time-constrained scheduling and resource-constrained scheduling. Time-constrained scheduling attempts to result into the lowest hardware cost (e.g., area or number of functional units) when the total number of control steps (states) is given (time constraint). Resource-constrained scheduling attempts to produce the fastest schedule (the fewest control states) when the amount of hardware resources or hardware area are given (resource constraint). Integer linear programming (ILP) formulations for the scheduling problem have been proposed. However, their execution time grows exponentially with the increase of the number of variables and inequalities. Therefore, ILP is generally impractical and it is suitable only for very small designs. Heuristic methods have been introduced to deal with large designs and to provide suboptimal but practical implementations. Heuristic scheduling uses in general two techniques: constructive solutions and iterative refinement. Two constructive methods are the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) approach. Both of these methods place the hardware operations in a precedence-based list. With the ASAP method, one operation is taken from the list at a time and the algorithm tries to position the operation at the earliest possible control step. With the ALAP method, each operation from the list is moved at the latest possible control step. The operations that were placed in the same control steps of by both ASAP and ALAP methods constitute the design's critical path.

In both ASAP and ALAP scheduling, the operations that belong to the critical path of the design are not given any special priority over other operations. Thus, and particularly when the resource constraints are too hard and so only a few operations can be assigned on similar functional units in each control cycle, excessive delay may be imposed on the critical path operations. This is not good for the quality of the produced implementation. On the contrary, list scheduling utilizes a global priority function to select the next operation to be scheduled. This global priority function can be either the mobility [1] of the operation or its urgency [2]. The mobility of an operation is the difference between its ASAP and ALAP control step in absolute terms. Force-directed scheduling [3] calculates the range of control steps for each operation between the operation's ASAP and ALAP state assignment. It then attempts to reduce the total number of functional units of the design's implementation, in order to evenly distribute the operations of the same type into all of the available states of the range. This is done by using distribution graphs of operations, which are assigned to the same state.

The problem with constructive scheduling is that there is not any lookahead into future assignment of operations into the same control step. In this way, the generated design implementation may be suboptimal. In contrast to this, the iterative scheduling produces new schedules, by iteratively rescheduling sequences of operations that maximally reduce the cost functions [4], after an initial schedule is delivered by any of the above scheduling algorithms. When no further improvement can be achieved, the scheduling execution stops. The above scheduling techniques are usually applied on linear dataflow dominated sequences of operations. In order to schedule and place control-intensive designs which include constructs such as loops, loop pipelining [5], and loop folding [6], different techniques have been reported in the bibliography.

2.2. Allocation and Binding Tasks. Allocation determines the type of resource storage and functional units, selected from the library of components, for each data object and operation of the input program. Allocation also calculates the number of resources of each type that are needed to implement every operation or data variable. Binding assigns operations, data variables, data structures, and data transfers onto functional units, storage elements (registers or memory blocks) and interconnections, respectively. Also binding makes sure that the design's functionality does not change by using the selected library components. The three interdependent tasks of binding are functional-unit binding, storage-element binding, and interconnection binding. Functional-unit binding assigns operations onto functional units and

operators (e.g., adders, subtractors, multipliers, ALUs) from the available resource library. Storage binding maps data objects such as variables, constants, and data structures (e.g., arrays, or records) onto hardware elements such as registers, wires (connected on the power or ground lines) and RAMs/ROMs, respectively. Interconnection binding maps data transfers onto sets of interconnection units, along with the necessary multiplexing to implement the required data routing in the delivered hardware implementation. All of these are placed on the datapath implementation of the design after HLS runs on it.

Generally, there are three kinds of solutions to the allocation problem: constructive techniques, decomposition techniques, and iterative approaches. Constructive allocation techniques start with an empty implementation and progressively build the datapath and control parts of the implementation by adding more functional, storage, and interconnection elements while they traverse the CDFG (control-data-flow-graph) or any other type of internal graph/representation formats. Constructive techniques are fairly simple but the implementations they produce are far from optimal. Decomposition techniques divide the allocation problem into a sequence of well-defined independent subtasks. Each such subtask is a graph-based theoretical problem which is solved with well-known graph methods. Three such graph-based methods are the clique partitioning, the left-edge technique, and the weighted bipartite matching technique. The three allocation subtasks of functional unit, storage, and interconnection allocation are mapped onto the problem of graph clique partitioning [7]. The nodes of the graph are operations, values, and interconnection elements. The task of finding the minimum cliques in the graph which is the solution for the subtasks, is an NP-hard problem, therefore, heuristic approaches [7] are utilized for allocation.

Because the conventional subtask of storage allocation ignores the side-effects between the storage and interconnections allocation, when using the clique partitioning technique, graph edges are enhanced with weights that represent the effect on interconnection complexity. This complexity is caused by sharing registers among different variables of the design [3]. The left-edge algorithm is applied on the storage allocation problem, and it allocates the minimum number of registers [8]. The left-edge algorithm has a polynomial complexity against the clique partitioning which is NP-complete. Nevertheless, the left-edge algorithm does not take into account the interdependence with the interconnect cost. This is considered in the weighted graph edges of the clique partitioning solution.

A weighted, bipartite-matching algorithm can be used to solve both the storage and functional unit allocation problems. First, a bipartite graph is generated which contains two disjoint sets, for example, one for variables and one for registers, or one for operations and one for functional units [9]. An edge between one node of the one of the sets and one node of the other represents an allocation of, for example, a variable to a register. The bipartite-matching algorithm has a polynomial complexity and it allocates the minimum number of registers. Moreover, this algorithm considers the effect of register allocation on the design's interconnection elements because the edges of the two sets of the graph are weighted [9]. The datapaths that are generated by either constructive or decomposition allocation techniques can be further improved iteratively. This is done either by a simple assignment exchange, using the pairwise exchange of the simulated annealing, or by using a branch-and-bound approach. The latter reallocates groups of elements of different types in order to refine the datapath implementation [10].

2.3. Early High-Level Synthesis. HLS has been an active research field for more than two decades now. Early approaches of experimental synthesis tools that synthesized small subsets of programming constructs or proprietary modeling formats have emerged since the late 1980s. As an example, an early tool that generated hardware structures from algorithmic code, written in the PASCAL-like, digital system specification language (DSL) is reported in [11]. In this work, the three domains of integrated circuit designs were defined as the behavioral, structural, and geometrical. Behavioral synthesis is the transformation of behavioral descriptions (e.g., program code) into circuit structures. This can be done at different levels, for example, at the register-transfer, at the gate or logic level, at the transistor or electric level, and so forth. The geometrical domain involves the generation of the integrated circuit geometrical features such as the circuit layout on the silicon array. What the authors in [11] named as behavioral synthesis is of course defined in our days in much more detail, it is extended at the abstraction level of the specifications and it comes under the name high-level synthesis (HLS). The problem of HLS is extremely complex, but it is much more understood in our days than in the early days of the first synthesis systems. The main tasks that were identified in [11] were compilation, datapath, and control synthesis from imperative specifications (e.g., in DSL), optimization (area and speed), and circuit assembly. The circuit structure generated in [11] is coded in the structure description language (STRUDEL), and this in turn in ported to the Carlsruhe Digital System CADDY to generate the geometrical description of the circuit. Examples of other behavioral circuit specification languages of that time, apart from DSL, were DAISY [12], ISPS [13], and MIMOLA [14]. The synthesis system in [11] performs the circuit compilation in two steps: first step is datapath synthesis which is followed by control synthesis. The authors claimed the lack of need for verification of the generated circuit after synthesis, since it is correct by construction due to automated circuit synthesis methods.

The PARSIFAL DSP synthesis system from GE Corporate and Development division was one of the earliest synthesizers that targeted DSP applications [15]. The designed circuit is described with a combination of algorithmic and structural level and using PARSIFAL it is synthesized in a bit-serial DSP circuit implementation. PARSIFAL is part of a larger E-CAD system called FACE and which included the FACE design representation and design manager core. FACE includes interfaces to synthesis tools, analysis tools, physical assembly tools, libraries, and external tools such as parsers

and formatters. The synthesis subsystem of FACE is interactive and it utilizes the FACE core functions. FACE focuses on design transformations and optimizations which are suitable for pipelined and nonpipelined architectures. The FACE synthesis process includes the following tasks: minimize the execution time of expressions, maximize hardware resource sharing, insert multiplexers, and schedule operations in pipeline stages. It is thus obvious that FACE and PARSIFAL were suitable for DSP-pipelined implementations, rather than they constitute a more general hardware synthesis system.

The synthesis optimization tasks such as scheduling of operations and allocation of registers and busses, considering timing and hardware resource-constraints, are analyzed in [16]. According to [16], scheduling consists of determining the propagation delay of each operation and then assigning all operations into control steps (states) of a finite-state machine. Different types of scheduling approaches are explained. Algorithms such as list-scheduling, attempt to minimize the total execution time of the state machine while obeying to resource constraints. List-scheduling uses a local priority function to postpone the assignment of operations into states, when resource constraints are violated. On the contrary, force-directed scheduling (FDS) tries to satisfy a global execution deadline (time constraint) while minimizing the utilized hardware resources (functional units, registers and busses). The way FDS does this is by positioning similar operations in different control states, so that the concurrency of operations is balanced without increasing the total execution time of the circuit. In this way, each structural unit retains a high utilization which results into reducing the total number of units that are required to implement the design. This is achieved in three basic steps: determine the time-frame of each operation, generate a distribution graph, and calculate the force associated with each operation assignment. In [16], minimizing the cost of storage units and interconnections is addressed as well. The force-directed list scheduling (FDLS) algorithm attempts to implement the fastest schedule while satisfying fixed hardware resource constraints. FDLS is similar to the list scheduling approach. However, in FDLS, the force is the priority function rather than the mobility of urgency of operations. Another implementation exploration approach is also outlined in [16] by combining FDS and FDLS. In this approach, first the FDS method is applied to find the near-optimal allocation by satisfying a fixed maximum time constraint. Then, the designer runs FDLS on the results of FDS in order to try and find an even faster implementation. After scheduling, the following problems are addressed: bind operations to functional units, bind storage actions to registers, and bind data-transfer operations to interconnections. Also merging registers, merging multiplexers, and good design partitioning are analyzed in [16]. However, there are no indications as to how fast the synthesis algorithms run using the publication's techniques.

The authors in [17] defined as the main problem in HLS the mapping of a behavioral description into a register-transfer level (RTL) circuit description which contains a datapath and a control unit. In our days, the latter can be implemented with a finite-state machine (RTL) which controls a datapath of operators, storage elements and a number of data-steering multiplexers. According to [17], the main tasks in HLS include allocation, scheduling, and binding. According to [18], scheduling is finding the sequence of which operations to execute in a specific order so as to produce a schedule of control steps with allocated operations in each step of the schedule; allocation is defining the required number of functional, storage, and interconnect units; binding is assigning operations to functional units, variables, and values to storage elements and forming the interconnections amongst them to form a complete working circuit that executes the functionality of the source behavioral model. First, the input behavioral description is transformed into a control/data-flow graph (CDFG). Then, various optimization algorithms run on this CDFG in order to drive the implementation of the final circuit implementation. The CDFG captures the algorithmic characteristics of the input behavioral program (e.g., in VHDL or Verilog) as well as the data and control dependency between the operations to be scheduled. Apparently, two operations that have a read-after-write dependency from one another, they cannot be scheduled in the same control step (or state). The authors in [18] introduce various problems that are encountered within various scheduling approaches: the unconstrained scheduling (UCS) problem, the time-constrained scheduling (TCS) problem, the resource-constrained scheduling (RCS) problem and mixed approaches such as the time- and resource-constrained scheduling (TRCS) problem. Also, advanced synthesis issues such as chaining (concatenating a different operations within the same control step in a chain) and multicycling (spreading the execution of an operation over more than one control step), handling in a special way control structures such as nested if-then-else and loop constructs, and various issues of constraining the global execution time and the latency of operations by the user of the synthesis tool, are addressed in [18]. Moreover, [18] defines and analyses the most common scheduling algorithms and approaches which include as-soon-as-possible (ASAP) scheduling, as-late-as-possible (ALAP) scheduling, list scheduling, force-directed scheduling, and integer linear programming (ILP).

The V compiler [19] translates sequential descriptions into RTL models using parsing, scheduling, and resource allocation. The source sequential descriptions are written in the V language which includes queues, asynchronous calls, and cycle blocks and it is tuned to a kind of parallel hardware RTL implementations. The utilized parser is built from a LALR grammar, and the parse tree includes leaves representing syntactic tokens and vertices (nodes) representing syntactic units of the source code. The V compiler marks the statements in the generated RTL and simulation code so that the user can trace the statements back in the V code by using and observing the token number. It also treats the hardware state machine as a directed, (possibly) cyclic control graph. Each vertex of the graph represents a state of the state machine and a set of operators to execute on the particular cycle. Each edge between vertices represents a state transition which can be guarded by a Boolean variable. Thus, if a vertex has multiple transitions to other vertices, then the conditions

of these multiple edges must be mutually exclusive. The inputs and outputs of operations and the conditions on the state transitions and operators are initially treated all as variables. Later and during RTL implementation, these variables are implemented with wires or with registers. The V compiler utilizes percolation scheduling [20] to "compress" the state machine in time, and achieve the required degree of parallelism by meeting time constraints. Apart from the RTL models, the compiler generates also simulation code in PL/I so to simulate and verify the generated hardware implementation models.

A timing network is generated once from every different behavioral design in [21] and is annotated with parameters for every different scheduling approach. The produced timing network is based solely on the control and data graphs that are derived from the input specification, before scheduling and allocation. The scheduling optimization approach in [21] attempts to satisfy a given design cycle for a given set of resource constraints, using the timing model parameters. An integrated approach for scheduling, allocation, and binding in datapath synthesis is explained in [22]. Using highly generalized modules, this approach uses an integer linear program (ILP) which minimizes a weighted sum of area and execution time of the implementation. We can say that this implements a mixed time and resource-constrained scheduling. The above modules can execute an arbitrary number of different operations, using, for example, different numbers of control steps for different operations. Moreover, the same operation can be executed on a variety of modules possibly involving different number of control steps. The synthesis approach in [22] attempts to minimize the execution time and the hardware area of an initial data-flow graph (DFG) by using two types of constraints: the data dependency constraints (DD-constraints) and an operation ordering based on the sharing of functional units by the operations (UU-constraints). Also, this work [22] includes extensions of the ILP approach for pipelined functional units and for operation chaining. A prototype synthesizer called Symphony [22], in combination with three benchmarks that were executed through the Symphony system, namely, a fifth-order elliptical wave filter, a differential equation, and a bandpass filter. For these benchmarks and according to the authors of [22], the Symphony tool delivers better area and speed than ADPS [23]. It seems from the type of scheduling approach as well as the presented tests that the approach in [22] is rather suitable for data-flow-dominated designs such as DSP blocks and not for more general complex control flow designs.

The CALLAS synthesis framework [24] transforms algorithmic, behavioral VHDL models into VHDL RTL and gate netlists, under timing constraints. These netlists are then implemented into available technologies using available commercial logic synthesis tools. If the timing constraints are too tight for the scheduler, then CALLAS produces an ASAP schedule and issues a relevant error message. The EXPANDER tool is connected to the back-end of CALLAS in order to support low-level synthesis of the produced implementation using specific delay, area, and library components. CALLAS produces the final implementation via a number of iterative high-level and RTL transformations upon an initial structure which is found in the algorithmic VHDL source code. The user of CALLAS can drive these transformations by using a synthesis script. Compilation of the algorithmic code (from a subset of the VHDL language) delivers initial data flow and control flow graphs, and an initial processing generates a starting ALAP schedule without resource constraints. Afterwards, the control flow graph is reduced so that the fixed I/O operation schedule is satisfied. The initial structure is optimized by a number of high-level and RTL refining transformations. Then, the produced structure is further going through a logic optimization and technology mapping by the EXPANDER tools, and thus a VHDL or EDIF [25, 26] netlist is generated. The generated circuit is implemented using a Moore-type finite-state machine (FSM), which is consistent with the semantics of the VHDL subset used for the specification code. The synthesis transformations in CALLAS include removal of superfluous edges in the control flow graph, removal of unnecessary data transfers between registers, and control flow graph reduction (scheduling) so as to meet the specified I/O-timing constraints. Other optimizations include lifetime analysis, register sharing, operator sharing, multiplexor optimization, arithmetic and logic transformations, optimizing of the datapath/controller interface, flattening of complex functional units, partitioning, and logic minimization. These optimizations utilize techniques such as clique partitioning, path analysis, and symbolic simulation. Formal verification techniques such as equivalence checking, which checks the equivalence between the original VHDL FSM and the synthesized FSM are used in the CALLAS framework by using the symbolic verifier of the circuit verification environment (CVE) system [27]. A number of benchmarks and industrial designs were executed within the CALLAS framework and confirmed its usability.

