Constrained resource sharing and conflict resolution in Hebe

David C. Ku and Giovanni De Micheli
Center for Integrated Systems, Stanford University, Stanford, CA 94305-4055, USA

Abstract. Hardware resources can be shared to reduce the area of the resulting design. The synthesis system must ensure that no *resource conflicts* arise due to simultaneous access of a shared hardware resource. With traditional scheduling formulations where operations are statically assigned to control steps, conflict resolution simply determines whether two operations can execute concurrently based on their control step assignment. In this case, operations are assumed to have fixed execution delay. For hardware models that supports external synchronization and handshaking, however, operations may have *unbounded execution delay*, e.g., detecting the rising edge of a signal. The presence of unbounded delay operations invalidates the traditional scheduling and conflict resolution approaches. We formulate in this paper conflict resolution as the task of serializing operation bound to the same hardware resource. A technique called *constrained conflict resolution* is presented to resolve resource conflicts such that the resulting design satisfies the required timing and handshaking requirements. The timing constraint topology is used to reduce the computation time of the algorithm. This technique extends the *relative scheduling* formulation to support resource sharing under timing constraints. We describe both exact and heuristic algorithms to resolve resource conflicts; these algorithms are implemented in a synthesis system called *Hebe* that is targeted towards the synthesis of *Application-Specific Integrated Circuit* designs. Results of applying the system to the design of benchmark and complex ASIC designs are presented.

Keywords. Resource conflict resolution, high-level synthesis, automated synthesis, behavioral synthesis, hardware model

1. Introduction

The trend of *Very Large Scale Integration* (VLSI) circuit designs is towards greater density and complexity. An effective way to deal with the increasing complexity of designs is to raise the level of abstraction at which circuits are

Elsevier
INTEGRATION, the VLSI journal 12 (1991) 131–165

designed. *High-leve synthesis* refers to computer-aided design approaches starting from the algorithmic description level. The benefits of such a methodology include shortened design time to reduce design cost, ease of modification of the hardware specifications to enhance design reusability, and the ability to more effectively explore the different design tradeoffs between the area and performance of the resulting hardware.

Previous work in high-level synthesis addressed mainly general-purpose processor and signal processing designs [1]. In these designs, the behavior usually consists of a set of computations that are performed within a certain amount of time. Synthesis of these designs can produce cost-effective implementations because the synthesis system can take advantage of domain-specific knowledge to optimize the underlying architecture. In contrast, *Application-Specific Integrated Circuit* (ASIC) designs perform computations that are *specific* to a particular application. An example in an Ethernet controller that coordinates the activities

David Ku was born in Tapei, Taiwan, in April 1964. He received in 1986 the BS degree in Electrical Engineering, *Summa cum Laude*, and the BS degree in Computer Science, *Summa cum Laude*, both from the University of Utah. He received the MS and PhD degrees in Electrical Engineering from Stanford University in 1987 and 1991, respectively. His research interests include high-level synthesis, control synthesis, design automation and CAD frameworks.

David is a CIS/Signetics FMA Fellow in the Center for Integrated Systems at Stanford University since 1989. He was granted the AT&T Fellowship in 1986. He received the *Most Outstanding Senior in Electrical Engineering* award from the University of Utah in 1986, and again the *Most Outstanding Junior in Electrical Engineering* in 1985. He also received the Minority Scholastic Award in 1986, the Clyde Christensen Scholarship in 1985, the Northwestern Energy Scholarship in 1984, and the University Scholarship from 1982–1986. David is a member of IEEE and ACM.

Giovanni De Micheli is Associate Professor of Electrical Engineering and, by courtesy, of Computer Science at Stanford University. From 1984 to 1986 he worked at the IBM TJ. Watson Research Center, Yorktown Heights, New York, where he was project leader of the Design Automation Workstation group. Previously he held positions at the Department of Electronics of the Politecnico di Milano, Italy and at Harris Semiconductor, Melbourne, Florida.

He received at Dr. Eng. degree, *Summa cum Laude*, in Nuclear Engineering from the Politecnico di Milano, Italy, in 1970, a MS and a PhD degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 1980 and 1983 respectively. Dr. De Micheli was granted a *Presidential Young Investigator* award in 1988. He received the 1987 *Best Paper Award* for the best paper published on the IEEE Transactions on CAD/ICAS and a *Best Paper Award* at the 20th Design Automation Conference, in June 1983.

His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, optimization and verification of VLSI circuits. He is co-editor of the book: *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1987. He was also co-director of the Advanced Study Institute on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, under the sponsorhip of NATO in 1986 and in 1987.

