High-level Synthesis and Optimization Strategies in Hercules and Hebe

David Ku Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

In this paper, we present an approach to automated synthesis of digital circuits from behavioral specifications. The system, called *Hercules* and *Hebe*, offers many advantages to the designer. First, the system supports constraint-driven synthesis where timing and resource constraints are applied to guide the synthesis decisions. Second, systematic design space exploration is possible, where the designer explores the tradeoff between area and performance to meet the design objectives. Third, logic synthesis techniques are uniformly incorporated within the synthesis framework to provide estimates to guide high-level decisions. Along with a synthesis oriented hardware description language called *HardwareC*, *Hercules/Hebe* provides an environment for the design of general synchronous digital circuits, with specific attention to the requirements of ASIC designs. The system has been applied to complex ASIC designs, including the the Digital Audio I/O, MAMA, and Bi-Dimensional DCT chips.

1 Introduction

Computer-aided synthesis of digital circuits from behavioral specification can greatly improve the complexity and quality of hardware design. By providing a synthesis framework of tools, a circuit can be specified by its functional description, timing requirements and interface constraints. The description can then be optimized by the high-level and logic synthesis systems to produce a logic level implementation that satisfies the user constraints. The benefits of a high-level synthesis-based design methodology include standardization, self-documentation and ease of modification of the hardware specifications, shortened design time, increased productivity of the designers, and better quality of the synthesized design due to the use of optimization techniques at the functional level.

While logic synthesis techniques have been established as standard steps in the design methodology for digital circuits [1], high-level synthesis techniques have been lagging behind for several reasons. First, most proposed and used hardware description languages (HDLs) have been conceived for hardware simulation and documentation. For a circuit description to be effective for synthesis, it is necessary to have a well-defined hardware synthesis semantic for the language. A consensus on a behavioral HDL platform for synthesis has not been reached yet. Second, given the diversity of the approaches to digital circuit designs, it is difficult to encode all implementation decisions in terms of algorithms or rules that can be universally applied. Therefore, practical high-level synthesis techniques need to support *automatic* and *user-driven* synthesis modes to leverage off the designer's knowledge and experience. Third, as designs increase in size and complexity, system integration issues often dominate a design. Hardware interfacing and design constraints on timing and area need to be addressed at both the design specification and design synthesis levels.

Previous work on high-level synthesis system addressed primarily microprocessor and digital signal processing designs [2], [3], [4], [5], [6], [7], [8], [9], [10], and [14]. We believe that high-level synthesis can be especially effective in the synthesis of Application Specific Integrated Circuits (ASICs). For this reason, we have developed a hardware description language and a high-level synthesis system for the synthesis and simulation of general-purpose digital circuits, with specific attention to the requirements of ASIC designs. The system is divided into two parts, called *Hercules* and *Hebe*, and has the following features:

- *Uses synthesis-oriented HDL. HardwareC* is a hardware description language designed for synthesis of digital circuits [12]. It supports hardware descriptions with both procedural and declarative semantics, as well as constraints on the hardware implementation. *HardwareC* also provides constructs to support the description of libraries of generic hardware resources.
- *Supports constraint-driven synthesis with partial binding*. Timing and resource requirements that are specified in the high-level description are used to guide the results of the synthesis optimizations. The system supports the synthesis of partially bound hardware descriptions, where certain operations are pre-bound to specific hardware modules, and performs synthesis on the remaining operations. It provides a flexible framework for systematic exploration of the design space of tradeoffs between area and performance, and guarantees *optimal* scheduling of each point in the design space.
- *Incorporates logic synthesis techniques*. To meet the area requirements, resource sharing is a necessary part of the synthesis system. Since resources correspond to models that are described and invoked in the high level description, the characterization of resources to evaluate sharing feasibility is carried out using logic synthesis techniques to provide estimates on timing and area. This methodology is particularly suited for ASIC designs that tend to rely on application-specific logic functions. The use of logic synthesis for estimates improves the quality of the synthesized designs, and avoids erroneous high-level decisions due to insufficient data or inappropriate assumptions.