The Prolemy framework [28] allows for an integrated hardware-software codesign methodology from the specification through to synthesis of hardware and software components, simulation, and evaluation of the implementation. Prolemy is a tool-set that allows the simulation, and rapid prototyping of heterogeneous hardware + software systems. The block is the basic unit of modularity inside Prolemy. Blocks communicate with each other and with their computing environment through portholes. The tools of Prolemy can synthesize assembly code for a programmable DSP core (e.g., DSP processor), which is built for a synthesis-oriented application. A domain in Ptolemy consists of a set of blocks, targets, and associated schedulers that conform to the operational semantics. These semantics determine how blocks interact. Some of the simulation domains supported by Ptolemy include the synchronous dataflow (SDF), dynamic dataflow (DDF), and digital hardware modeling (Thor). For example, for every commercial DSP processor there are corresponding models and a simulator. This simulator is invoked when the user wants to verify a design that contains the corresponding processor. Mixed digital and analog components, for example, A/D and D/A converters and filters can be represented as components with their functional models in the SDF domain. The engineers of

Ptolemy have supported the generation of C and C++ code for a variety of processors. In Ptolemy, an initial model of the entire system is partitioned into the software and hardware parts which are synthesized in combination with their interface synthesis. Then, the hardware, software, and interface implementation models can be cosimulated and the overall system prototype can be evaluated. The unified representation of hardware and software components allows the migration of functions between the two implementations with their interfaces being automatically synthesized as well. This process is not fully automatic but the users of Ptolemy are benefited with interoperability of the tools.

The Cosyma framework [29] realizes an iterative partitioning process, based on hardware extraction algorithm which is driven by a cost function. The primary target in this work is to minimize customized hardware within microcontrollers but the same time to allow for space exploration of large designs. The Cosyma hardware-software cosynthesis targets a processor core, memory, and custom coprocessing engines. In the Cosyma flow, the implementation of the system focuses on the generation of machine code for the embedded microprocessor. Custom hardware replaces the equivalent software parts, only when timing constraints are violated by the generalized processor code, or when the completion of the embedded system requires basic, available, and cheap I/O peripherals. The specialized coprocessors of the embedded system can be synthesized using HLS tools. Hardware/software partitioning is automatic in the Cosyma flow. Initially, the whole system is implemented in a set of hardware components. Then, gradually as many as possible of these hardware components are transformed into software components, with the precondition that timing constraints and system synchronization are satisfied. The specification language is based on C and it contains the following extensions: timing (minimum and maximum delays), tasks, and task intercommunication. Partitioning occurs at different levels of system granularity: task, function, basic block, and single statement. Parallelism in the Cosyma C language is explicit and it is defined by the user (the programmer). The extended syntax (ES) graph is used as the internal representation of the design in Cosyma. ESG is extended by a symbol table as well as data and control dependencies. The ES graph is used for both partitioning and cost estimation as well as for software and hardware C generation. The hardware description is in turn ported to the HLS Olympus tool [30]. Cosyma utilizes its ES internal format to estimate possible speedups of the critical loops in the design and, therefore, aid towards the required software-hardware partitioning. Partitioning is based on a partitioning cost function to drive the hardware implementation of the system components that can be implemented well in hardware. Such a cost function includes knowledge about synthesis, compilers, and libraries. An example is a specific cost function for the extraction of coprocessors that implement computation-time-intensive parts of the application such as nested loops. The work in [29] included tests and experimental results based on a configuration of an embedded system, which is built around the Sparc microprocessor.

AMICAL is a VHDL-based behavioral synthesis tool of the early 1990s [31]. A number of constraints were imposed on the writing style of VHDL in order to use AMICAL for HLS compilation of hardware architectures. One strong application use of AMICAL was the synthesis of control-intensive communication protocols. In order to achieve this, AMICA utilizes control-flow graphs, dynamic loop scheduling, in order to represent and process constructs such as nested loops, unstructured control statements such as loop exits wait statements used for synchronization. Nevertheless, AMICAL is not strictly an HLS system with the meaning given to HLS in this paper, since instead of general purpose programming format, it accepts (and it is oriented to) descriptions in VHDL which requires the designer to think about hardware-specific features when modeling of the system is realized.

The work in [32] discusses a methodology for cosimulation and cosynthesis of mixed hardware-software specifications. During cosynthesis, hardware-software partitioning is executed in combination with control parallelism transformations. The hardware-software partition is defined by a set of application-level functions which are implemented with application-specific hardware. The control parallelism is defined by the interaction of the processes of the functional behavior of the specified system. Finding the appropriate control concurrency involves splitting of merging processes or moving functionality from one process to another. The cosimulation environment produces a mixed system model that is functionally correct but it may not meet design goals. The cosynthesis tools are then used to modify the hardware-software partition and the control concurrency so that design goals are satisfied. Afterwards, the software part is implemented with standard compilation into system memory and the hardware part is synthesized with HLS tools and implemented with reconfigurable Xilinx FPGAs and two field-programmable interconnect chips from Aptix. All of these modules are plugged in the backplane of the host computer, so that implementation measurements can be realized. There are three abstractions of hardware-software interactions: send/receive/wait transactions between application program and custom hardware, register read/write between the I/O driver running in the host computer and the bus interface of the custom hardware, and bus transactions between the two I/O bus sides. The system behavior is modeled using a set of communicating sequential processes [33]. Each process can be assigned either to hardware or to software implementation. The following types of interprocess communication primitives exist in the system: synchronized data transfer, unsynchronized (unbuffered) data transfer, synchronization without data transfer, and communication with a shared memory space. Cosimulation is implemented in [32] using a Verilog simulator and a Verilog PLI interface. Two example applications were used to evaluate the cosynthesis and cosimulation environment: the Sphinx speech phoneme recognition system and a data compression/encryption application.

Yet another hardware-software codesign methodology is presented in [34], which employs synthesis of heterogeneous systems. The synthesis process is driven by timing constraints

which drive the mapping of tasks onto hardware or software parts so that the performance requirements of the intended system are met. This method is based on using modeling and synthesis of programs written in the HardwareC language. This enables the use of the Olympus chip synthesis system for prototyping of the designed application [30]. A set of interacting processes which are instantiated in design blocks using declarative semantics are included in the HardwareC model. When all tasks are completed, the hosting process restarts itself. All of the processes can execute concurrently in the system model. Hierarchically related sequencing graphs are produced from the input HardwareC specification. Within each graph, vertices represent input program operations and edges represent dependencies between operations. Two vertices, namely, the source (beginning) and sink (end) represent no operations. Operations in different graphs can pass messages to each other in the graph model, in the same manner as send and receive. This is a very important feature in modeling of heterogeneous systems because the processor (which implements the software part of specification) and the custom hardware (which implements the hardware part of specification) may run on different clocks and speeds. Timing constraints are used to select the specific system implementation so as to satisfy specific performance requirements. Timing constraints are of two types: min/max delay constraints and execution rate constraints. For example, minimum delay constraints are captured by providing weights at the edges of the graph, to indicate delay to the corresponding source operations of each edge. Performance measurement is done on the basis of operation delays. These delays are estimated separately for the hardware and software parts of the system based on the type of hardware technology which is used to implement the hardware part of the system, and the processor which is used to run the software. The assignment of an operation to software or hardware implementations affects the delay of the operation. Moreover, moving operations from the hardware to software parts and vise versa involve additional delays due to emerging intercommunication delays. All these delays are used to determine the hardware/software partitioning of the final system implementation. An example application which was used to test the methodology in [34] was an Ethernet-based network coprocessor. The authors concluded that the use of their proposed hardware-software codesign methodology can aid significantly the design and development of embedded real-time systems which have a simple configuration as compared to that of a general purpose computing system.

2.4. Next-Generation High-Level Synthesis Tools. More advanced methodologies and tools started appearing from the late 1990s and continue with more improved input programming code sets as well as scheduling and other optimization algorithms. Furthermore, system level synthesis matured in the last decade by using more (application-wise) specialized and platform-oriented methodologies. The CoWare hardware-software codesign environment [35] is based on a data model that allows the user to specify, simulate, and produce heterogeneous implementations from heterogeneous specification source models. The choice of implementing real-time telecommunications DSP transformations on programmable DSP processors or application-specific hardware is driven by tradeoffs between cost, power, performance, and flexibility. The synthesis approach in [35] focuses on designing telecommunication systems that contain DSP, control loops, and user interfaces. The synchronous dataflow (SDF) type of algorithms, found in a category of DSP applications, can easily be synthesized into hardware from languages such as SILAGE [36], DFL [37], and LUSTRE [38]. The advantage of this type of designs is that they can be scheduled at compile time and the execution of the compiled code can be two orders of magnitude faster than event-driven VHDL (e.g., RTL) simulations. In contrast to this, dynamic dataflow (DDF) algorithms consume and produce tokens that are data dependent, and thus they allow for complex if-then-else and while loop control constructs. One way to deal with the data-dependent DDF algorithms is to map them onto the worst case SDF and schedule them at compile time. Another way is to partition the DDF into partial SDFs that are triggered by internal or external Boolean conditions. Then, these partial SDFs need to be scheduled at run time using the I/O timing constraints of the DSP signals and other external events. CAD systems that allow for specifying both SDF and DDF algorithms and perform as much as possible static scheduling are the DSP station from Mentor Graphics [39], PTOLEMY [40], GRAPE-II [41], COSSAP from Synopsys, and SPW from the Alta group [42]. Processes are used to realize modularity in the specification models of the CoWare tool [35]. A behavioral interface with read/write ports implements the communication between processes. Process ports that communicate with each other are connected through a channel. The data model is hierarchical and allows for gradual refinement of channels, ports, and protocols into lower levels of objects, by continuously adding detail. The most abstract object is the primitive object. In contrast, a hierarchical object contains implementation detail. A thread is a single flow of control within a process. There are slave threads and autonomous threads [35]. Communication between threads in different processes is called interprocess communication. Shared variables or signals, that are declared within the context of a process, are used for intraprocess communication. Channels and ports can be refined via adding more detail onto them, through the CoWare's design flow. The CoWare data model is suitable for merging of processes and for design for reuse and reuse of designs. Software/hardware communication is implemented in CoWare by means of memory-mapped I/O, instruction-programmed I/O, and interrupt control modules. The CoWare methodology was evaluated in [35] using a design example, which is a pager, based on spread-spectrum techniques. One important conclusion in [35] was that there is a pressing need for bottom-up formal verification tools, which can evaluate both functionality and timing of the design before and after synthesis.

C models that include dynamic memory allocation, pointers, and the functions malloc and free are mapped onto hardware in [43]. The implementation method in [43] instantiates a custom (to the application) hardware memory

allocator. The allocator is coupled with the specific memory architecture. Moreover, this work supports the resolution of pointers without any restriction on the underlying data structures. Many networking and multimedia applications are implemented in hardware or mixed hardware/software platforms and they feature heavy use of complex data structures which are sometimes stored in one or multiple memory banks. An immediate result of this is that some features of C/C++ which were originally designed for software development are now strong candidates for hardware design as well. The SpC tool which was developed in [43] resolves pointer variables at compile time and thus C functional models are synthesized into hardware efficiently. In a hardware implementation of pointers, memory allocation may be distributed onto multiple memories, and the data which are referenced by the pointers may be stored in memories, registers, or wires. Therefore, the synthesis tool needs to automatically generate the appropriate circuit to allocate, access (read/write) and deallocate data. The potential values of all pointers of an application program are identified by a compiler technique called pointer analysis. In order to implement dynamic memory allocation in hardware, there is a need to synthesize circuits to access, modify, or deallocate the location which is referenced by each pointer. For this purpose, the aliasing information [43] must be both safe and accurate. The authors in [43] assume that the computational complexity of flow-sensitive and context-sensitive analysis is not high because of the small size and simplicity of the programs and function calls which are used in hardware synthesis. This of course is not guaranteed since modern system descriptions could easily contain some thousands of lines of hierarchical code to describe complex hardware architectures. The subset of C which is accepted by the methodology in [43] includes malloc/free and all types of pointers and type casting. However, pointers that point to data outside the scope of a process (e.g., global variables) are not allowed. The synthesis of functions in C, and, therefore, the resolution of pointers and malloc/free inside of functions, is not included in this work. In order for the C code with the pointers to be efficiently mapped onto hardware, first the memory is partitioned into sets which can include memories, registers, or wires, and which can also represent pointers. Pointers are resolved by encoding their value and generating branching statements for loads and stores. Dynamic memory allocation and deallocation is executed by custom hardware memory allocators. The SpC tool [43] takes a C function with complex data structures and generates a Verilog model. The different techniques and optimizations described above have been implemented using the SUIF compiler environment [44]. The memory model consists of distinct location sets, and it is used to map memory locations onto variables and arrays in Verilog. Then, the generated Verilog module can be synthesized using commercial synthesis tools such as the behavioral compiler of synopsys. The case studies that evaluated and tested this methodology included a video algorithm and an asynchronous transfer mode (ATM) segmentation engine.

A heuristic for scheduling behavioral specifications that include a lot of conditional control flow is presented in [45]. This heuristic is based on a specific intermediate design representation which, apart from established techniques such as chaining and multicycling, enables more advanced techniques, such as conditional resource sharing and speculative execution, which are suitable for scheduling conditional behaviors. This work intends to bridge the gap in design implementation quality between HLS results from dataflow-dominated descriptions, and those from conditional control-flow-dominated source models. Generally, although HLS was accepted by the engineering community earlier for dataflow oriented applications, it took some time before it became adopted, and it is still not widely accepted for designs that contain complex conditional control flow, such as nested if-then-else and loop constructs. This intermediate design representation is called hierarchical conditional dependency graph (HCDG). The heuristics for HLS tasks that are invented for the HCDG have been developed to deal with complex control flow that involves a degree of control hierarchy. HCDGs introduced two new concepts: a hierarchical control representation and the explicit representation of both data and control dependencies in the design. This explicit representation of control dependencies is suitable for exploring maximum parallelism in the implementation, by rearranging these control dependencies. Because exploiting parallelism is easier for custom hardware designs than for software ones, being able to express maximum parallelism at the intermediate form level of a hardware design is essential. The HCDG can be successful in avoiding the negative effects of syntactic variance effects in the specification code of the designed system.

The hierarchical control representation of HCDG enables the HLS tasks such as scheduling, allocation, and binding. In [45], symbolic names are given to the Boolean conditions under which the various operations are executed and values are assigned to variables. Those symbolic names are called guards. In an HCDG, there are two types of nodes: guard nodes and operation nodes. Guard nodes represent the symbolic names of the various conditions under which operations are executed. Operation nodes represent I/Os, computations, data multiplexing, and storage elements. In an HCDG, there are two types of edges: data dependencies and control dependencies. Data dependencies are precedence constraints from one operation node to another. This defines the dataflow-dependent order of operation execution order. Control dependencies designate which conditions (guards) must evaluate to true so that the data values are computed and considered as valid. Each operation node has its control dependency edge from its guard. Guards can be also hierarchical, which results into a graphical representation of nested control constructs (e.g., and if-then-else nested inside another if-then-else, etc.). Therefore, there is a guard hierarchy graph for every design in [45]. Deriving HCDGs from conditional behaviors is being exercised in [45], but deriving them from loop constructs is reported in the particular work [45] as being the subject of future work. In order to schedule conditional behaviors efficiently, the mutual exclusiveness of the conditions must be exploited. This means being able to conditionally share resources and schedule operations effectively. In order to do this, complete

lists of mutually exclusive guards have to be constructed. For large and complex designs, this means that a very large number of mutual exclusiveness tests have to be performed on potential pairs of guards. Nevertheless, this number of tests can be drastically reduced in [45] by using the inclusion relations represented by the guard hierarchy graph. Using the above techniques, the following HLS transformations are enabled: lazy execution, node duplication, speculative execution, false-path elimination, and conditional resource sharing. Moreover, operation chaining and multicycle operations are considered. A special priority function based on guard hierarchy and graph node mobility is utilized in order to obtain the node priorities when executing scheduling. Mutual exclusiveness information is very useful for applying register allocation and for other types of resource sharing such as the one applied to interconnects. The HLS techniques presented in [45] were implemented in a prototype graphical interactive tool called CODESIS which used HCDG as its internal design representation. The tool can generate VHDL or C code from the HCDG, but no reports about translating a standard programming language into HCDG are known so far.