Dr. De Micheli is a Senior Member of IEEE. He is associate editor of the IEEE *Transactions on Circuit and Systems* and of *Integration: the VLSI Journal*. He was technical and general chairman of the *International Conference on Computer Design — ICCD* in 1988 and 1989 respectively. He has served as member of the technical committee of the *ICCD*, *ICCAD* and *DAC* Conferences. He served also as a member of the executive committee of the New York Chapter of the Computer Society in 1985 and 1986.

between a microprocessor and an Ethernet line. In this case the controller is constrained to both the microprocessor architecture and the Ethernet protocol, requiring complicated handshaking protocols to interface between concurrently executing components and strict timing constraints on the handshaking.

We believe ASIC designs to be particularly suited for high-level synthesis because the manual synthesis of these designs is tedious and error prone. The use of a high-level synthesis methodology significantly reduces the design time and cost, which is often as important as minimizing area or improving performance. Although logic synthesis techniques are well established and have been used for industrial ASIC chip designs [2], very few commercial designs have been synthesized using high-level synthesis techniques. This lack of acceptance is most likely due to a mismatch between the requirements of ASIC designs and the assumptions and capabilities of existing high-level synthesis systems. One largely unresolved issue is the difficulty of integrating a synthesized design with other components in the system. In particular, a synthesized design needs to communicate with other modules in the system using a given handshaking protocol and possibly under timing requirements.

To address the issues related to ASIC synthesis, we have developed a high-level synthesis system which consists of two parts: *Hercules* that performs the front-end parsing and behavioral optimizations [3], and *Hebe* that synthesizes one or more logic-level implementations that realize the given behavior [4]. This paper presents the hardware model and synthesis methodology of Hebe, focusing on its resource sharing and conflict resolution strategies. Specifically, a novel technique called *constrained conflict resolution* is presented to resolve resource conflicts by serializing operations bound to the same hardware resource, such that the resulting design satisfies the required timing and handshaking requirements. In addition to supporting unbounded delay operations and timing constraints, the technique uses the timing constraint topology to reduce the computation time of the algorithm. This technique extends the *relative scheduling formulation* [5] to support resource sharing under timing constraints.

This paper is organized as follows. We put our research in perspective by summarizing the related research in the area in Section 2 and describing the overall synthesis flow in Section 3. Section 4 describes the *sequencing graph* model of hardware behavior that is used as the underlying representation for the synthesis algorithms in Hebe. Hardware resources and the design space formulation are described in Section 5. Section 6 presents the constrained conflict resolution formulation and algorithms as the major contribution of this paper. The system has been implemented and applied to the synthesis of benchmark circuits and ASIC designs starting from behavioral level specification. We present the experimental results and conclude in Section 7.

2. Related research

The focus of most high-level synthesis efforts todate has been on synthesizing and optimizing the data-path [1]. While these systems have been effective in

synthesizing certain types of designs and efficient algorithms have been developed to address many difficult synthesis problems, they do not adequately address the synthesis of ASIC designs with complex handshaking protocols and strict timing requirements.

Most approaches assume that the execution delay of operations is bounded, which stems from the use of pre-designed micro-architectural library modules as primitive hardware elements for the data-path. This implies that hardware interfacing and synchronization, modeled as operations with unbounded execution delay, are not supported. This research incorporates external interfacing and handshaking requirements as an integral part of hardware model and performs synthesis based on this hardware model.

In contrast to micro-architectural synthesis approaches where the final implementation is an interconnection of primitive functional blocks, this research uses logic synthesis as the underlying synthesis base. The characterization of resources to evaluate hardware 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.

With the exception of CADDY [6], SAW [7], and SALSA [8], most synthesis approaches do not support detailed timing constraints. That is, they support either no timing constraints at all or they support at most constraints on the overall latency. This may be inadequate to describe complicated requirements on the timing of operations. SAW, because of the heuristic nature of its scheduling step, cannot guarantee that if the algorithm fails to find a solution that satisfies the timing constraints then no solution is possible. Rigorous analysis of the consistency of detailed timing constraints is either limited or lacking. In contrast, our approach considers synthesis under detailed timing constraints in both the synthesis formulation and the algorithms. The proposed synthesis approach in the sequel guarantees that these synchronization and timing requirements are satisfied by the resulting synthesized hardware, when the constraints are satisfiable.

3. System overview

We consider synchronous non-pipelined hardware implementations. Hardware is modeled in the *HardwareC* language [9] and compiled into a logic-level circuit specification by two programs, called *Hercules* and *Hebe*. They form the front-end to the *Stanford Olympus Synthesis system*, a research project in computer-aided synthesis at Stanford University [10]. A block diagram of the Olympus system is shown in Fig. 1. We refer the reader to [10] and [12] for the details of the system.