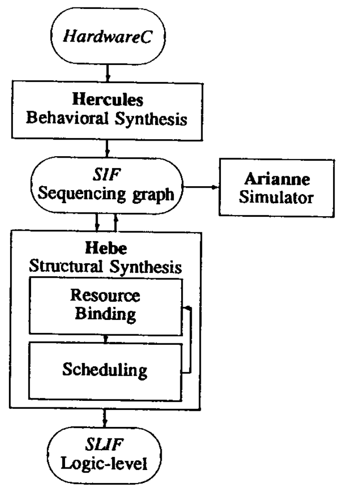

Figure 1: Block diagram of the *Hercules* and *Hebe* system.

- *Supports automatic and user-driven synthesis.* The synthesis flow can be fully automated, transforming an input *HardwareC* description directly to a logic-level implementation. The system also supports user-driven synthesis, where a designer can intervene and drive high-level decisions based on an evaluation of the possible design tradeoffs.

We have implemented two programs for the high-level synthesis of synchronous digital circuits, called **Hercules** [11] and **Hebe**. The two programs transform a behavioral description of hardware in *HardwareC*, through a series of translations and optimizations, to a synchronous logic implementation that satisfies the timing and resource constraints that are imposed on the design. *Hercules* performs the front-end parsing and behavioral optimizations. It generates an implementation-independent description of the hardware behavior in a graph-based representation, called the *Sequencing Intermediate Form* (SIF). The SIF can be simulated by *Arianne* to provide feedback to the designer on the functional correctness of the input description. *Hebe* maps the SIF into a logic-level implementation, described in the *Structural/Logic Intermediate Form* (SLIF). A block diagram of the system is shown in Figure 1.

2 Hardware Modeling

HardwareC is a high-level hardware description language with a C-like syntax. The language has its own hardware semantics, and it differs from the C programming language in many respects. *HardwareC* supports both declarative semantic (e.g. interconnection of modules) and procedural semantic (e.g. set of operations ordered in time) in the modeling of hardware. There are four fundamental design abstractions, corresponding to *block*, *process*, *procedure*, and *function models*. At the topmost level, a design is described in terms of a block, which contains an interconnection of logic and instances of other blocks and processes. A process consists of a hierarchy of procedures and functions, and represents a functionality that executes repeatedly, restarting itself upon completion. Since a process executes concurrently and independently with respect to the other processes in the system, it allows the modeling of *coarse-grain* parallelism at the functional level. A procedure or function is an encapsulation of operations, and may contain calls to other procedures and functions.

In contrast to micro-architectural synthesis systems that use a predefined set of library elements as building blocks, *Hercules* and *Hebe* treat each *model* in the input description as a *resource* that can be allocated and shared among the calls to the models (either procedures or functions). For example, two calls to a model `A` can be implemented either by a single resource corresponding to the hardware implementation of `A`, where both calls share the use of the resource; or by two resources, where each call is implemented by a different resource. Operators such as $+$ or $-$ can either be converted into calls to the appropriate library models, or by default be implemented in terms of logic expressions.

HardwareC supports the usual iterative and branching constructs, including both fixed-iteration and data-dependent looping constructs. Data-dependent loops can be used to detect signal transitions, which are important in describing external interfaces. For example, the construct `while (data==0);` will wait until the rising transition of the signal `data`. In addition, there are several features of *HardwareC* that support hardware specification and synthesis:

- *Interprocess communication* – To support communication and synchronization among the concurrent processes, *HardwareC* supports both *parameter passing* and *message passing*. The former assumes the existence of a shared medium (e.g. shared bus or memory) that interconnects the hardware modules implementing processes. The handshaking protocols are described in the *HardwareC* description. The latter uses a synchronous *send/receive* mechanism that can be used for synchronization or data transfer. The corresponding hardware for communication, as well as its protocol, are automatically synthesized.

- *Explicit instantiation of models* – Hierarchical designs are supported through the use of model calls. A call to a model can be either *generic* or *instantiated*: a generic call invokes a model without specifying the particular instance that is used to implement the call, whereas an instantiated call identifies also a specific instance of the model which will implement the call. Through explicit instantiation of model calls, *HardwareC* supports resource constraints and partial bindings of operations to resources. The designer can constrain the synthesis system to explore a subset of the possible structures corresponding to a behavioral model to satisfy a particular architectural requirement.