The HLS approach presented in [46] utilizes a coordinated set of coarse-grain and fine-grain parallelizing transformations on the input design model. These transformations are executed in order to deliver synthesis results that do not suffer from the negative effects of complex control constructs in the specification code. These transformations are applied both during a presynthesis phase and during scheduling, in order to improve the quality of synthesis output. During presynthesis, the following transformations are applied: common subexpression elimination (CSE), copy propagation, dead code elimination, loop-invariant code motion, as well as restructuring transformations such as loop unrolling and loop fusion. Then, during scheduling, aggressive speculative code motions (transformations) are used to reorder, speculate, and sometimes duplicate operations in the design. In this way, maximum parallelizing is applied on the synthesis results. A technique called dynamic CSE, dynamically coordinates CSE, speculation, and conditional speculation during scheduling. During scheduling, specific code motions are enabled, which move operations through, beyond, and into conditional blocks with the purpose of maximizing parallelism and, therefore, increase design performance. The scheduling heuristic, the code motion heuristic, dynamic transformations, and loop pipelining are executed. All of these tasks use functions from a tool library, which includes percolation and trailblazing, speculative code motions, chaining across conditions, CSE, and copy propagation. Then, during the binding and control synthesis steps, the operation and variable binding as well as FSM generation and optimization are executed. All these techniques were implemented in the SPARK HLS tool, which transforms specifications in a small subset of C into RTL VHDL hardware models. A resource-constrained scheduler is used in SPARK and it is essentially a priority-based global list scheduling heuristic. The user provides SPARK with a library of resources, which include among other details the type and number of each resource. This user library is used by the HLS tool, to allocate operations and registers onto library components. In terms of intermediate design representations, SPARK utilizes both control/data flow graphs (CDFGs) as well as an encapsulation of basic design blocks inside hierarchical task graphs (HTGs) [46]. HTGs allow for coarse-grain code restructuring such as loop transformations and an efficient way to move operations across large pieces of specification code. This is why the combination of CDFGs and HTGs in SPARK is so successful. Nevertheless, there are serious restrictions on the subset of the C language that SPARK accepts as input, and limitations such as inability to accept design hierarchy modules (e.g., subprograms) and of "while" type of loops. SPARK is validated in [46] by synthesizing three large examples: MPEG-1, MPEG-2, and the GIMP image processing tool.

Typical HLS tasks such as scheduling, resource allocation, module binding, module selection, register binding, and clock selection are executed simultaneously in [47] so as to achieve better optimization in design energy, power, and area. The scheduling algorithm utilized in [47] applies concurrent loop optimization and multicycling and it is driven by resource constraints. The state transition graph (STG) of the design is simulated in order to generate switched capacitance matrices. These matrices are then used to estimate power/energy consumption of the design's datapath. The initial schedule is optimized by multiple execution sequences of module selection, module sharing, and register sharing tasks. Nevertheless, the input to the HLS tool which was developed in [47] is not program code in a popular language but a proprietary format representing an enhanced control-data-flow graph (CDFG) as well as an RTL design library and resource constraints. In order to facilitate the capturing of control constructs such as if-then-else and loops, as well as memory access sequences, special nodes and edges were added to enhance this proprietary CDFG. The scheduler takes the CDFG and resource constraints as input and produces a result in the form of an optimized STG. In the synthesis algorithm, the cost function (for optimization) can be area, power, or energy. The synthesis process is iterative and it continuously improves the cost function until all constraints and data dependencies are met. The iterative improvement algorithm is executed by means of multiple passes until there is no potential improvement on the cost functions. In every pass, a sequence of the following moves is generated; the moves can be module selection, module sharing and register sharing. After each move, the behavior of the system is rescheduled and the cost is reestimated. If that move generates the best reduction of cost, then the move is saved, otherwise, different moves are selected. If the cost is reduced in the current pass, then a new pass is generated and the scheduling continues. This iterative process runs until there is no potential improvement in the cost functions. The tool generates RTL Verilog implementations. The developed HLS system is targeted at control-intensive applications, and it is also applicable to dataflow-dominated designs. The system was tested using a number of control-intensive benchmarks, such as for loop, concurrent loops, nested loops, greatest common divisor, a fifth-order Elliptic wave filter, and a popular dataflow-dominated benchmark. The

synthesis results focused more on power reduction up to 70% rather than area or speed results. Most of the benchmarks took a number of minutes to execute on a conventional Pentium III PC.

An incremental floorplanner is described in [48] which is used in order to combine an incremental behavioral and physical optimization into HLS. These techniques were integrated into an existing interconnect-aware HLS tool called ISCALP [49]. The new combination was named IFP-HLS (incremental floorplanner high-level synthesis tool), and it attempts to concurrently improve the design's schedule, resource binding and floorplan, by integrating high-level and physical design algorithms. Moreover, the impact of interconnect on area and power consumption of integrated circuits was considered in this work. To define the problem this method is based on the following equation:

$$T_{\text{clock}} = \frac{T_s}{c_{\text{steps}}}, \tag{1}$$

where T_{clock} is the system clock period, T_s is the constraint on the input data rate (sample period), and c_{steps} is the number of clock cycles required to process an input sample. Given c_{steps}, an ASAP schedule is generated for an initial solution to determine whether it meets timing. An iterative improvement algorithm is then applied on this initial solution, in order to reduce the switched capacitance while it still satisfies the sample period constraint. From the way the problem and the solution are defined in this HLS approach, it seems that the latter is suitable for dataflow-dominated designs and not for control-intensive applications. IFP-HLS generates one initial solution, at the maximum number of c_{steps} and then it applies incremental floorplanning and it eliminates redundant operations. In this way, the solution is improving as the c_{steps} decreases. If a solution meets its timing requirement after rescheduling, then rebinding is not necessary. In any other case, it rebinds some tasks and uses parallel execution to improve performance. Possible pairs of tasks that are initially assigned to the same functional unit are split onto separate functional units [48].

For a given c_{steps}, the floorplan is incrementally modified to see if this improves the solution quality. If it does, then this change is saved. Otherwise, the floorplan change is rejected and other modifications are attempted to determine whether they improve the solution, and so on [48]. In order to guide these changes, the tool extracts physical information from the current incrementally generated floorplan. IFP-HLS incrementally performs scheduling, allocation, and binding by modifying iteratively c_{steps}, and it determines which operations need to be rescheduled or rebound (split) in order to meet the timing constraints. In each step, the floorplanner is updated. An incremental simulated annealing floorplanner is embedded into the IFP-HLS tool which was designed for design quality and not for speed. The floorplanner handles blocks with different aspect ratios and generates nonslicing floorplans. Every synthesis move either removes a single module or it splits a module into two. Therefore, most of the modifications are small and their effects on the floorplan are local, rather than global. In this way, an existing floorplan can be used as the base for each new floorplan. In practice, the authors found this approach to deliver quality-of-results and performance improvements, even compared with a very fast constructive floorplanner.

Fifteen different benchmarks were used to evaluate the utility of this approach in [48]. The average improvements of IFP-HLS over ISCALP, for implementations with nonunity aspect ratio functional units, are 14% in area, 4% in power consumption, 172% in reduction in the number of merge operations, and 369% in CPU time. The average improvements of IFP-HLS over ISCALP, for implementations with unity aspect ratio functional units, are 12% in area, 7% in power consumption, 100% in reduction in the number of merge operations, and for some benchmarks the IFP-HLS CPU run time was 6 times less than that of the ISCALP method.

The study in [50] discusses an HLS methodology which is suitable for the design of distributed logic and memory architectures. Beginning with a behavioral description of the system in C, the methodology starts with behavioral profiling in order to extract simulation statistics of computations and references of array data. This allows the generation of footprints which contain the accessed array locations and the frequency of their occurrence. Array data reference operations with similar access patterns are grouped together into a computation partition, using these footprints. A method to assign each such partition onto a different subsystem is used. In this way, a cost function is minimized that includes balancing the workloads, synchronization overheads, and locality of data accesses. Then, array data are distributed into different partitions. This is done so that the data accesses will be as much as possible local to each subsystem, based on the clustering of their reference operations. Synchronization code is inserted into the implementation's behavior, in order to implement correct communication between different partitions. This results into a distributed logic/memory microarchitecture RTL model, which is synthesizable with existing RTL synthesizers, and which consists of two or more partitions, depending on the clustering of operations that was applied earlier. These techniques are implemented into an industrial tool called Cyber [51]. Several benchmark applications were run on the tool to produce distributed logic/memory implementations. The results featured a performance increase of up to twice and reduction up to 2.7 times of the delay X energy product over single-memory and homogeneously partitioned designs.

Communicating processes which are part of a system specification are implemented in [52]. In contrast to the conventional HLS approach which synthesizes each concurrent process of the system individually, the impact of the operation scheduling is considered globally in [52], in the system critical path (as opposed to the individual process critical path). First, the system is scheduled by assuming that there are unlimited resources for each process. Then, the scheduled design is simulated, and using the simulation's execution traces, system performance is analyzed and the critical path(s) of the behavior is (are) extracted. Using this information about the design, the criticality of operations is calculated based upon whether they belong to the critical path(s) or the near-critical path(s). Then, the relative

resource requirement of each process is calculated which depends on the type and number of critical operations that a process contains. With this information for each process, the resources for the overall system are budgeted. The resource budget is then used as a constraint to reschedule the whole design. The rescheduled design is simulated again and the critical paths are yet one more time extracted from the traces. If the critical path changes, then the above process is repeated again and again until the critical path remains the same after each resource reallocation. When the extracted critical paths become the same, and using the last resource budget, the behavior model is passed to the rest of the HLS tasks, such as resource sharing and generation of the controller and datapath. In this way, the RTL hardware implementation of the multiple processes is built. It is argued by the authors in [52] that this methodology allocates the resources where they are mostly needed in the system, which is in the critical paths, and in this way it improves the overall multiprocess designed system performance.

The work in [53] contributes towards incorporating memory access management within an HLS design flow. It mainly targets digital signal processing (DSP) applications but also more general streaming systems can be included along with specific performance constraints. A particular memory sequencer architecture is discussed in [53] and utilized by its methodology. This methodology can pipeline both static and dynamic memory access sequences. In order to take advantage of the memory sequencer, specific enhancements of the typical HLS flow are introduced. The targeted architecture template for the signal processors includes the processing unit (PU) which contains the datapath and a controller, the memory unit (MemU) which executes pipeline accesses to memories, and the communication unit (ComU) which handles communication from/to the rest of the design's computing environment. The synthesis process is performed on the extended data-flow graph (EDFG) which is based on the signal flow graph. The EDFG models both the access and data computations, the transfers of data, and the condition statements for addressing, computation, and data transfers, respectively. Mutually exclusive scheduling methods [54, 55] are implemented with the EDFG. This is achieved because EDFG allows for data and conditional semantics to be handled in the same way, and thus the exploitation of potential design parallelism can be maximized.

Special EDFG structure nodes are defined, so as to represent the arrays and their components access in the application. In order to handle memory access dependencies, the write after write, and read after write dependencies are taken into account and the structure nodes are renamed after for example, a write access. This is done in order to remove ambiguous dependency accesses for scalar load and store operations. Conditional nodes are also defined in EDFG. This is done so as to model conditioned operations and memory accesses. There are dependencies between the calculation of the condition's value and all the included conditioned operations (inside the conditional structure). The function $t(u)$, for operation u, annotates the EDFG edge, in order to capture the delay (time) that operation takes from the change of its inputs to propagate the result at the outputs (see the following paragraphs for HLS internal format descriptions). This delay is essentially the transfer time from the predecessor of the operation, to its successor. In a first annotation step, all operations, including the dynamic address calculations, are assumed to be implemented in the datapath unit of PU. Moreover, using the available memory mapping data, the data nodes are also annotated. In order to transform an annotated graph into a coherent graph [53], the location of the graph nodes is checked. If the location of all the predecessors and successors of a node is not the same, then a transfer node is inserted. Based on a set of criteria [53], dynamic address computation operations are moved from the datapath unit onto the sequencer, which is called address computation balancing. This is done so as to increase the overall system performance. The processed by these annotations and improvements graph is then given to the GAUT HLS tool [56] to perform operator selection and allocation, scheduling, and binding. GAUT implements a static list scheduling algorithm so as to maximally parallelize the initial schedule. This methodology is suitable for dataflow-dominated applications such as video streaming and linear DSP algorithms.

A combined execution of decomposition and pattern-matching techniques is applied on HLS problems, in order to reduce the total circuit area in [57]. The datapath area is reduced by decomposing multicycle operations, so that they are executed on monocycle functional units (FUs that take one clock cycle to execute and deliver their results). Furthermore, when other techniques used to guide operator, decompositions such as regularity exploitation can deliver high-quality circuits. In this way, operations extract their most common operation pattern, usually repeated in many clock cycles. Thus, the circuit that is needed to execute the selected operation pattern is shared among many operations in many cycles, and, therefore, the total hardware area is drastically reduced. The algorithm presented in [57] takes as input a behavioral design description and time constraints, and it selectively decomposes complex operations into smaller ones, in order to schedule in every clock cycle a similar number of decomposed fragments of operators, with the same pattern. This method considers only operation decompositions that meet the time constraints. Also, some of the decompositions reduce the length of the clock cycle. This results into increasing the system's performance. The HLS output is a complete datapath with FUs, multiplexers, registers and some glue logic, as well as a controller. The number, type, and width of the resources used in the produced datapath are generally independent from the input behavioral hardware description. This is due to the operation decompositions which are applied through the synthesis process.

A simple formal model that relies on an FSM-based formalism for describing and synthesizing on-chip communication protocols and protocol converters between different bus-based protocols is discussed in [58]. The discussed formalism enables the detailed modeling of existing commercial protocols, and the analysis of protocol compatibility. Furthermore, the most important is that it facilitates the

automated and correct-by-construction synthesis of protocol converters between existing popular communication protocols. The utilized FSM-based format is at an abstraction level which is low enough for its automatic translation into HDL descriptions. The generated HDL models are synthesizable with commercial tools. Typically a system-on-a-chip (SoC) includes intellectual property (IP) blocks that are connected together either directly on a bus or via specialized wrappers. The wrappers play the role of converters from the IP's interface into the bus protocol, so that all the SoC parts collaborate with each other. Usually engineers build these wrappers manually, and this is done, based on a nonformal knowledge about the bus protocol. Up to the publishing of this work, there were no automated converter synthesis techniques employed in industrial or academic practice. The work in [58] contributes towards three aspects of protocol converter synthesis: a formal, FSM-based model for protocol definition, a precise definition of protocol compatibility, and a definition of converters and converter correctness (for a given pair of existing and known protocols).

Synchronous FSMs with bounded counters that communicate via channels are used in [58] to model communication protocols. Protocol channels are attributed with a type and a direction. A channel type can be either control or data, and direction can be either input or output. A channel action can be write, read, or a value test on a channel. The bounded counters are used in the model so as to keep a data item valid on a channel, for a number of clock cycles. Between two changes in the counter value, any read or write action indicates repetition of data values. Bounded counters facilitate smaller and precise models of data bursts on channels. Protocols which execute concurrently are described by the parallel composition of the protocols. The parallel composition of two protocols describes all the possible control states that the protocols may be in, when they run concurrently. The following constraints must be satisfied, in order to make sure that data flows between these protocols: data is read by one protocol only when it is written by the other; a specific data item can be read as distinct exactly once; no deadlocks can occur and livelocks can always be avoided. The last condition makes sure that every data transfer can terminate in a finite number of steps. In order to satisfy the second constraint of correct data flow, the data actions, along a path between two protocols, need to be correct. This means that every written data item is read as new before it is read as a repeated item. Also, a new data item can be written only if the previous one has been read. Furthermore, there should be no read repetition between a new write and a new read action. The utilized formal model allows for analyzing and checking compatibility between two existing protocols. The model devised in [58] is validated with an example of communication protocol pairs which included AMBA APB and ASB. These protocols are checked regarding their compatibility, by using the formal model.

The following constraints have to be checked by the formal techniques, so as to make sure that correct data flow is happening through a protocol converter in [58]: data is read by a protocol(/converter) only when the data is written by the converter(/a protocol); a data item can be read as distinct exactly once; no deadlocks and livelocks can occur; every data that is sent (written) from P1 (P2) to the converter will be sent (written) by the converter to P2 (P1); every data written to the converter was previously written by a protocol. A converter itself is an FSM with bounded counters and a finite buffer for each output data channel. A pair of protocols, data channel mapping, and buffer sizes is given as input to the converter synthesizer. The converter synthesizer generates the most general (correct) protocol converter. Protocol converter testcases that were used to evaluate the work in [58] incluced an ASB to APB converter and a set of converters between the open core protocol (OCP) and the AMBA family of bus protocols. The existing synthesis framework is limited to protocols that can be defined by a single FSM, and for more than one FSM per protocol description capabilities, future work on this is envisaged by the authors.