- *Template models* – Templates are models that take one or more integer arguments, and support *polymorphism* in the language by modeling several behaviors with a single description. As an example, a single template can be used to describe a family of adders of different size. Templates are therefore very useful in describing libraries of hardware operators at a high level.

- *Degree of parallelism* – For procedural semantic models, *HardwareC* offers the designer the ability to adjust the degree of parallelism in a given design through the use of *sequential*, *data-parallel*, or *parallel* groupings of operations. In the first case, operations are executed sequentially. In the second one, all operations are executed in parallel, unless data dependency requires serialization. In the last case, all operations execute in parallel unconditionally.

```
process gcd ( xin, yin, restart, result )
    in port xin[8], yin[8], restart;
    out port result[8];
[
    boolean x[8], y[8];
    tag a, b;
    constraint mintime from a to b = 3 cycles;

    /* set output to zero during computation */
    write result = 0;

    /* wait for restart to go low */
    a: while ( restart )
        ;

    /* sample inputs */
    b: x = read(xin); y = read(yin);

    /* Euclid's algorithm */
    if ( (x != 0) & (y != 0) ) {
        repeat {
            while (x >= y)
                x = x - y;
            /* swap values */
            < y = x; x = y; >
        } until (y == 0);
    } else
        x = 0;

    /* write result to output */
    write result = x;
]
```

Figure 2: Example of a *HardwareC* description to find the greatest common divisor of two values.

- *Constraint specification* – Timing constraints are supported through tagging of operations, where lower and upper bounds are imposed on the time separation between the tags. Timing constraint are useful in interface specification by constraining the time separation between I/O operations. Resource constraints limit the number of and binding of operations to resources in the final implementation.

An example of a *HardwareC* description that computes the greatest common divisor of two numbers is given in Figure 2. The model `gcd` waits until the `restart` signal is low, samples the inputs, then performs Euclid's algorithm iteratively. A minimum timing constraint is specified from the start of the loop to the reading of the inputs.

3 Hercules – Behavioral Synthesis

The objective of *behavioral synthesis* is to identify the maximal parallelism that exists in the input description. This gives an indication of the *fastest* design that the system can produce, assuming that in the design implementation each operation is implemented by a dedicated hardware component. While this assumption may not be realistic in some cases due to area and interconnection costs, it is important to compute the related performance as a limiting bound for a given behavior.

The input *HardwareC* description is parsed and translated first into an abstract syntax tree representation, which provides the underlying model for semantic analysis and behavioral transformations. The transformations are categorized into *user-driven* and *automatic* transformations. User-driven transformations are optional, and allow the designer the capability of modifying the model calls and hierarchy of the input description. They include the following:

- *Selective in-line expansion of model calls*, where a call to a model is replaced by the functionality of the called model. Once expanded, the optimization algorithms can be applied across the call hierarchy.
- *Selective operator to library mapping*, where operators, such as "+" or ">=", in the input description are mapped into calls to specific library template models. Although an operator can be synthesized in a variety of different implementation styles, the designer is often constrained to elements of a particular library. With such mapping, the designer has the flexibility to select the specific implementation for the operators. If no mapping is given, then by default the operators are implemented as combinational logic expressions.

Automatic transformations optimize the behavior by performing transformations similar to those found in optimizing compilers. The automatic transformations are carried out without human intervention, and include the following:

- `For`*-loop unrolling*, where fixed-iteration loops are unrolled to increase the scope of the optimizations.
- *Constant and variable propagation*, where the reference to a variable is replaced by its last assigned value.
- *Reference stack resolution*, where multiple and conditional assignments to variables are resolved by creating *multiplexed values* that can be referenced and assigned [11].
- *Common sub-expression elimination*, where redundant operations that produce the same results are removed.
- *Dead-code elimination*, where operations whose effects are not visible outside the model are removed.
- *Collapse conditional*, where conditionals with branches containing only combinational logic are collapsed to increase the scope in which logic synthesis can be applied.
- *Data-flow analysis*, where data and control dependencies among the operations are identified.

Upon completion of the automatic transformations, the behavior is optimized with respect to the data-dependencies that exist among the operations. At this point, *combinational coalescing* is performed to group together combinational logic operations into *expression blocks*. The expression blocks define the largest scope (without crossing register boundaries) in which logic synthesis can be applied, and identify the critical combinational logic delays through the design. They are directly passed to logic synthesis for minimization, the results of which are fed-back as estimates on area and timing that are used to refine the design. Combinational coalescing is important particularly for ASIC designs because of their extensive use of logic expressions in the hardware specification.

Sequencing Graph Model. The optimized behavior resulting from behavioral synthesis is translated into a graph abstraction called the *sequencing intermediate form* (SIF). The sequencing graph is a concise way of capturing the partial order among a set of operations, and it is modeled as a polar, directed acyclic graph. The vertices represent the operations to be performed, and the edges represent the dependencies that are either explicit in the hardware specification, or represent dependencies due to *data-flow* restrictions or hardware *resource-sharing* considerations. A vertex is enabled when all its

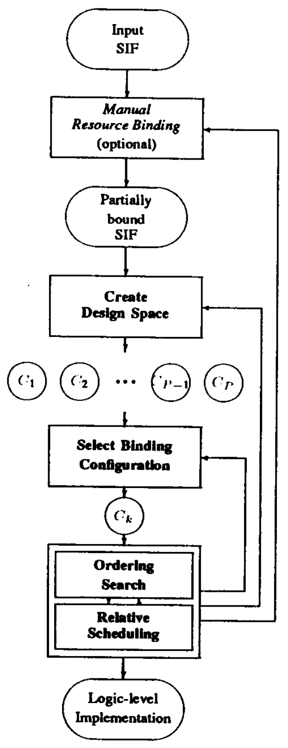
Input SIF
Manual Resource Binding (optional)
Partially bound SIF
Create Design Space
C_1 C_2 ... C_{P-1} C_P
Select Binding Configuration
C_k
Ordering Search
Relative Scheduling
Logic-level Implementation

Figure 3: Block diagram of the *Hebe* structural synthesis system.

predecessors have completed execution. Since a vertex may have multiple predecessors and successors, the model supports *multiple threads of concurrent execution flow*.

The vertices are categorized as either *simple* or *complex* vertices. Simple vertices include primitive computations in the language, such as arithmetic or logic expressions and message passing commands. Complex vertices allow groups of operations to be performed, and include model calls, conditionals, and loops. The complex vertices induce a hierarchical relationship among the graphs. A call vertex invokes the sequencing graph corresponding to the called model. A conditional vertex selects among a number of branches, each of which is modeled by a sequencing graph. A loop vertex iterates its body until the exit condition is satisfied; the body of the loop is also a sequencing graph. The sequencing graph is *acyclic* because only structured control-flow constructs are assumed (no `goto`), and loops are broken through the use of hierarchy.

The sequencing graph model can be simulated by **Arianne**. Therefore, *HardwareC* descriptions can be simulated at the functional level by transforming them first into the SIF representation. Since behavioral synthesis in Hercules is very efficient and fast, validation of the *HardwareC* models via simulation can be effectively supported. The SIF is the underlying representation for structural synthesis.

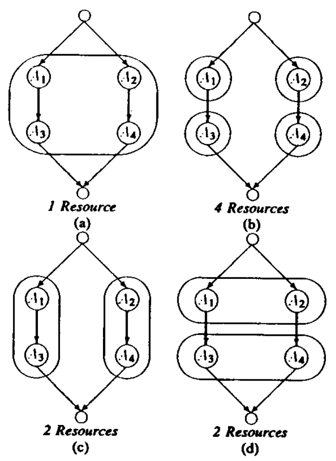

Figure 4: Examples of binding configurations, where operations within a group are bound to the same resource instance.

4 Hebe – Structural Synthesis

The objective of the *Hebe* structural synthesis system is to explore the design tradeoffs to obtain a suitable structure that satisfies the user constraints on area and timing. The constraints can either be specified in the input *HardwareC* description, or entered interactively by the designer. *Hebe* provides a flexible underlying representation of the design space that serves as the basis both for algorithmic exploration as well as for user-driven synthesis. In particular, the designer can either let *Hebe* explore the design tradeoffs automatically, or manually guide the direction of synthesis by imposing constraints on the design.