The methodology of SystemCoDesigner [59] uses an actor-oriented approach so as to integrate HLS into electronic system level (ESL) design space exploration tools. Its main aim is to automate the design and building of correct-by-construction system on a chip (SoC) implementations from a behavioral model. The design starts with an executable SystemC system model. Then, commercial synthesizers such as Forte's Cynthesizer are used in order to generate hardware implementations of actors from the behavioral model. The produced actor implementations are characterized based on their area (number of lookup tables, and block memories) and their latency. This enables the design space exploration in finding the best candidate architectures (mixtures of hardware and software modules). After deciding on the chosen solution, the suitable target platform is then synthesized with the implementations of the hardware and software parts.

Modules or processes are modeled in [59] as actors which communicate with other actors via a number of communication channels. This is the starting point for modeling a system in [59]. The specification language of an actor is a subset of SystemC which is defined in SysteMoC library [60]. SystemC actors communicate via SystemC FIFO channels, and their functionality is implemented in a single SystemC thread. Each SysteMoC actor is specified by a finite state machine (FSM), which defines the communication methods that the FSM controls. Each such actor can be transformed into hardware (using Cynthesizer) or software implementations. Moreover, the use of a commercial tool for hardware synthesis allows for arriving at design solution evaluations, so as to decide about the most suited solution, in terms of hardware resources and system throughput. The performance information, the executable actor system specification, and an architecture template are the inputs for design space exploration. The architecture template is represented by a graph, which represents all the possible hardware modules, the processors, and the communication infrastructure. The designer can select from within this graph the solutions that satisfy the user requirements and which produce tradeoffs between hardware size and performance.

The final step of this methodology is to generate the FPGA-based SoC implementation from the chosen hardware/software solution. This is done by connecting existing IP blocks and processor cores with the communication elements from an appropriate library. Moreover, the program code for each processor is generated, in order to achieve rapid prototyping. The final FPGA bitstream is generated in [59] using the Xilinx EDK (Embedded Development Kit) tools. A motion-JPEG test application was used to validate the proposed methodology in [59]. The architecture template used an embedded Microblaze processor core, 224 FIFOs, and 19 modules generated by the HLS tool. The complete system specification used 319 actor mapping edges and the design space exploration produced 5×10^{33} different alternative solutions. An instance of the Microblaze core including memory and bus resources was instantiated during platform synthesis and for each processor used. The rest of the hardware modules were inserted in the form of Verilog netlists which were generated by Cynthesizer (HLS) and Synplify by Synplicity (RTL synthesis) tools. Moreover, FIFO primitives were instantiated for communication between the system's blocks. For the particular testcase (JPEG), the objectives taken into account during design space exploration, included throughput, latency, number of flip-flops, number of lookup tables, block-RAMS, and multipliers. Based on the proposed methodology, the formal underlying mechanisms and the used examples, it was concluded that SystemCoDesigner method is suitable for stream-based applications, found in areas such as DSP, image filtering, and communications. Up to now, there are no indications on how well this methodology would perform in applications with complex control flow.

A formal approach is followed in [61] so as to prove that every HLS translation of a source code model produces a RTL model that is functionally equivalent to the one in the behavioral input to the HLS tools. This technique is called translation validation and it has been maturing via its use in the optimizing software compilers. In [61], HLS is seen as a sequence of refinements of the original specification (behavioral) code down to the final RTL implementation model. It is argued in [61] that if it is formally proved that these refinement steps maintain the behavioral properties of the original design, then it reduces the need to reverify (e.g., with simulations) the correctness of the produced RTL. The validating system in [61] is called SURYA and it uses the Simplify theorem prover to implement the validation algorithms. SURYA was used to validate the SPARK HLS tool [46], and SURYA managed to find two bugs in the SPARK compilations, which were unknown before.

The translation validation methodology [61] consists of two algorithm components: the checking algorithm and the inference algorithm. Given a simulation relation, the checking algorithm determines whether or not this relation is a correct refinement simulation relation. The inference algorithm uses the specification and the implementation programs to infer a correct simulation relation between them. The inference algorithm establishes a simulation relation that defines which points in the specification program are related to the corresponding points in the implementation program. First the inference algorithm is applied to infer a simulation relation and then the checking algorithm is used to verify that the produced relation is the required one, in order to check that a program is a relation of another program. The translation validation algorithm models the environment of the design as a set of processes, which are executed in parallel with the processes of the specification and the implementation. The simulation relation includes a set of entries of the form gl1, gl2, $\varnothing$, where gl1 and gl2 are locations in the specification and implementation programs, respectively, and $\varnothing$ is a predicate over variables in the specification and the implementation. The pair gl1, gl2 captures how control flow points (control states) relate in the specification and implementation programs, and $\varnothing$ captures how data are related between the two programs. The checking algorithm establishes correctness of a relation, if having an entry (gl1, gl2, $\varnothing$), and if the specification and implementation programs start executing in parallel from states gl1 and gl2, respectively, where $\varnothing$ holds, and they reach another pair of states gl1′ and gl2′, then in the resulting simulation relation entry (gl1′, gl2′, $\varnothing'$), $\varnothing'$ holds in the resulting states. If there are multiple paths from an entry, the checking algorithm in [61] checks all of them.

In [61], the inference algorithm begins with finding the points in the specification and simulation programs that need to be related in the simulation. Then, it moves further down in the control flow, in both the specification and implementation programs until it finds a branch or a read/write operation on a visible channel. Furthermore, the algorithm corelates the branches in the specification and the implementation, and it finds the local conditions which must hold in order for the visible instructions to match. When instructions write to visible output channels, the written values must be the same. When instructions read from externally visible input channels, the local conditions state that the specification and implementation programs read from the same point in the conceptual stream of input values. Once the related pairs of locations gl1, gl2 are all collected, a constraint variable is defined to represent the state-relating formula that will be used in the relation for this pair. Then, a set of constraints are applied on these constraint variables to make sure that the relation is indeed a simulation. The first kind of constraints makes sure that the computed simulation relation is strong enough, so that the visible instructions have the same behavior in the specification and the implementation programs. A second kind of constraints is used to state the relationship between a pair of related locations and other pairs of related locations. When all the constraints are generated, then an algorithm sets all the constraint variables to true and strengthens the constraint variables until a theorem prover can show that all constraints are satisfied.

An assumption is made by the formal model of refinement in [61], that the specification and the implementation are single entry and single exit programs. A transition diagram represents every process in these programs. This diagram uses generalized program locations and program transitions. A program location represents a point in the control flow of the program and it is either a node identifier,

or a pair of two locations which refer to the state of two processes that are running in parallel. A program transition is represented by instructions and it defines how the program state changes from one location to another. Within this concept of models, two execution sequences of programs are equivalent if the two sequences contain visible instructions that are pairwise equivalent. In case that the inference algorithm cannot find an appropriate relation, then the user can provide the simulation relation by hand, and use the checking algorithm in order to verify that the relation is a correct one. The SURYA validation system was applied on SPARK HLS compilations to evaluate the equivalence between the intermediate representation (IR) of SPARK and the scheduled IR of the same translation process.

The replacement of flip-flop registers with latches is proposed in [62] in order to yield better timing in the implemented designs. The justification for this is that latches are inherently more tolerant to process variations than flip-flops. The latch replacement in [62] is executed not only during the register allocation task, but in all steps of HLS, including scheduling, allocation, and control synthesis. A prerequisite for correct latch replacement is to avoid having latches being read and written at the same time. The concept of the p-step is introduced in [62]. The p-step is the period during which the clock is high or low. Using the p-step as the elementary time unit instead of the conventional clock-cycle makes scheduling more flexible and it becomes easier to reduce the latency. In [62], the list scheduling algorithm is enhanced with the p-step concept, and a method is used to reduce the latency by determining the duty cycle of the clock, and therefore the p-step. In order to control the p-step schedule, dual-edge-triggered flip-flops are used in controller synthesis, since both clock edges are used to define the boundaries of the p-steps. These techniques were integrated into a tool called HLS-1. HLS-1 translates behavioral VHDL code into a synthesized netlist. The method was evaluated with a number of behavioral benchmarks as well as an industrial H.264 video coding application.

The execution time of operators (e.g., multipliers and adders) is formulated mathematically in [62]. This is realized using the clock period, the maximum delay of a functional unit (FU) executing the specific operator, the clock-to-Q delay, the setup time for the utilized flip-flops, and the delay of multiplexers. The transparent phase is the period of time that the clock is high, and the nontransparent phase is the remainder of the clock period. The mathematical analysis of timing using latches is formulated in [62] using the clock period, the time of the transparent phase, and the residual delay. The latter is the remainder of the delay time from modulo-dividing with the clock period. Using these mathematical models, and assuming a 30% duty cycle, it is shown in [62] that an example multiplier will need four p-steps, if it is scheduled to start at the first transparent phase, and will need three p-steps if it is scheduled to start at the second (nontransparent) phase. When flip-flops are used instead, the same multiplier will need two clock steps (equivalent to four p-steps) to execute which is longer than in the case of latches. The whole schedule can be optimized by using the p-step as the basic scheduling unit and by modifying the duty cycle of the clock. Tighter schedules can be produced since p-steps allow scheduling at both edges of the clock. Moreover, register allocation is facilitated since operations can complete in a nontransparent p-step, and, therefore, read/write conflicts that are inherent in latch registers can be resolved. This method assumes that the delay of the controller is negligible, as compared to the transparent and nontransparent phase times. Nevertheless, implementing registers with latches instead of edge-triggered flip-flops is generally considered to be cumbersome due to the complicated timing behavior of latches.

2.5. Considering Interconnect Area and Delay. Usually, HLS tools estimate design area and timing using an aggregate sum of area and timing models of their functional unit, storage, and interconnect elements taken from the component libraries that they use. However, when moving into the deep submicron technologies, the area and timing of the chip's wires become significant compared to those of the implementation logic. Thus, new models and optimization approaches are needed in order to take into account the effect of the interconnections. In this direction, mature HLS tools consider more accurate models of the impact of interconnect on area and timing of the implementation. Given a DFG, performance estimation tools use a set of resources and resource delays, as well as the clock cycle to calculate a lower-bound completion delay for nonpipelined, resource constrained scheduling problems [63]. The work in [64] presents an algorithm which computes lower bounds on the number of functional units of each type. These functional units are required to schedule a DFG in a given number of control steps [64]. The lower bounds are found by relaxing either the precedence constraints or the integrity constraints, and the method estimates functional-unit area in order to generate resource constraints and thus reduce the search space, or in combination with an exact formulation for design space exploration.

In [65], a high-level approach to estimate the area is proposed. This approach focuses on predicting the interconnect area and it is suitable for standard cell implementations. The work in [66] proposes simultaneous functional unit binding and floorplanning during synthesis. An analytical technique, which includes the placement and binding problems in a single, mixed, ILP model, is discussed in [67] for a linear, bit-slice architecture. This model is able to minimize the overall interconnections of the resulting datapath. In [68] synthesis for datapath is discussed which is based on multiple-width shared bus architecture. This technique utilizes models of circuit area, delay, power consumption, and output noise which are related to functional unit grouping, binding, allocation, and different word-lengths. Functional unit grouping and multiple-width bus partitioning are executed during allocation but before scheduling. According to the authors, this increases the synthesis flexibility and the possibility for better synthesis results. The aim is to reduce the delay, as well as the interconnection cost and power consumption of the implementation.

2.6. Synthesis for Testability. HLS for testability can be achieved by reducing the number of self-looped registers, while considering the tradeoff between testability improvement and area increase. In [69], the switching property of multiplexors and buses is used to reduce the area and test generation costs. This is achieved by analyzing the location of switches during the selection of partial test registers, and by using these switches to transfer test data through them. In [70], the testability of the design is analyzed at the behavioral level, by considering loops and other control structures. This is done so as to improve the testability by including test constraints, during allocation of registers and production of interconnections. Simultaneous scheduling and allocation of testable functional units and registers under testability, area and performance constraints, are performed in [71] using a problem-space genetic algorithm. Binding for testability is implemented in two stages in [72]. First, a binder with test cost function generates a design almost without any loops. Then, the remaining register self-loops are broken by alternating the module and register binding.

A two-stage objective function is used in [73] so that synthesis of the behavioral code requires less area and test cost, by estimating the area and testability as well as the effects of every synthesis transformation. Next, a randomized branch-and-bound decent algorithm is used to identify that particular sequence of transformations. This achieves the best results in terms of area and test cost. A high-level synthesis-for-testability approach was followed in [74] where the testability of the hardware was increased by improving the controllability of the circuit's controller. This is a testability increase via improving the circuit's controller design.

2.7. Synthesis for Low Power. A number of portable and embedded computing systems and applications such as mobile (smart) phones, PDAs, and so forth, require low power consumption, therefore, synthesis for low energy is becoming very important in the whole area of VLSI and embedded system design. During the last decade industry and academia invested on significant part of research regarding VLSI techniques and HLS for low power design. In order to achieve low energy in the results of HLS and system design, new techniques that help to estimate power consumption at the high-level description level are needed. In [75], switching activity and power consumption are estimated at the RTL description taking also into account the glitching activity on a number of signals of the datapath and the controller. Important algorithmic properties in the behavioral description are identified in [76] such as the spatial locality, the regularity, the operation count, and the ratio of critical path to available time. All of these aim to reduce the power consumption of the interconnections. The HLS scheduling, allocation, and binding tasks consider such algorithmic statistics and properties in order to reduce the fanins and fanouts of the interconnect wires. This will result into reducing the complexity and the power consumed on the capacitance of the interconnection buses [77].

The effect of the controller on the power consumption of the datapath is considered in [78]. The authors suggest a special datapath allocation technique that achieves low power. Pipelining and module selection was proposed in [79] for low power consumption. The activity of the functional units was reduced in [80] by minimizing the transitions of the functional unit's inputs. This was utilized in a scheduling and resource binding algorithm, in order to reduce power consumption. In [81], the DFG is simulated with profiling stimuli, provided by the user, in order to measure the activity of operations and data carriers. Then, the switching activity is reduced, by selecting a special module set and schedule. Reducing supply voltage, disabling the clock of idle elements, and architectural tradeoffs were utilized in [82] in order to minimize power consumption within an HLS environment.

Estimating the power consumption of a system, chip of SoC (system-on-a-chip), often requires a lot of design detail to be taken into account. This, in turn, leads to very slow simulations, although these are accurate, and thus results into very long design times, when low energy is considered throughout the development flow. In order to achieve faster simulation times for energy consumption, than RTL simulations, the cycle-accurate, bit-accurate (CABA) level is utilized in [83], in order to produce fast and more effective power estimates. Bit-accurate means that the communication protocols between design components are implemented at a bit level in the CABA model. Moreover, the CABA level simulation is realized cycle-by-cycle which achieves the required detail of the power estimates. In order to increase the efficiency and productivity in designing complex SoCs, the Model-Driven Engineering methodology (MDE) is adopted in [83]. MDE enables the automatic generation of system simulation components from models described in a diagrammatic UML format. For this, a UML profile is used (for more details about UML in designing systems, please refer to following paragraphs/sections). Using low level characterization, accurate power models are constructed which contain energy information for the different activities of system components. Then, these power models are simulated to add the power cost of the specific activity of the component to the overall power consumption estimate. The MDE module generation approach in [83] utilizes activity counters of white-box IP blocks and the connectivity of estimate modules for black-box IP blocks, which are used to calculate the energy consumption of complex SoCs. The simulations are realized at the SystemC level which achieves the intended estimation accuracy and simulation speed for the whole SoC.

The energy consumption of memory subsystem and the communication lines within a multiprocessor system-on-a-chip (MPSoC) is addressed in [84]. This work targets streaming applications such as image and video processing that have regular memory access patterns. These patterns can be statically analyzed and predicted at compile time. This enables the replacement of conventional cache memories with scratch-pad memories (SPMs), which are more efficient than cache, since they do not use tagged memory blocks. The way to realize optimal solutions for MPSoCs is to execute the memory architecture definition and the connectivity synthesis in the same step. In [84], memory and communication are cosynthesized, based on multiprocessor data reuse analysis.

The data reuse analysis determines a number of buffers which contain the frequently used data in the main memory. Then, the buffer mapping onto physical memory blocks is done simultaneously with communication synthesis, for minimal energy, while satisfying delay constraints.