Structural synthesis involves performing two tasks – *resource binding*, where operations are assigned to hardware resources, and *scheduling*, where operations are assigned to control states. An effective strategy is to perform resource binding before schedule to provide scheduling with detailed interconnection delays, as in *Caddy* [14] and *BUD* [15]. This basic approach is extended in *Hebe* to provide closer interaction and guidance to the designer, and is shown in Figure 3. The flow of structural synthesis in *Hebe* is to first bind operations to specific resources, then perform scheduling to find a schedule that satisfy the timing constraints. The process repeats for different possible binding alternatives.

A **resource pool** is a set of hardware resources (e.g. implementations of models) with an upper-bound on the number of instances of each type of hardware resources that the user allows in the final implementation. A **binding configuration** is a matching of the operations (i.e. the vertices of the sequencing graph) with specific resources in the resource pool. The **design space** is the entire set of binding configurations. Examples of binding configurations for a sequencing graph containing four calls to model A are shown in Figure 4. All operations that are grouped together share the same resource instance in the final implementation, e.g. the binding configuration of Figure 4(a) utilizes one resource instance, the binding configuration of Figure 4(b) utilizes four resource instances, etc.

An important aspect of the design space formulation is that it is a *complete* characterization of the entire set of possible design tradeoffs for a given allocation of resources, and offers two important advantages:

- *Uniformly incorporates partial binding information.* In some circuits the designer may wish to bind certain operations to resources in order to achieve high-level design goals. This information can be used to limit the design space such that the synthesis system focus on the remaining unbound operations. At the extreme, if all operations are bound, then the design space trivially reduces to a single point.
- *Optimal scheduling under timing constraints.* Since *Hebe* decouples resource binding from scheduling, the latter problem can be solved exactly and efficiently, even under timing constraints, for each binding configuration.

The size of the design space may be large, because it grows exponentially with the number of shareable resources. However, it is often the case in ASIC designs that the number of shareable resources is sufficiently small to make systematic exploration of all binding configurations practical. Furthermore, bounding techniques based on evaluation of various *cost criteria* can be used to prune the design space, and speed the search for a suitable implementation.

Select Binding Configuration. Given a design space, *Hebe* supports both *exact* and *heuristic* search of the binding configurations. The system supports a set of *cost criteria* that are used to evaluate the design space. The cost criteria represent the effect of a particular binding configuration on the area, interconnection, and delay of the final implementation, and include:

- *Area Cost*: A binding configuration implies a certain degree of resource utilization and sharing. The area cost corresponds to the area costs of the resources in the resource pool.
- *Interconnection Cost*: The interconnection structure is the steering logic that guides the appropriate values to their proper destinations in the final implementation. Since a binding configuration is a complete assignment of operations to resources, the interconnection structure is completely specified. The interconnection cost is proportional to the size and delay of the interconnection structure.
- *Width Cost*: The *width* of a binding configuration is the number of threads of parallelism that *need to be serialized* in order to avoid resource contentions. In particular, all operations bound to the same resource instance should not execute simultaneously, e.g. either there exists sequencing dependencies among the operations, or the operations occur in mutually exclusive branches of a conditional. The width cost is a heuristic measure of the effect of the binding configuration on the performance of the design.

The decision of whether one alternative is favorable with respect to another depends on the relative importance of these criteria, which is determined by the value of a *weight* associated with each criterion. *Hebe* provides a flexible framework in which the designer can experiment with different design goals by adjusting the values of the weights. The design goals indicate the emphasis of the final implementation with respect to area and/or performance, and include:

- Find the *minimum area* configuration that satisfies the timing constraints, or

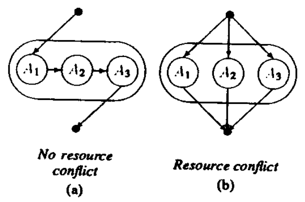

Figure 5: Example of (a) no resource conflict, and (b) resource conflict, among operations bound to the same resource.

- Find the *maximum performance* configuration that satisfies the area constraints

The designer can focus the synthesis efforts on the binding configurations with acceptible costs. For example, if the goal is to minimize the area, then the area and interconnection costs can be used to identify the binding configurations with minimal area. Likewise, if the goal is to maximize performance under area constraints, then the area and interconnection costs can bound the search to those configurations that meet the area constraints, while the width cost can provide further pruning of the design space.

Ordering Search. Once a resource binding is selected, the operations bound to the same resource component are *ordered* to resolve any resource conflicts that may occur. For example, three calls to model A executing in parallel but bound to the same resource must be serialized to ensure that they cannot execute simultaneously, as shown in Figure 5. The goal of the *ordering search* is to find a set of dependencies among the operations that resolves the resource conflicts. A branch-and-bound approach is used to explore the ordering alternatives. Since resource sharing can occur across the sequencing graph hierarchy, and dependencies may exist among the operations, not all possible orderings are valid. Therefore, significant pruning of the search space can be achieved. Once an ordering is found, the sequencing graph is serialized accordingly and scheduling is applied. If no schedule exists under timing constraints, then another ordering is tried. If no schedule exists for any valid ordering, then the binding configuration is discarded.

Relative Schedule. Given an ordering, the sequencing graph is free from resource conflicts. However, scheduling is necessary to define the detailed temporal relationships among the operations to satisfy the imposed timing constraints. A complication arises in that some of the operations may have *unbounded execution delays*, corresponding to synchronization primitives and data-dependent loops. The unbounded delay operations invalidate the traditional scheduling formulation where operations are statically assigned to specific time slots. We use a novel technique called *relative scheduling* that uniformly supports operations with fixed and unbounded delays [16].

The scheduling problem under timing constraints is modeled by means of a *constraint graph*. The vertices of the polar directed graph represent the operations, and the weights on the edges represent the timing constraints between pairs of operations. We define a subset of the vertices, called *anchors*, that serve as reference points for specifying the start times of the operations. The anchors consist of the source vertex and the set of unbounded delay vertices. *Offsets* are then defined with respect to each anchor of the graph. In particular, the *anchor set* of a vertex is the set of anchors that are predecessors to the vertex, and represents the *unknown* factors that affect the activation time of the vertex. The start time of a vertex is then generalized in terms of fixed time offsets from the completion of each anchor in its anchor set. Note that if there are no unbounded delay vertices in the graph, then the start times of all operations will be specified in terms of offsets from the source vertex, which reduces to the traditional scheduling formulation.

An important consideration during scheduling is whether the timing constraints can be satisfied for any value of the unbounded delay operations. We use the concept of *well-posed* verses *ill-posed* timing constraints in the presence of unbounded delays [16]. Specifically, a timing constraint is well-posed if its satisfiability does not depend on any unbounded delays. Given an ordering of shareable operations to resolve resource conflicts, additional serialization may be required to make the constraints well-posed. If no consistent serialization can be found, or if the constraints are not satisfiable, then the ordering is rejected as unfeasible. Otherwise, a *minimum* relative schedule can be computed by means of an *iterative incremental scheduling* algorithm. The time complexity of making the constraints well-posed and the scheduling algorithm are both polynomial. This allows relative scheduling to be effectively integrated within the ordering search.

Once a satisfactory structure with respect to both resource and timing constraints is obtained, a logic-level description is generated for the data-path, the interconnection, and the control circuitry to activate the data-path according to a given schedule. We use logic synthesis to perform combined synthesis of both control and data-path, the result of which can be mapped to a library, or implemented as macro-cells.

5 Implementation and Design Experiences

Hercules and *Hebe* have been implemented in C, with approximately 100,000 lines of code. They have been tested on the benchmark circuits for high-level synthesis. In addition, the system has been used to design three ASIC circuits at Stanford University, namely a Bi-dimensional Discrete Cosine Transform (BDCT) chip [17], a Digital Audio Input Output (DAIO) chip [18], and a decoder chip for the Multi-Anode Microchannel Array (MAMA) detector for the space telescope [19].

The BDCT chip is used for video compression applications. An 8×8 BDCT architecture was synthesized by *Hercules* and implemented in a compiled macro-cell design style [13] as a 9×9 mm^2 image in $2\mu_m$ CMOS technology. The DAIO chip provides an interface, following the Audio Engineering Standard (AES) protocol, between a standard 16/32 microprocessor bus with audio devices, such as compact disk or digital audio tape player. The DAIO specification in HardwareC was compiled and mapped into a logic netlist suitable for implementation in LSI Logic 9K-series sea-of-gates technology. The logic specification had about 6000 equivalent gates. The MAMA chip is designed to discriminate the information generated by a multi-anode detector in a space telescope. Also described in HardwareC, it was synthesized and fabricated with LSI Logic 9K-series sea-of-gates technology.