2.8. Controller versus Datapath Tradeoffs. With the conventional synthesis approaches, the controller is implemented after the datapath is completed. Nevertheless, there are tradeoffs between datapath and controller in terms of area and performance. In this direction, excessive loop unrolling, particularly on many levels of nested loops in the behavioral model, can produce hundreds of states which generate a very large state encoder. This will reduce the achievable clock speed, which will definitely reduce the performance of the implementation. On the other hand, a very simple controller with straight command wires towards the datapath, will end up in heavy use of multiplexers (used for data routing through the functional units) which will have again a negative impact on the minimum clock cycle which can be achieved. In [85], the controller overhead is taken into account at every level of the hierarchy before the datapath is fully implemented. This approach is suitable for regular behavioral model algorithms. The system clock period is minimized during allocation in [86], by an allocation technique which considers the effect of the controller on the critical path delay.

2.9. Internal and Intermediate Design Formats Used in Synthesis. One of the most important aspects of compilers and therefore HLS synthesizers is the internal representation of the algorithmic information that they use. The internal format used in [1] consists of three graphs which all share the same vertices. If P is a program in DSL, after decomposing complex expressions into simple, three address operations, W is the set of variables and V the set of operations then the internal form S of the program contains the three directed graphs as follows:

$$S = (G_S, G_D, G_C), \tag{2}$$

where

$$\begin{aligned} G_S &= (V, E_S), \\ G_D &= (V \ U \ W, E_D), \\ G_C &= (V, E_C), \end{aligned} \tag{3}$$

and where the edges E correspond to E_S = sequence, E_D = dataflow, and E_C = constraints. The sequences set of edges are related to the dependencies from one operation to the other. The dataflow edges represent the data processing structure. These edges connect inputs and output of every operation inside the specified circuit. The constraints edges represent timing deadlines from each operation.

The HCDG [45] consists of operation nodes and guard nodes. The HCDG also contains edges which are precedence constraints for data which determine the order of operations, and for control (guards), which determine which Boolean expressions must evaluate to true, in order for the dependent operations to execute. Guards are a special type of node and they represent Boolean conditions that guard the execution of nodes which operate on data. If there are multiple but mutually exclusive assignments to the same variable which happens as an example within different if-then-else true/false branches, then this is represented in HCDG with a static single assignment. This assignment receives the results of the mutually exclusive operations through a multiplexing logic block (operation node in HCDG). The HCDG itself and the mutual exclusiveness of pairs of guards are utilized in [45] to aid the efficient HLS scheduling transformations of the synthesis method developed there.

The hierarchical task graph (HTG) is a hierarchical intermediate representation for control-intensive designs and it is used by the SPARK HLS tools [46]. An HTG is a hierarchy of directed acyclic graphs, that are defined by $G_{\text{HTG}}(V_{\text{HTG}}, E_{\text{HTG}})$, where the vertices V_{HTG} are nodes of one of the three following types.

(1) Single nodes that have no subnodes and they capture basic blocks. A basic block is a sequence of operations that have no control flow branches between them.

(2) (Hierarchical) Compound nodes that are HTGs which contain other HTG nodes. They capture if-then-else or switch case blocks or a series of other HTGs.

(3) Loop nodes that capture loops. Loop nodes contain the loop head and loop tail. The latter are simple nodes. Also, loop nodes contain the loop body which is a compound node.

The set of edges E_{HTG} indicate flow of control between HTG nodes. An edge (htg_i, htg_j) in E_{HTG}, where $htg_i, htg_j \in V_{\text{HTG}}$, where htg_i is the start node and htg_i is the end node of the edge, denotes that htg_j executes after htg_i has finished its execution. Each node htg_i in V_{HTG} has two special nodes, $htg_{\text{Start}}(i)$ and $htg_{\text{Stop}}(i)$, which also belong to V_{HTG}. The htg_{Start} and htg_{Stop} nodes for all compound and loop HTG nodes are always single nodes. The htg_{Start} and htg_{Stop} nodes of a loop HTG node are the loop head and loop tail, respectively. The htg_{Start} and htg_{Stop} of a single node represent the node itself [46]. HTGs are providing the means for coarse-grain parallelizing transformations in HLS, since they allow the movement of operations through and across large chunks of structured high-level code blocks. These blocks are complex conditional structures in the C language. In this way, the HTGs are very useful for the optimizing transformations of the HLS scheduler. Their drawback though is their high complexity in translating from regular programming languages' code into a combination of CDFGs and HTGs, such as it happens in the SPARK HLS tool [46].

A special version of CDFG is developed in [47], which serves two purposes: to be the input design format for their HLS tool, and to facilitate HLS optimizations. In particular, its purpose is to preserve and exploit further the parallelism inherent in control-dominated designs. In order to achieve this, the following special nodes were introduced in the CDFG.

(1) The SLP node represents the start of a loop and it selects between the initial value of the loop variable and the value, which is calculated from the last iteration of the executed loop.

(2) The EIF node represents the end of an if-then-else branch and it is used to select the correct value from the if-and-else branches. Both SLP and EIF nodes are implemented with multiplexers in the corresponding datapath.

(3) The BLP node sends the values of the variables which are computed by the current iteration of a loop, back to the SLP node which will select the value for the next loop iteration.

(4) The ELP represents the end of the loop. All the loop variables have to be passed through the ELP node, before they are fed to operations which use them, outside of the loop.

In this way, the synthesizer can identify easily the beginning, the end and the range of a loop using the SLP, BLP, and ELP nodes of the graph. The CDFG model in [47] includes support for memory operations such as load and store. Memory access sequences are defined with a special edge of the CDFG between the corresponding load and store nodes.

The extended data-flow graph (EDFG) which is used in [53] is a finite, directed, and weighted graph:

$$G = (V, E, t), \tag{4}$$

where V is the set of vertices (nodes) of computation and memory access, E is the set of edges representing precedence rules between the nodes, and the function $f(u)$ is the computation delay of node u. A path in G is a connected sequence of nodes and edges of the graph G. A synthesis system which includes optimizations on the computing part and the sequencer part of a custom DSP subsystem utilizes the above EDFG [53].

2.10. Diagrammatic Input Formats and Embedded Systems. The Unified Modeling Language or UML is a general purpose language for modeling in a diagrammatic way, systems which are encountered in object-oriented software engineering. In 1997, the Object Management Group (OMG) has added UML in its list of adopted technologies, and since then, UML has become the industry standard for modeling software-intensive systems [87]. Although UML is widely known for its diagrammatic ways to model systems, these models contain more information than just system diagrams. These diagrams are usually formed by experienced system architects. UML diagrams belong to two general categories: static or structural diagrams and dynamic or behavioral diagrams. UML models contain also information about what drives and checks the diagrammatic and other elements (such as the use cases). Structural diagrams include the class, component, composite structure, deployment, object, package, and profile diagram. Dynamic diagrams include the activity, communication, sequence, state, interaction overview, timing, and use case diagram [87].

OMG has defined a standard set of languages for model transformation called QVT (query/view/transformation). The set of QVT-like implementations of UML include the Tefkat language, the ALT transformation language which is a component of the Eclipse project, and the Model Transformation Framework (MTF) which was the outcome of the IBM project alphaworks [88]. Using QVT-like transformation languages, UML models can be translated to other languages, for example, programming language formats. UML can be extended with various ways for customization: tag definitions, constraints, and stereotypes, which are applied to elements of the model. UML Profiles are collections of such UML extensions that collectively customize UML for a particular design/application area (e.g., banking, aerospace, healthcare, financial) or platform (e.g. .NET). A stereotype is rendered as a name enclosed by *guillemets* ($\ll \gg$ or $\langle\langle \rangle\rangle$) and placed above the name of another element.

The area of real-time and embedded systems is a domain that UML can be extended so that it provides more detail, in modeling events and features of such systems, for example, mutual exclusion techniques, concurrency, and deadline specifications. After the standard of UML2 was adopted by OMG the earlier "UML Profile for Schedulability, Performance and Time" (SPT) for embedded systems was found problematic in terms of flexibility and expression capabilities, and consequently OMG issued a new request for proposals (RFPs) for a profile of this specific domain. The result of this effort was a UML profile called MARTE (Modeling and Analysis of Real-Time and Embedded systems) [89]. The primary goal of MARTE was to enable the formal specification of real-time constraints and the embedded system characteristics such as memory size/capacity and power consumption, modeling of component-based architectures, and the adaptability to different computational paradigms such as synchronous, asynchronous and timed.

In order to support modeling of embedded and real-time systems, the MARTE architecture offers the following four modeling capabilities.

(i) *QoS-aware modeling* with the following formats: HLAM: for modeling high-level RT QoS (Real-Time Quality of Service). NFP: for declaring, qualifying, and applying semantically well-formed nonfunctional issues. Time: for defining and manipulating time. VSL: the Value Specification Language is a textual language for specifying algebraic expressions.

(ii) *Architecture modeling* with the following features: GCM: for architecture modeling based on interacting components. Alloc: for specifying allocation of functionalities to entities that realize these functionalities.

(iii) *Platform-based modeling* with the following capabilities: GRM: for modeling and specifying the usage of common platform resources at system-level. SRM: for modeling multitask-based designs. HRM: for modeling hardware platforms.

(iv) *Model-based QoS analysis* with the following features: GQAM: for annotating models subject to quantitative

analysis. SAM: for annotating models subject of scheduling analysis. PAM: for annotating models subject of performance analysis.

For more details on the above acronyms and UML MARTE features, the reader can refer to [89, 90].

Gaspard2 is an Integrated Development Environment (IDE) for SoC (system-on-chip) visual comodeling. It allows or will allow modeling, simulation, testing, and code generation of SoC applications and hardware architectures. Gaspard2 is an autonomous application based on Eclipse [91]. Specification of the Gaspard UML profile can be found in [92]. This profile is an extension of UML semantics to enable the modeling of a SoC, and it is based on MARTE. According to the Gaspard methodology, this is done in three steps: the application, the hardware architecture, and the association of the application to the hardware architecture. The application includes the behavior of the SoC, and it is implemented with a data-flow model, but control flows are allowed via additional mechanisms. The Gaspard profile allows also the modeling of massively parallel and repetitive systems. Gaspard achieves the generation of system components by using model transformations, and it targets VHDL and SystemC code.

Among others, the Gaspard UML profile includes the hardware architecture package, which allows modeling of hardware at a system level. This is of particular interest to the subject of this review paper. For this purpose, the Gaspard UML profile uses the *Hardware Component*. The Hardware Component can be refined using stereotypes to a Processor, a Communication, or a Memory. The Gaspard2 profile enables the modeling of the parallelism and the data dependencies, which are inherent in the application to be designed every time. Application control concepts are modeled in the profile, by allowing the description of synchronous reactive systems [92]. Among the applications which were experimentally designed with the Gaspard2 tool and reported, there is a correlation algorithm and its transformation to the VHDL model, and a radar application, which was implemented as an SoC including 8 PowerPC cores. Moreover, experiments with automatic generation and compilation of SystemC code are reported.

3. The Intermediate Predicate Format

The Intermediate Predicate Format (IPF) (The Intermediate Predicate Format is patented with patent number: 1006354, 15/4/2009, from the Greek Industrial Property Organization.) was invented and designed by the author of this paper as a tool and media for the design encapsulation and the HLS transformations in the CCC (Custom Co-processor Compilation) hardware compilation tool (this hardware compiler method is patented with patent number: 1005308, 5/10/2006, from the Greek Industrial Property Organization). A near-complete analysis of IPF syntax and semantics can be found in [93]. Here, the basic features and declarative semantics of IPF are discussed along with some example IPF data. The IPF file consists of a number of tables (lists) of homogeneous Prolog predicate facts. These facts are logic relations between a number of objects that are sited as a list of positional symbols or reference numbers of other facts of the same, or other tables inside the IPF database. Therefore, the IPF style allows for both declarative (Prolog) and sequential (list-based) processing of the logic facts of IPF by the CCC (Custom Co-processor Compilation) HLS tool transformations in a formal way.

The formal methodology discussed here is motivated by the benefits of using predicate logic to describe the intermediate representations of compilation steps. Another way to use the logic predicate facts of IPF is to use the resolution of a set of transformation Horn clauses [94], as the building blocks of an HLS compiler with formal transformations and a state machine optimization engine. This logic inference engine constitutes the most critical (HLS-oriented back-end) phase of the CCC hardware compiler. The inference engine allows for an efficient implementation of the hardware compilation tasks, such as the mapping of complex data and control structures into equivalent hardware architectures in an optimal way, as well as scheduling and grouping the abstract data operations of the source programs into hardware machine (FSM) control states. The choice of logic predicates turns the HLS process into a formal one. Moreover, there have been a number of other successful uses of the Prolog language in critical areas, for example, in solving the scheduling problem of air-flight control and aircraft landing, expert decision systems, and so forth.

The IPF database is produced by the front-end phase of the CCC (Custom Co-processor Compilation) compiler (see following paragraphs), and it captures the complete algorithmic, structural, and data typing information of the source programs. The source programs model the system's behavior in a number of ADA subroutines. The IPF facts are formal relations between values, types, objects and other programming entities of the source program, as well as other IPF facts. The general syntax of the IPF facts follows the following format:

$$\text{fact\_id}(\text{object1}, \text{object2}, \ldots, \text{object}N) \tag{5}$$

The predicate name fact_id relates in this fact the objects object1 to object$N$ in a logical way. IPF facts represent an algorithmic feature such as a program operation, a data object description, a data type, an operator, and a subprogram call. An example of a plus (+) operation which is an addition is described by the following, program table fact:

$$\text{prog\_stmt}(\text{“subprogram2”}, 3,0,63,3,9,10,5) \tag{6}$$

The predicate fact prog_stmt of (6) describes an addition (operator reference = 63), which is the 3rd operation of the ADA subprogram subprogram2. This operation adds two operands with reference numbers 3 and 9 and produces a result on a variable with reference number 10. These operands and result data descriptions are part of the data table

and for this example they are given in the following examples ((7), (8), and (9)):

data_stmt("subprogram2", "*dx*", 3, 2, "var", sym("node")) (7)

...

data_stmt("subprogram2", "*dy*", 9, 2, "var", sym("node")) (8)

data_stmt("subprogram2", "*xc*", 10, 2,"var", sym("node")). (9)

In the above data table facts, we see their reference numbers 3, 9, and 10 which are used in the program table fact of (6), and their names (variable ids) are *dx*, *dy*, and *xc*, respectively. Apart from their host subprogram, these facts describe variables (see the "var" object) and they are of type 2 (integer). This type is described in the type table of the same IPF database with the type fact as it is written bellow:

type_def(2,"integer", 32, "standard", 0, "single_t", 0, 0, 0) (10)

The type definition of the integer type is given in (10), and there are various objects related under this type fact such as the kind of this type ("single_t" which means with no components or with a single component), the name ("integer") of the type and its size in bits (32). A complete description of the IPF fact objects and structure is not the purpose of this work. Nevertheless, a near-complete description can be found in [93].

From the above, it can be concluded that the IPF facts are logical relations of objects, and this formal description is used by the back-end phase of the CCC compiler in order to implement the HLS transformations of the tool. The CCC HLS transformations use the IPF facts of the source design, along with other logical rules in order to "conclude" and generate the RTL hardware models at the output of the CCC compiler. The generated RTL models (coded in VHDL) can be synthesized, in turn, by commercial and research RTL synthesizers, into technology gate-netlists, along with other, technology-specific and EDA vendor-specific ECAD back-end tools, in order to produce the final implementations of the described system.

It can be derived from the above forms (IPF excerpts) that IPF is suitable for declarative processing which is done by Prolog code, as well as linear, sequential processing. The latter is enabled from the way that some IPF facts (e.g., data table facts) are referenced with their linear entry numbers in other IPF facts (e.g., program table facts). In this way, lists can be built and processed by prolog predicates that utilize list editing and with recursive processing so as to easily implement the generation and processing of lists, for example, the initial schedule (state list) which includes the FSM states and their components (e.g., operations and conditions-guards). Also, both these types of processing in Prolog as well as the nature of the IPF predicate facts are formal, which makes the whole transformation from the source code down to the generated FSM-controlled hardware, a provably correct one. This eliminates the need to reverify the functionality of the generated hardware with traditional and lengthy RTL simulations, since due to the formal nature of the transformations the generated hardware is correct-by-construction. Therefore, valuable development time is saved in order to take important decisions about the high-level architecture template options such as the positioning of large data objects such as arrays on embedded scratch pad or external shared memories.