6 Acknowledgments

Thomas Truong implemented the SIF simulator *Arianne*. The authors would like to thank R. Gupta and T. Truong for helpful comments.

This research was sponsored by NSF, under grant No. MIP 8719546, by AT&T and DEC jointly with NSF, under a PYI Award program, and by a fellowship provided by Phillips/Signetics.

References

[1] A. de Geus, *Logic synthesis Speeds ASIC Designs*, IEEE Spectrum, Vol 26, No. 8, August 1989, pp. 27-31.

[2] M. Crastes de Paulet, C. Duff, R. Leveugle, F. Poirot, G. Saucier, P. Sicard, *ASYL: A Logic and Architecture Design Automation System*, Proceedings of EuroAsic 89, Jan 1989.

[3] J. Rabaey, H. De Man, et. al., *Cathedral II: A Synthesis System for Multiprocessor DSP Systems*, in *Silicon Compilation*, Ed. D. Gajski, Addison Wesley 1988, p. 311-360.

[4] J.Huisken, H.Janssen, P.Lippens, O.McArdle, R.Segers, P.Zegers, A. Delaruelle and J. van Meerbergen, *Efficient Design of Systems on Silicon with PYRAMID*, in *Logic and Architecture Synthesis for Silicon Compilers*, North Holland, Amsterdam, 1989.

[5] R. Walker, D. Thomas, *The Systems Architect Workbench*, Kleuer Academic Press, 1989.

[6] A. Parker, J. Pizarro, M. Mlinar, *MAHA: A Program for Data Path Synthesis*, Proceedings 23th Design Automation Conference, June 1986, p. 461-466.

[7] F. Brewer, D. Gajski, *Knowledge Based Control in Micro Architecture Design*, Proceeding 24th DAC, p. 203-209, June 1987.

[8] P. G. Paulin, J. P. Knight, E. F. Girczyc, *HAL: A Multi-Paradigm Approach to Automatic Data-path Synthesis*, Proceedings 23th Design Automation Conference, June 1986, p. 263-270.

[9] R. Brayton, R. Camposano, G. De Micheli, R. Otten, J. van Eijndhoven, *The Yorktown Silicon Compiler System* in *Silicon Compilation*, Ed. D. Gajski, Addison Wesley 1988, p. 204-310.

[10] G.Zimmermann, *The MIMOLA Design System: Detailed Description of the Software System*, Proc 16th Des Autom. Conf, 1979, pp 56-63.

[11] G. De Micheli, D. Ku, *HERCULES - A System for High-Level Synthesis* Proceedings 25th Design Automation Conference, June 1988, p. 483-488.

[12] D. Ku, G. De Micheli, *HardwareC - A Language for Hardware Design* Stanford Technical Report, CSL-TR-88-362, August 1988, and CSL-TR-90, April 1990 (Version 2.0).

[13] F. Mailhot, G. De Micheli, *Automatic Layout and Optimization of Static CMOS Cells* Proceedings of ICCD, Rye, New York, pp. 180-185, 1988.

[14] R. Camposano, W. Rosenstiel, *Synthesizing Circuits from Behavioral Descriptions*, IEEE Trans. on CAD, p. 171-180, Feb 1989.

[15] M. J. McFarland, *Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions*, Proceedings 23rd Design Automation conference, pp. 474-480, 1986.

[16] D. Ku, G. De Micheli, *Relative Scheduling Under Timing Constraints*, Stanford Technical Report, CSL-TR-89-401, November 1989, and Proceedings of 27th Design Automation Conference, Orlando, Florida, June, 1990.

[17] V. Rampa, G. De Micheli, *The Bi-dimensional DCT Chip*, Proceedings of ISCAS, 1988.

[18] M. Ligthart, A. Bechtolsheim, G. De Micheli, A. El Gamal, *Design of a Digital Audio Input Output Chip*, Custom IC Conference, May 1989.

[19] D. B. Kasle, *High resolution decoding techniques and single-chip decoders for multi-anode microchannel arrays*, Proceedings of SPIE, Vol. 1158, 1989, pp. 311-318.