4. Hardware Compilation Flow

The front-end compiler translates the algorithmic data of the source programs into the IPF's logic statements (logic facts). The inference logic rules of the back-end compiler transform the IPF facts into the hardware implementations. There is a one-to-one correspondence between the source specification's subroutines and the generated hardware implementations. The source code subroutines can be hierarchical, and this hierarchy is maintained in the generated hardware implementation. Each generated hardware model is a FSM-controlled custom processor (or coprocessor, or accelerator), that executes a specific task, described in the source program code. This hardware synthesis flow is depicted in Figure 1.

Essentially, the front-end compilation resembles software compilation and the back-end compilation executes formal transformation tasks that are normally found in HLS tools. This whole compilation flow is a formal transformation process, which converts the source code programs into implementable RTL (Register-Transfer Level) VHDL hardware accelerator models. If there are function calls in the specification code, then each subprogram call is transformed into an interface event in the generated hardware FSM. The interface event is used so that the "calling" accelerator uses the "services" of the "called" accelerator, as it can be thought in the source code hierarchy.

5. Back-End Compiler Inference Logic Rules

The back-end compiler consists of a very large number of logic rules. The back-end compiler logic rules are coded with logic programming techniques, which are used to implement the HLS algorithms of the back-end compilation phase. As an example, one of the latter algorithms reads and incorporates the IPF tables' facts into the compiler's internal inference engine of logic predicates and rules [94]. The back-end compiler rules are given as a great number of definite clauses of the following form:

$$A_0 \longleftarrow A_1 \wedge \cdots \wedge A_n \text{ (where } n \geq 0), \tag{11}$$

where $\leftarrow$ is the logical implication symbol ($A \leftarrow B$ means that if B applies then A applies), and $A_0, \ldots, A_n$ are atomic formulas (logic facts) of the form:

$$\text{predicate\_symbol}(\text{Var\_1}, \text{Var\_2}, \ldots, \text{Var\_}N), \tag{12}$$

where the positional parameters Var_1,..., Var_N of the above predicate "predicate_symbol" are either variable names

(in the case of the back-end compiler inference rules), or constants (in the case of the IPF table statements) [93, 94]. The predicate syntax in (12) is typical of the way of the IPF facts and other facts interact with each other, they are organized and they are used internally in the inference engine. Thus, the hardware descriptions are generated as "conclusions" of the inference engine and in a formal way from the input programs by the back-end phase, which turns the overall transformation into a provably correct compilation process. In essence, the IPF file consists of a number of such atomic formulas, which are grouped in the IPF tables. Each such table contains a list of homogeneous facts which describe a certain aspect of the compiled program. For example all prog_stmt facts for a given subprogram are grouped together in the listing of the program statements table.

6. Inference Logic and Back-End Transformations

The inference engine of the back-end compiler consists of a great number of logic rules (like the one in (11)) which conclude on a number of input logic predicate facts and produce another set of logic facts and so on. Eventually, the inference logic rules produce the logic predicates that encapsulate the writing of RTL VHDL hardware coprocessor models. These hardware models are directly implementable to any hardware (e.g., ASIC or FPGA) technology, since they are technology and platform independent. For example, generated RTL models produced in this way from the prototype compiler were synthesized successfully into hardware implementations using the Synopsys DC Ultra, the Xilinx ISE, and the Mentor Graphics Precision software without the need of any manual alterations of the produced RTL VHDL code. In what follows, an example of such an inference rule is shown:

$$\begin{aligned} &\text{dont\_schedule(Operation1, Operation2)} \\ &\quad \longleftarrow \text{examine(Operation1, Operation2),} \\ &\qquad \text{predecessor(Operation1, Operation2)} \end{aligned} \tag{13}$$

The meaning of this rule that combines two input logic predicate facts to produce another logic relation (dont_schedule), is that when two operations (Operation1 and Operation2) are examined and the first is a predecessor of the second (in terms of data and control dependencies), then do not schedule them in the same control step. This rule is part of a parallelizing optimizer which is embedded in the inference engine and it aggressively "compresses" the execution time of the generated coprocessor by making a number of operations parallel in the same control step, as soon as of course the data and control dependencies are still satisfied. This parallelizing optimizer is called "PARCS" (meaning: Parallel, Abstract Resource-Constrained Scheduler).

The way that the inference engine rules (predicates relations-productions) work is depicted in Figure 2. The last produced (from its rule) predicate fact is the VHDL RTL writing predicate at the top of the diagram. Right bellow level 0 of predicate production rule there is a rule at the level-1, then level-2 and so on. The first predicates that are fed into

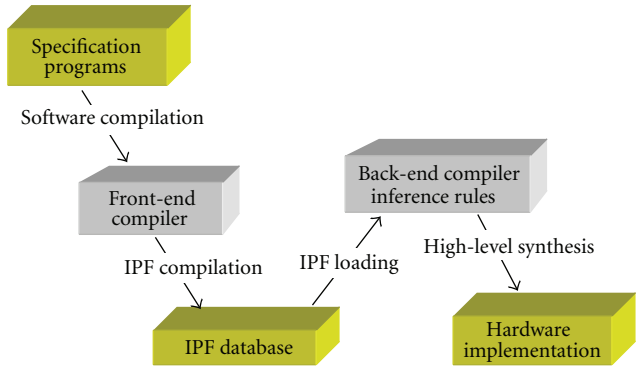


Figure 1: Hardware synthesis flow and tools.

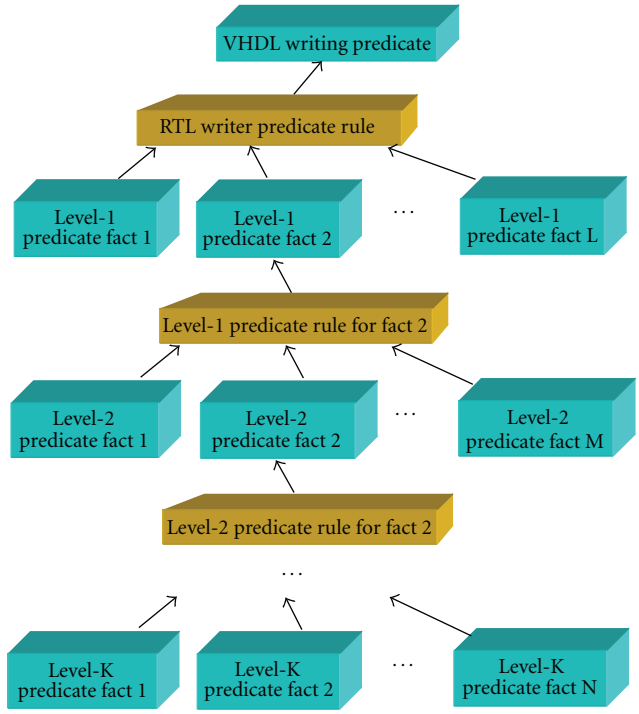


Figure 2: The back-end inference logic rules structure.

this engine of production rules belong to level-K, as shown in this figure. Level-K predicate facts include of course the IPF facts that are loaded into the inference engine along with the other predicates of this level. It is clear from all this that the back-end compiler works with inference logic on the basis of predicate relation rules and, therefore, this process is a formal transformation of the IPF source program definitions into the hardware accelerator (implementable) models. There is no traditional (imperative) software programming involved in this compilation phase, and the whole implementation of the back-end compiler is done using logic programming techniques. Of course in the case of the prototype compiler, there is a very large number of predicates and their relation rules that are defined inside the implementation code of the back-end compiler, but the whole concept of implementing this phase is as shown in Figure 2.

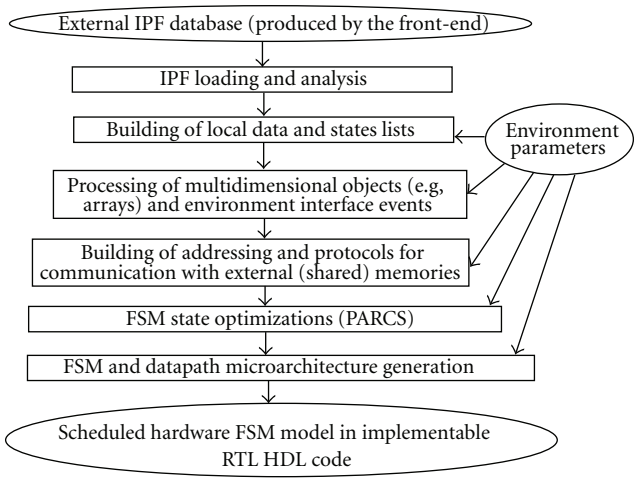


Figure 3: The processing stages of the back-end compiler.

The above fact production rules include the loading of IPF, the generation of list of data objects (virtual registers) and operators (including the special operators of communication with memories and the external computing environment), the initial and the processed list of states, the list of predecessor relations between operations, the PARCS optimizer, the datapath and FSM architecture generators, and so forth. The user of the back-end compiler can select certain environment command list options as well as build an external memory port parameter file as well as drive the compiler's optimizer with specific resource constraints of the available hardware operators.

The above predicate rules implement the various stages of the back-end compiler. The most important of these stages can be seen in Figure 3. The compilation process starts with the loading of the IPF facts into the inference rule engine. After the IPF database is analyzed, the local data object and operation and initial state lists are built. Then, the environment options are read and the temporary lists are updated with the special (communication) operations as well as the predecessor and successor dependency relation lists. After the complete initial schedule is built and concluded, the PARCS optimizer is run and the optimized schedule is delivered to the microarchitecture generator. The transformation is concluded with the formation of the FSM and datapath implementation and the writing of the RTL VHDL model for each accelerator that is defined in each subprogram of the source code program.

As already mentioned, from each subprogram which is coded in the specification program, a separate hardware accelerator model is generated. All these hardware models are directly implementable into hardware using commercial CAD tools, such as the Synopsys DC-ultra, the Xilinx ISE, and the Mentor Graphics Precision RTL synthesizers. Also, the hierarchy of the source program modules (subprograms) is maintained and the generated accelerators may be hierarchical. This means that an accelerator can invoke the services of another accelerator from within its processing states, and that other accelerator may use the services of yet another accelerator, and so on. In this way, a subprogram call in the source code is translated into an external coprocessor interface event of the corresponding hardware accelerator.

7. The PARCS Optimizer

The various source program operations are scheduled using the optimizing algorithm PARCS. PARCS always attempts to schedule as many as possible operations in the same control step, satisfying of the data and control dependencies and the specific resource (operator) constraints if any, provided by the user.

The pseudocode for the main procedures of the PARCS scheduler is shown in Algorithm 1. The PARCS scheduler consists of a great number of predicate rules. All these are part of the inference engine of the back-end compiler. The utilized predicates are produced with formal logic rules such as the one in (11). A new design to be synthesized is loaded via its IPF into the back-end compiler's inference engine. Hence, the IPF's facts as well as the newly created predicate facts from the so far logic processing, "drive" the logic rules of the back-end compiler which generate provably correct hardware architectures. For example, the following PARCS scheduler rule:

$$\begin{aligned} &\text{process\_state\_ops(Current\_state, Current\_list\_of\_operations,}\\ &\qquad\qquad \text{New\_states\_list\_of\_operations)}\\ &\longleftarrow \text{no\_dependencies(Current\_list\_of\_operations,}\\ &\qquad\qquad\qquad \text{New\_states\_list\_of\_operations),}\\ &\quad \text{absorb(Current\_state, New\_states\_list\_of\_operations)} \end{aligned} \tag{14}$$

is a logic production between three predicates. The meaning is that if predicate no_dependencies is true (in other words if there are not any dependencies between the list of the current state operations and those of the next—under examination—state), then predicate absorb is examined. If in turn absorb is true, which applies when it is executed to absorb the new list of operations into the current state, then process_state_ops is produced, which in turn means that the production rule is executed successfully. For the inference engine execution, this means that another fact of predicate process_state_ops is produced and the PARCS algorithm can continue with another state (if there are any left). The desirable side-effect of this production is that the list of operations of the newly examined state are absorbed into the current PARCS state, and PARCS processing can continue with the remaining states (of the initial schedule).

It is worthy to mention that although the HLS transformations are implemented with logic predicate rules, the PARCS optimizer is very efficient and fast. In most of benchmark cases that were run through the prototype hardware compiler flow, compilation did not exceed 1–10 minutes and the results of the compilation were very efficient as it will be explained bellow in the paragraph of experimental results. This cause and result/effect relation between these two predicates is characteristic of the way the back-end compiler's inference engine rules work. By

(1) start with the initial schedule (including the special external port operations)
(2) Current PARCS state < −1
(3) Get the 1st state and make it the current state
(4) Get the next state
(5) Examine the next state's operations to find out if there are any dependencies with the current state
(6) If there are no dependencies then absorb the next state's operations into the current PARCS state; If there are dependencies then finalize the so far absorbed operations into the current PARCS state, store the current PARCS state, PARCS state < −PARCS state +1; make next state the current state; store the new state's operations into the current PARCS state
(7) If next state is of conditional type (it is enabled by guarding conditions) then call the conditional (true/false branch) processing predicates, else continue
(8) If there are more states to process then go to step 4, otherwise finalize the so far operations of the current PARCS state and terminate

Algorithm 1: Pseudocode of the PARCS scheduling algorithm.

using this type of logic rules the back-end compiler achieves provably correct transformations on the source programs.

8. Generated Hardware Architectures

As mentioned already the hardware architectures that are generated by the back-end synthesizer are platform and RTL tool-independent. This means that generalized micro-architectures are produced, and they can be synthesized into hardware implementations by using any commercial RTL synthesizer without the slightest manual modifications of the generated RTL VHDL code. Furthermore, the generated accelerator architecture models are not dependent to any hardware implementation technology. Therefore, they can be, for example, synthesized into any ASIC or FPGA technology. When they need to be implemented by RTL synthesizers targeting FPGAs, with restricted functional unit and register on-chip resources, then the user of the tools can run the back-end compiler with a resource constraint file. Thus, PARCS will parallelize operations only to the extent that the resource constraints are not violated.

As already mentioned, the back-end stage of microarchitecture generation can be driven by command-line options. One of the options, for example, is to generate massively parallel architectures. The results of this option are shown in Figure 4. This option generates a single process-FSM VHDL description with all the data operations being dependent on different machine states. This means that every operator is enabled by single wire activation commands that are driven by different state register values. This in turn means that there is a redundancy in the generated hardware, in a way that during part of execution time, a number of state-dedicated operators remain idle. However, this redundancy is balanced by the fact that this option achieves the fastest clock cycle, since the state command encoder, as well as the data multiplexers are replaced by single wire commands which do not exhibit any additional delay.

Another microarchitecture option is the generation of traditional FSM + datapath-based VHDL models. The results

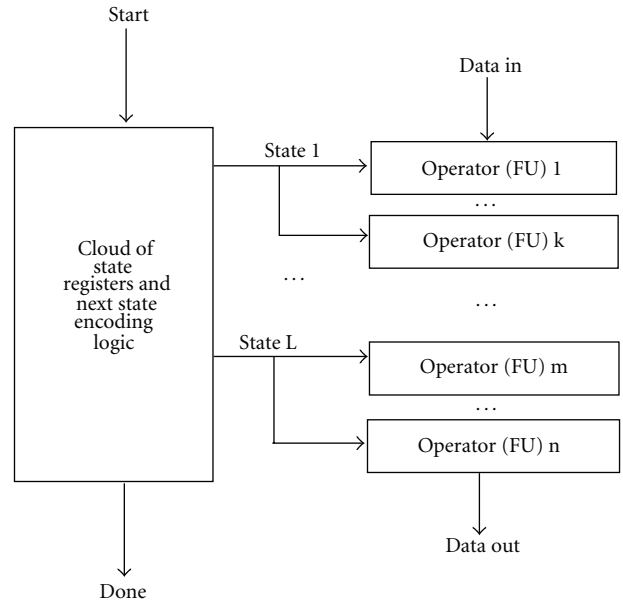


Figure 4: Massively-parallel microarchitecture generation option.

of this option are shown in Figure 5. With this option activated, the generated VHDL models of the hardware accelerators include a next state process as well as signal assignments with multiplexing which correspond to the input data multiplexers of the activated operators. Although this option produces smaller hardware structures (than the massively parallel option), it can exceed the target clock period due to larger delays through the data multiplexers that are used in the datapath of the accelerator.

Using the above microarchitecture options, the user of the inference-based hardware compiler can select various solutions between the fastest and larger massively parallel microarchitecture, which may be suitable for richer technologies in terms of operators such as large ASICs, and smaller and more economic (in terms of available resources) technologies such as smaller FPGAs.

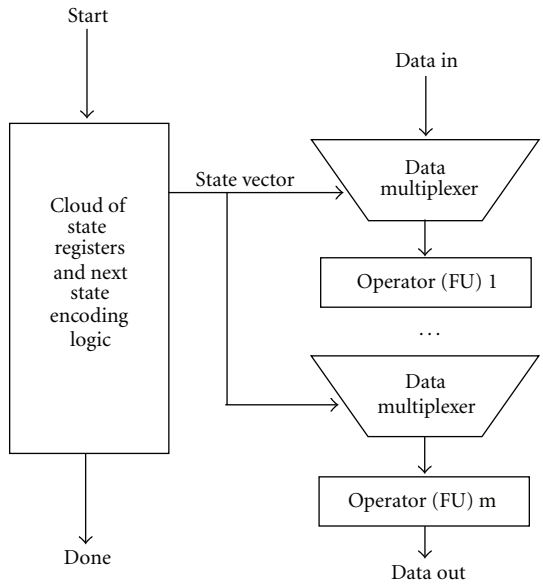


Figure 5: The traditional FSM + datapath generated microarchitecture option.

As it can be seen in Figures 4 and 5, the produced coprocessors (accelerators) are initiated with the input command signal START. Upon receiving this command the coprocessors respond to the controlling environment with the handshake output signal BUSY and right after they start processing the input data in order to produce the results. This process may take a number of clock cycles and it is controlled by a set of states (discrete control steps). When the coprocessors complete their processing, they notify their environment with the output signal DONE. In order to conclude the handshake, the controlling environment (e.g., a controlling central processing unit) responds with the handshake input RESULTS_READ, to notify the accelerator that the processed result data have been read by the environment. This handshake protocol is also followed when one (higher-level) coprocessor calls the services of another (lower-level) coprocessor. The handshake is implemented between any number of accelerators (in pairs) using the START/BUSY and DONE/RESULTS_READ signals. Therefore, the set of executing coprocessors can be also hierarchical in this way.

Other environment options, passed to the back-end compiler, control the way that the data object resources are used, such as registers and memories. Using a memory port configuration file, the user can determine that certain multidimensional data objects, such as arrays and array aggregates, are implemented in external (e.g., central, shared) memories (e.g., system RAM). Otherwise, the default option remains that all data objects are allocated to hardware (e.g., on-chip) registers. Of course the external memory option is more economic in terms of accelerator compilation time and register use, but it causes a longer processing time, due to the communication protocols that are generated, every time that a datum is accessed from/to the external shared memory. Nevertheless, all such memory communication protocols and hardware ports/signals are automatically generated by the back-end synthesizer, and without the need for any manual editing of the RTL code by the user. Both synchronous and asynchronous memory communication protocol generation are supported in the memory port options file.

9. Coprocessor-Based, Hardware/Software Codesign and Coexecution Approach

The generated accelerators can be placed inside the computing environment that they accelerate or can be executed standalone. For every subprogram in the source specification code one coprocessor is generated to speed up (accelerate) the particular system task. The whole system can be modeled in the ADA software code to implement both the software and hardware models of the system. The whole ADA code set can be compiled and executed with the host compiler and linker to run and verify the operation of the whole system at the program code level. This high-level verification is done by compiling the synthesizable ADA along with the system's testbench code and linking it with the host compiler and linker. Then the test vectors can be fed to the system-under-test and the results can be stored in test files, by simply executing the ADA code. In this way, extremely fast verification can be achieved at the algorithmic level. It is evident that such behavioral (high-level) compilation and execution is orders of magnitude faster than conventional RTL simulations. In this way, the hardware/software codesign flow is enabled and mixed systems can be modeled, verified and prototyped in a fraction of the time needed with more conventional approaches.

It is worthy to mention that the whole codesign flow is free of any target architecture templates, platforms or IP-based design formats, which makes it portable, and adaptable to the target system updates and continuous architecture evolution. This is due to the fact that the generated coprocessors as well as their hardware-to-hardware and hardware-to-software interfaces are of generic type and so they can be easily enhanced to plug in the most demanding computing environments, with a minimal level of effort. This is not the case with IP-based design approaches, since the predesigned, and preconfigured IPs are most of the times difficult to adapt to existing target architectures, and their interfaces are often the most constraining obstacles for the integration with the rest of the developed system. Moreover, this paper's codesign approach is free of any target architecture templates and, therefore, free of any core or interface constraints for the integration of the generated custom hardware modules. Therefore, using the general I/O handshake of this approach, extremely complex systems can be delivered in a fraction of time which is needed when using platform or IP based approaches.

After the required coprocessors are specified, coded in ADA, generated with the prototype hardware compiler, and implemented with commercial back-end tools, they can be downloaded into the target computing system (if the target technology consists of FPGAs) and executed to

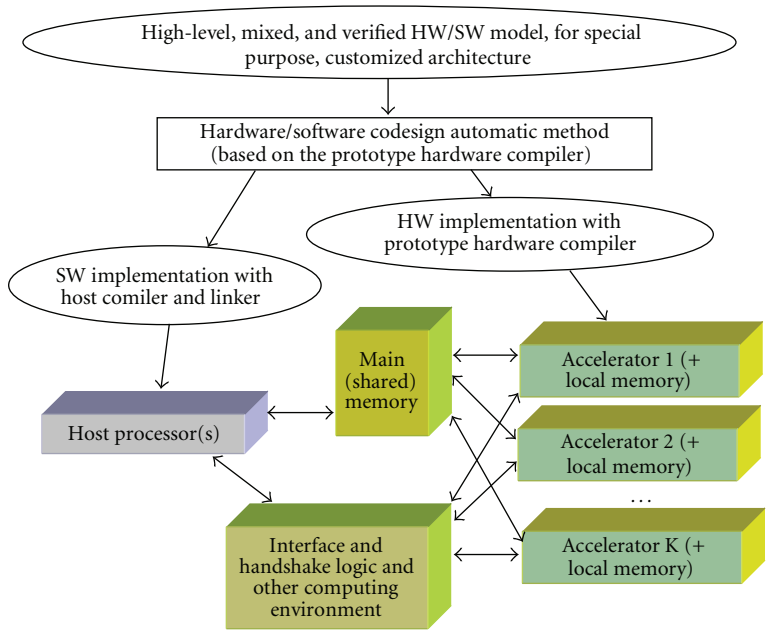


Figure 6: Host computing environment for hardware/software co-design and co-execution.

accelerate certain system tasks. This process is shown in Figure 6. Every accelerator can have both local (register file) and external (system RAM) memory resources. The accelerators can communicate with each other and with the host computing environment using synchronous handshake signals and connections with the system's handshake logic. In case that the host processor features handshake capabilities, the communication can be done directly with the custom accelerator, which in this case operates as a coprocessor with the main processor.

10. Experimental Results and Evaluation of the Method

In order to evaluate the efficiency of the presented HLS and ESL method, five benchmarks from the area of hardware compilation and high-level synthesis were run through the front-end and the back-end compilers. The programs were executed on a Pentium-4 platform running the MS-Windows-XP-SP2 operating system. The five design benchmarks include a DSP FIR filter, a second-order differential equation iterative solver which has long been a well-known high-level synthesis benchmark, an RSA crypto-processor from cryptography applications, a synthetic benchmark that uses two level nested for-loops, and a large MPEG video compression engine. The fourth benchmark includes subroutines with two-dimensional data arrays stored in external memories. These data arrays are processed within the bodies of 2-level nested loops.

All of the above-generated accelerators were simulated and the RTL models matched the input source program's functionality. The state number reduction after applying the PARCS optimizer, on the various modules of the five benchmarks is shown in Table 1. State reduction rate of up to 35% was achieved with the back-end compiler flow. Also, the number of lines of RTL code is orders of magnitude more compared with the lines of the source code model for each submodule. This indicates the gain in engineering performance when the prototype ESL tools are used to automatically implement the computing products. This is due to the time gained by building and verifying (by program execution) fast executable specifications in high-level programs, as opposed to run the system verification only when all the hardware details are fixed and then perform time-consuming detailed (RTL or structural) hardware design and simulations. It is well accepted in the engineering community that the verification time at the algorithmic program level is only a small fraction of the time required for verifying designs at the RTL or the gate-netlist level. This gain is invaluable for developing complex computing systems. It can also be used to eliminate the need for design cycle iterations, later on in the design flow. Such design iterations would be caused by functional errors and specification mismatches, which are very frequent when using conventional implementation methodologies, such as RTL coding or schematic design.

The relative reduction of the number of states in the largest design module (subroutine) of the application, before and after the PARCS parallelizing optimizer, is shown graphically in Figure 7. The reduction of states reaches up to 30 to 40% at some cases which is a significant improvement. Such optimizations are usually very difficult to be done manually, even by experienced ASIC, HDL designers when the number of states exceeds a couple of dozens in the designed application. Noticeably, there were more than 400 states in the initial schedule of the MPEG benchmark. In addition to this, manual coding is extremely prone to errors

Table 1: State reduction statistics from the IKBS PARCS optimizer for the five benchmarks.

Module name	Initial schedule states	PARCS parallelized states	State reduction rate
FIR filter main routine	17	10	41%
Differential equation solver	20	13	35%
RSA main routine	16	11	31%
Nested loops 1st subroutine	28	20	29%
Nested loops 2nd subroutine (with embedded mem)	36	26	28%
Nested loops 2nd subroutine (with external mem)	96	79	18%
Nested loops 3rd subroutine	15	10	33%
Nested loops 4th subroutine	18	12	33%
Nested loops 5th subroutine	17	13	24%
MPEG 1st subroutine	88	56	36%
MPEG 2nd subroutine	88	56	36%
MPEG 3rd subroutine	37	25	32%
MPEG top subroutine (with embeded mem)	326	223	32%
MPEG top subroutine (with external mem)	462	343	26%

Table 2: Area and timing statistics from UMC 65 nm technology implementation.

Area/time statistic	Massively parallel, initial schedule	Massively parallel, PARCS schedule	FSM + datapath, initial schedule	FSM + datapath, PARCS schedule
Area in square nm	117486	114579	111025	107242
Equivalent number of NAND2 gates	91876	89515	86738	83783
Achievable clock period	2 ns	2 ns	2 ns	2 ns
Achievable clock frequency	500 MHz	500 MHz	500 MHz	500 MHz

which are very cumbersome and time consuming to correct with (traditional) RTL simulations and debugging.

The specification (source code) model of the various benchmarks, and all of the designs using the prototype compilation flow, contains unaltered regular ADA program code, without additional semantics, and compilation directives which are usual in other synthesis tools which compile code in SystemC, HandelC, or any other modified program code with additional object class and TLM primitive libraries. This advantage of the presented methodology eliminates the need for the system designers to learn a new language, a new set of program constructs or a new set of custom libraries.

Moreover, the programming constructs and semantics, that the prototype HLS compiler utilizes are the subset which is common to almost all of the imperative and procedural programming languages such as ANSI C, Pascal, Modula, and Basic. Therefore, it is very easy for a user that is familiar with these other imperative languages, to get also familiar with the rich subset of ADA that the prototype hardware compiler processes. It is estimated that this familiarization does not exceed a few days, if not hours for the very experienced software/system programmer/modeler.

Table 2 contains the area and timing statistics of the main module of the MPEG application synthesis runs. Synthesis was executed on a Ubuntu 10.04 LTS linux server with Synopsys DC-Ultra synthesizer and the 65 nm UMC technology libraries. From this table, a reduction in terms of area can be observed for the FSM + datapath implementation against the massively parallel one. Nevertheless, due to the quality of the technology libraries the speed target of 2 ns clock period was achieved in all 4 cases.

Moreover, the area reduction for the FSM + datapath implementations of both the initial schedule and the optimized (by PARCS) one is not dramatic and it reaches to about 6%. This happens because the overhead of massively-parallel operators is balanced by the large amount of data and control multiplexing in the case of the FSM + datapath option.

11. Conclusions and Future Work

This paper includes a discussion and survey of past and present existing ESL HLS tools and related synthesis methodologies. Formal and heuristic techniques for the HLS tasks are discussed and more specific synthesis issues are analyzed. The conclusion from this survey is that the author's prototype ESL behavioral synthesizer is unique in terms of generality of input code constructs, the formal methodologies employed, and the speed and utility of the developed hardware compiler.

The main contribution of this paper is a provably correct, ESL, and high-level hardware synthesis method and a unified prototype tool-chain, which is based on compiler-compiler and logic inference techniques. The prototype tools transform a number of arbitrary input subprograms (at the moment coded in the ADA language) into an equivalent

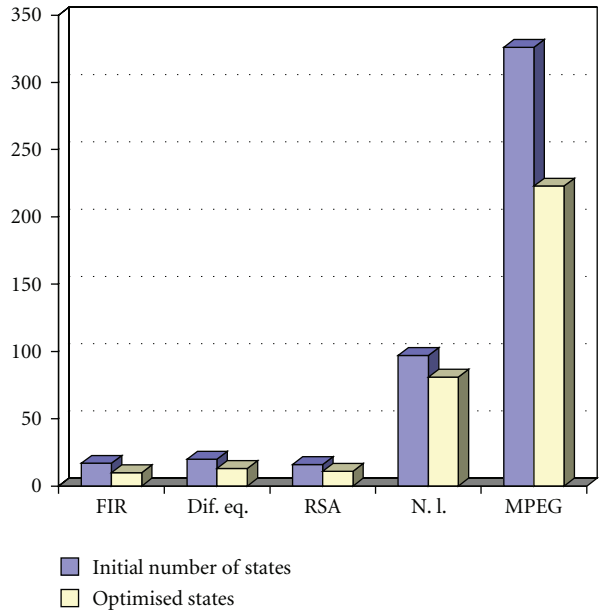


Figure 7: Optimization (reduction) of number of states.

number of functionally equivalent RTL VHDL hardware accelerator descriptions. A very large number of input program applications were run through the hardware compiler, five of which were evaluated in this paper. In all cases, the functionality of the produced hardware implementations matched that of the input subprograms. This was expected due to the formal definition/implementation of the transformations of the hardware compiler, including the intermediate IPF form and the inference rules of the back-end phase. Encouraging state-reduction rates of the PARCS scheduler optimizer were observed for five benchmarks in this paper, which exceed 30% in some cases. Using its formal flow, the prototype hardware compiler can be used to develop complex systems in orders of magnitude shorter time and lower engineering effort, than that which are usually required using conventional design approaches such as RTL coding or IP encapsulation and schematic entry using custom libraries.

Existing HLS tools compile usually a small subset of the programming language, and sometimes with severe restrictions in the type of constructs they accept (some of them do not accept while-loops for example). Also, most of them produce better results on linear, data-flow-oriented specifications. However, a large number of applications found in embedded and telecommunication systems, mobile, and other portable computing platforms involve a great deal of complex control flow with nesting and hierarchy levels. For this kind of applications, a lot of HLS tools produce low level of quality results. The prototype ESL tool developed by the author has proved that it can deliver a better quality of results in applications with complex control such as image compression and processing standards. Moreover, a more general class of input code constructs are accepted by the front-end compiler and, therefore, a wider range of applications can be prototyped using the CCC HLS tool.

Moreover, using the executable ADA (and soon ANSI-C) models (executable specifications) for the designed system, the user of this hardware/software codesign methodology, can easily evolve the formally verified and synthesized, mixed architecture of the product or application that he intends to develop, and in a fraction of time that this is done with platform or IP based, or other conventional design methodologies. This is due to the core functionality and interface constraints that are introduced when predesigned architecture templates or IP blocks are involved. Moreover, due to the latter problem, large gaps in the intended-to-resulted system performance are usually observed, which often constitutes even a development halt for commercial (e.g., embedded) and other computing products.

Future extensions of this work include undergoing work to upgrade the front-end phase to accommodate more input programming languages (e.g., ANSI-C, C++) and the back-end HDL writer to include more back-end RTL languages (e.g., Verilog HDL), which are currently under development. Another extension could be the inclusion of more than 2 operand operations as well as multicycle arithmetic unit modules, such as multicycle operators, to be used in datapath pipelining. Moreover, there is ongoing work to extend the IPF's semantics so that it can accommodate embedding of IP blocks (such as floating-point units) into the compilation flow and enhance further the schedule optimizer algorithm for even more reduced schedules. Also, other compiler phase validation techniques based on formal semantic such as RDF and XML flows are investigated and are currently under development. Furthermore, connection flows from the front-end compiler to even more front-end diagrammatic system modeling formats such as the UML formulation are currently investigated.

References

[1] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1098–1112, 1986.

[2] E. F. Girczyc, R. J. A. Buhr, and J. P. Knight, "Applicability of a subset of Ada as an algorithmic hardware description language for graph-based hardware compilation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 2, pp. 134–142, 1985.

[3] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[4] I. C. Park and C. M. Kyung, "Fast and near optimal scheduling in automatic data path synthesis," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 680–685, June 1991.

[5] N. Park and A. C. Parker, "Sehwa: a software package for synthesis of pipelined data path from behavioral specification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 3, pp. 356–370, 1988.

[6] E. F. Girczyc, "Loop winding—a data flow approach to functional pipelining," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 382–385, 1987.

[7] C. J. Tseng and D. P. Siewiorek, "Automatic synthesis of data path on digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[8] F. J. Kurdahi and A. C. Parker, "REAL: a program for register allocation," in *Proceedings of Design Automation Conference (DAC '87)*, pp. 210–215, Miami Beach, Fla, USA, 1987.

[9] C. Y. Huang, Y. S. Chen, Y. L. Lin, and Y. C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 499–504, June 1990.

[10] F. S. Tsay and Y. C. Hsu, "Data path construction and refinement. Digest of Technical papers," in *Proceedings of the The International Conference on Computer-Aided Design (ICCAD '90)*, pp. 308–311, Santa Clara, Calif, USA, 1990.

[11] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioral descriptions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 2, pp. 171–180, 1989.

[12] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*, MIT Press, Cambridge, Mass, USA, 1984.

[13] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The ISPS computer description language," Tech. Rep. CMU-CS-79-137, School of Computer Science, Carnegie Mellon University, 1979.

[14] P. Marwedel, "The MIMOLA design system: tools for the design of digital processors," in *Proceedings of the 21st Design Automation Conference*, pp. 587–593, 1984.

[15] A. E. Casavant, M. A. d'Abreu, M. Dragomirecky et al., "Synthesis environment for designing DSP systems," *IEEE Design and Test of Computers*, vol. 6, no. 2, pp. 35–44, 1989.

[16] P. G. Paulin and J. P. Knight, "Algorithms for high-level synthesis," *IEEE Design and Test of Computers*, vol. 6, no. 6, pp. 18–31, 1989.

[17] D. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis," *IEEE Design and Test of Computers*, vol. 11, no. 4, pp. 44–54, 1994.

[18] R. A. Walker and S. Chaudhuri, "Introduction to the scheduling problem," *IEEE Design and Test of Computers*, vol. 12, no. 2, pp. 60–69, 1995.

[19] V. Berstis, "V compiler: automating hardware design," *IEEE Design and Test of Computers*, vol. 6, no. 2, pp. 8–17, 1989.

[20] J. A. Fisher, "Trace Scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, 1981.

[21] A. Kuehlmann and R. A. Bergamaschi, "Timing analysis in high-level synthesis," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '92)*, pp. 349–354, November 1992.

[22] T. C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji, "ILP solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis," *VLSI Design*, vol. 3, no. 1, pp. 21–36, 1995.

[23] C. A. Papachristou and H. Konuk, "A linear program driven scheduling and allocation method followed by an interconnect optimization algorithm," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 77–83, June 1990.

[24] J. Biesenack, M. Koster, A. Langmaier et al., "Siemens high-level synthesis system CALLAS," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 3, pp. 244–252, 1993.

[25] The Electronic Design Interchange Format, Wikipedia, the free encyclopedia, 2011, http://en.wikipedia.org/wiki/EDIF.

[26] M. S. Rubin, "Computer Aids for VLSI Design. Appendix D: Electronic Design Interchange Format," 2011, http://www.rulabinsky.com/cavd/text/chapd.html.

[27] T. Filkorn, "A method for symbolic verification of synchronous circuits," in *Proceedings of the Computer Hardware Description Languages and their Applications (CHDL '91)*, pp. 229–239, Marseille, France, 1991.

[28] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 16–28, 1993.

[29] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design and Test of Computers*, vol. 10, no. 4, pp. 64–75, 1993.

[30] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The Olympus synthesis system," *IEEE Design and Test of Computers*, vol. 7, no. 5, pp. 37–53, 1990.

[31] A. Jerraya, I. Park, and K. O'Brien, "AMICAL: an interactive high level synthesis environment," in *Proceedings of the 4th European Conference on Design Automation with the European Event in ASIC*, pp. 58–62, 1993.

[32] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 6–15, 1993.

[33] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1985.

[34] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design and Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[35] I. Bolsens, H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest, "Hardware/software co-design of digital telecommunication systems," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 391–417, 1997.

[36] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. De Man, "DSP specification using the Silage language," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pp. 1057–1060, April 1990.

[37] P. Willekens, "Algorithm specification in DSP station using data flow language," *DSP Applications*, vol. 3, no. 1, pp. 8–16, 1994.

[38] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[39] M. Van Canneyt, "Specification, simulation and implementation of a GSM speech codec with DSP station," *DSP and Multimedia Technology*, vol. 3, no. 5, pp. 6–15, 1994.

[40] J. T. Buck, "PTOLEMY: a framework for simulating and prototyping heterogeneous systems," *International Journal in Computer Simulation*, pp. 1–34, 1992.

[41] R. Lauwereins, M. Engels, M. Ade, and J. A. Peperstraete, "Grape-II: a system-level prototyping environment for DSP applications," *Computer*, vol. 28, no. 2, pp. 35–43, 1995.

[42] M. S. Rafie, "Rapid design and prototyping of a direct sequence spread-spectrum ASIC over a wireless link," *DSP and Multimedia Technology*, vol. 3, no. 6, pp. 6–12, 1994.

[43] L. Séméria, K. Sato, and G. De Micheli, "Synthesis of hardware models in C with pointers and complex data structures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 743–756, 2001.

[44] R. P. Wilson, "Suif: an infrastructure for research on parallelizing and optimizing compilers," *ACM SIPLAN Notices*, vol. 28, no. 9, pp. 67–70, 1994.

[45] A. A. Kountouris and C. Wolinski, "Efficient scheduling of conditional behaviors for high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 3, pp. 380–412, 2002.

[46] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 441–470, 2004.

[47] W. Wang and T. K. Tan, "A comprehensive high-level synthesis system for control-flow intensive behaviors," in *Proceedings of the 13th ACM Great Lakes symppsium on VLSI (GLSVLSI '03)*, pp. 11–14, 2003.

[48] Z. P. Gu, J. Wang, R. P. Dick, and H. Zhou, "Incremental exploration of the combined physical and behavioral design space," in *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pp. 208–213, 2005.

[49] L. Zhong and N. K. Jha, "Interconnect-aware high-level synthesis for low power," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '02)*, pp. 110–117, November 2002.

[50] C. 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*, vol. 15, no. 11, pp. 1191–1203, 2007.

[51] K. Wakabayashi, "C-based synthesis experiences with a behavior synthesizer, "Cyber"," in *Proceedings of the Design, Automation and Test in Europe*, pp. 390–393, 1999.

[52] W. Wang, A. Raghunathan, N. K. Jha, and S. Dey, "High-level synthesis of multi-process behavioral descriptions," in *Proceedings of the 16th IEEE International Conference on VLSI Design (VLSI '03)*, pp. 467–473, 2003.

[53] B. Le Gal, E. Casseau, and S. Huet, "Dynamic memory access management for high-performance DSP applications using high-level synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1454–1464, 2008.

[54] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors," in *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 112–115, June 1992.

[55] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits," *IEE Proceedings*, vol. 150, no. 5, pp. 330–337, 2003.

[56] E. Martin, O. Santieys, and J. Philippe, "GAUT, an architecture synthesis tool for dedicated signal processors," in *Proceedings of the IEEE International European Design Automation Conference (Euro-DAC '93)*, pp. 14–19, Hamburg, Germany, 1993.

[57] M. C. Molina, R. Ruiz-Sautua, P. García-Repetto, and R. Hermida, "Frequent-pattern-guided multilevel decomposition of behavioral specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 60–73, 2009.

[58] K. Avnit, V. D'Silva, A. Sowmya, S. Ramesh, and S. Parameswaran, "Provably correct on-chip communication: a formal approach to automatic protocol converter synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 2, article 19, 2009.

[59] J. Keinert, M. Streubuhr, T. Schlichter et al., "SystemCoDesigneran—automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, article 1, 2009.

[60] J. Falk, C. Haubelt, and J. Teich, "Efficient representation and simulation of model-based designs in SystemC," in *Proceedings of the Forum on Design Languages*, pp. 129–134, Darmstadt, Germany, 2006.

[61] S. Kundu, S. Lerner, and R. K. Gupta, "Translation validation of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 4, pp. 566–579, 2010.

[62] S. Paik, I. Shin, T. Kim, and Y. Shin, "HLS-1: a high-level synthesis framework for latch-based architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 5, pp. 657–670, 2010.

[63] M. Rim and R. Jain, "Lower-bound performance estimation for high-level synthesis scheduling problem," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 451–458, 1994.

[64] S. Chaudhuri and R. A. Walker, "Computing lower bounds on functional units before scheduling," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 2, pp. 273–279, 1996.

[65] H. Mecha, M. Fernandez, F. Tirado, J. Septien, D. Mozos, and K. Olcoz, "Method for area estimation of data-path in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 2, pp. 258–265, 1996.

[66] Y. M. Fang and D. F. Wong, "Simultaneous functional-unit binding and floorplanning. Digest of Technical Papers," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '94)*, pp. 317–321, Santa Clara, Calif, USA, 1994.

[67] M. Munch, N. Wehn, and M. Glesner, "Optimum simultaneous placement and binding for bit-slice architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '95)*, pp. 735–742, September 1995.

[68] A. Ahmadi and M. Zwolinski, "Multiple-width bus partitioning approach to datapath synthesis," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 2994–2997, May 2007.

[69] R. Gupta and M. A. Breuer, "Partial scan design of register-transfer level circuits," *Journal of Electronic Testing*, vol. 7, no. 1-2, pp. 25–46, 1995.

[70] M. L. Flottes, D. Hammad, and B. Rouzeyre, "High level synthesis for easy testability," in *Proceedings of the European Design and Test Conference*, pp. 198–206, Paris, France, 1995.

[71] M. K. Dhodhi, F. H. Hielscher, R. H. Storer, and J. Bhasker, "Datapath synthesis using a problem-space genetic algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 934–944, 1995.

[72] A. Mujumdar, R. Jain, and K. Saluja, "Incorporating testability considerations in high-level synthesis," *Journal of Electronic Testing*, vol. 5, no. 1, pp. 43–55, 1994.

[73] M. Potkonjak, S. Dey, and R. K. Roy, "Synthesis-for-testability using transformations," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '95)*, pp. 485–490, September 1995.

[74] F. F. Hsu, E. M. Rudnick, and J. H. Patel, "Enhancing high-level control-flow for improved testability. Digest of Technical Papers," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '96)*, pp. 322–328, San Jose, Calif, USA, 1996.

[75] A. Raghunathan, S. Dey, and N. K. Jha, "Register-transfer level estimation techniques for switching activity and power consumption," in *Proceedings of the IEEE/ACM International*

Conference on Computer-Aided Design, pp. 158–165, November 1996.

[76] J. Rabaey, L. Guerra, and R. Mehra, "Design guidance in the power dimension," in *Proceedings of the 1995 20th International Conference on Acoustics, Speech, and Signal Processing*, pp. 2837–2840, May 1995.

[77] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 166–172, November 1996.

[78] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 318–322, October 1994.

[79] L. Goodby, A. Orailoglu, and P. M. Chau, "Microarchitectural synthesis of performance-constrained, low-power VLSI designs," in *Proceedings of the IEEE International Conference on Computer Design*, pp. 323–326, October 1994.

[80] E. Musoll and J. Cortadella, "Scheduling and resource binding for low power," in *Proceedings of the 8th International Symposium on System Synthesis*, pp. 104–109, September 1995.

[81] N. Kumar, S. Katkoori, L. Rader, and R. Vemuri, "Profile-driven behavioral synthesis for low-power VLSI systems," *IEEE Design and Test of Computers*, vol. 12, no. 3, pp. 70–84, 1995.

[82] R. San Martin and J. P. Knight, "Power-profiler: optimizing ASICs power consumption at the behavioral level," in *Proceedings of the 32nd Design Automation Conference*, pp. 42–47, June 1995.

[83] C. Trabelsi, R. Ben Atitallah, S. Meftali, J. L. Dekeyser, and A. Jemai, "A model-driven approach for hybrid power estimation in embedded systems design," *Eurasip Journal on Embedded Systems*, vol. 2011, Article ID 569031, 2011.

[84] I. Issenin, E. Brockmeyer, B. Durinck, and N. D. Dutt, "Data-reuse-driven energy-aware cosynthesis of scratch pad memory and hierarchical bus-based communication architecture for multiprocessor streaming applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1439–1452, 2008.

[85] D. S. Rao and F. J. Kurdahi, "Controller and datapath trade-offs in hierarchical RT-level synthesis," in *Proceedings of the 7th International Symposium on High-Level Synthesis*, pp. 152–157, May 1994.

[86] S. C. Y. Huang and W. H. Wolf, "How datapath allocation affects controller delay," in *Proceedings of the 7th International Symposium on High-Level Synthesis*, pp. 158–163, May 1994.

[87] UML 2.0, 2012, http://www.omg.org/spec/UML/2.0/.

[88] Model Transformation Framework, 2012, http://www.alphaworks.ibm.com/tech/mtf.

[89] UML MARTE, 2012, http://www.omgwiki.org/marte/.

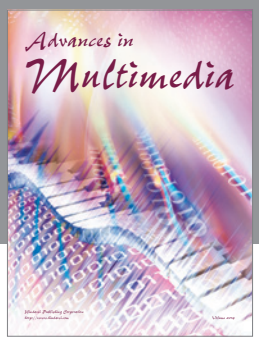
[90] MARTE specification version 1.0., 2012, http://www.omg.org/spec/MARTE/1.0/PDF/.

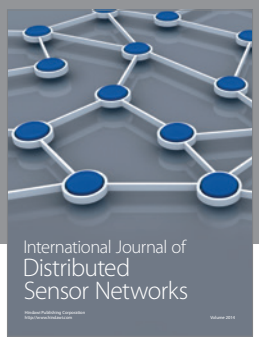
[91] The Eclipse Software, 2012, http://en.wikipedia.org/wiki/Eclipse_(software).

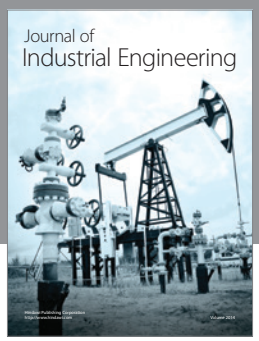
[92] Gaspard2 UML profile documentation, 2012, http://www.lifl.fr/west/gaspard/gaspard-profile.pdf.

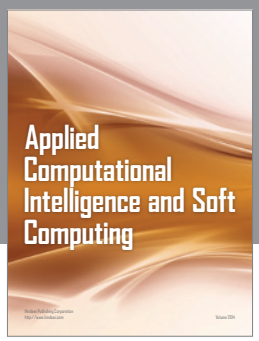
[93] M. Dossis, "Intermediate predicate format for design automation tools," *Journal of Next Generation Information Technology*, vol. 1, no. 1, pp. 100–117, 2010.

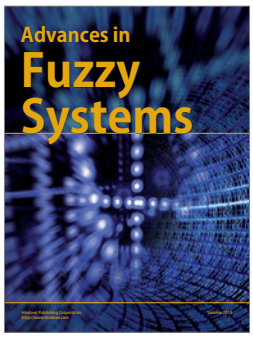
[94] U. Nilsson and J. Maluszynski, *Logic Programming and Prolog*, John Wiley & Sons, 2nd edition, 1995.

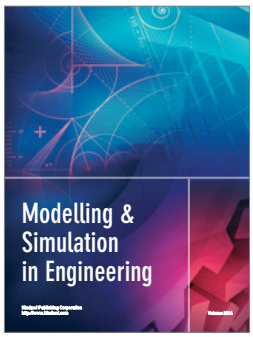


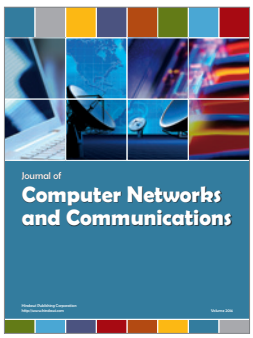












Hindawi
Submit your manuscripts at
http://www.hindawi.com