Hercules takes as input an algorithmic description of hardware behavior in *HardwareC* [9]. It identifies the inter-operation parallelism in the input behavioral description by performing compiler optimizations such as dead-code

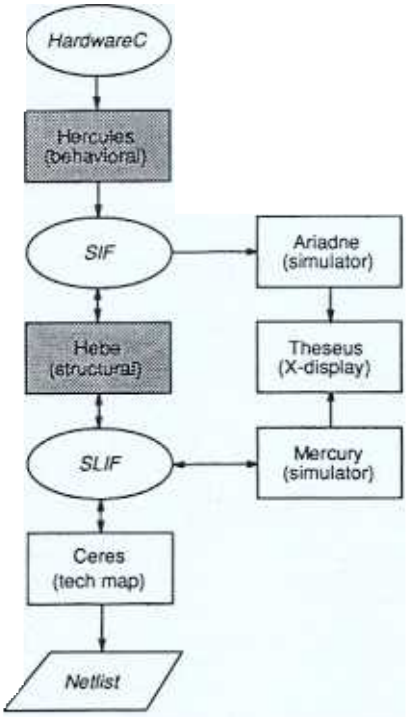


Fig. 1. The Stanford Olympus Synthesis system.

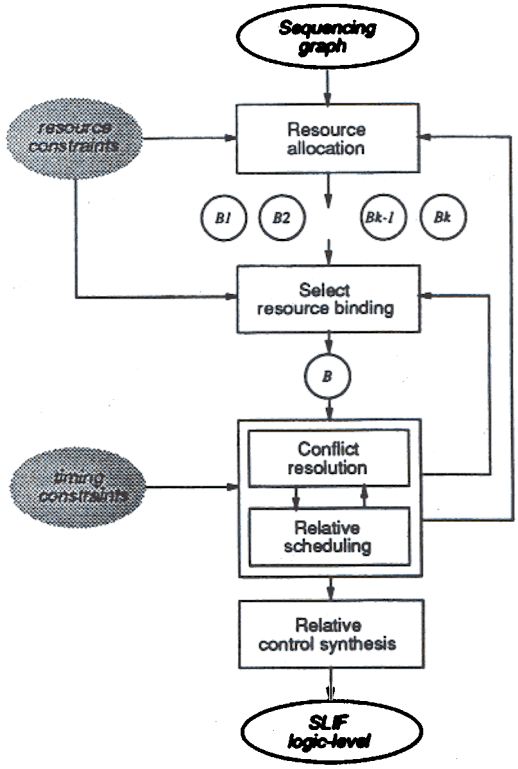


Fig. 2. Structural synthesis flow in Hebe.

elimination, constant and variable propagation, loop unrolling, and common subexpression elimination. Logic operations in the description are clustered to form blocks of combinational logic that are passed directly to logic synthesis for minimization and delay/area estimates. *Operation chaining*, where multiple operations are packed within a single control state, is supported through combinational coalescing. The optimized behavior is translated to an implementation-independent description of the hardware behavior in a graph-based representation, called the *Sequencing Intermediate Form* (SIF).

Hebe takes as input a hardware behavior represented by a sequencing (SIF) graph, and produces a synchronous logic-level implementation that realizes the original behavior. The input to Hebe consists of a sequencing graph model and the following constraints: *timing constraints* that specify upper and lower bounds on the time separation between activation of operation, *resource constraints* that both limit the number of instances allocated for each resource type and partially bind operations to specific allocated resources, and the *cycle time* for the final synchronous logic implementation. These constraints can be specified either in the input description or entered interactively by the designer. Note that they are not mandatory. For example, if the cycle time is not given, then the cycle time is by default equal to the critical combinational logic delay in the final logic-level

implementation. The final implementation contains both data-path and control. The data-path is an interconnection of functional units, registers and multiplexers.

Hebe performs resource allocation and binding before scheduling. This strategy has the advantages of providing scheduling with detailed interconnection delays and incorporating partial binding information to limit the number of design choices. The structural synthesis flow in Hebe is illustrated in Fig. 2. Among the features of the Hebe system, we would like to stress the support for:

- *Hardware model with concurrency, external synchronization, and detailed timing constraints.* To provide support for the requirements of ASIC designs with handshaking and timing requirements, our underlying hardware model supports multiple threads of concurrent execution flow, external synchronization modeled as *unbounded delay operations*, and minimum and/or maximum timing constraints on the activation of operations.
- *Partial binding of operations to structure.* Often the designer may wish to share resources by manually binding certain operations to resources in order to meet some high-level design requirements. Hebe incorporates this information to restrict the search space for a valid implementation.
- *Constraint-driven synthesis algorithms with provable properties.* Synthesis algorithms in Hebe are driven by timing and synchronization requirements which guarantee that the resulting implementation satisfies these constraints, or detect if no such implementations exist.
- *Systemic exploration of the design space.* Tradeoffs between area and performance provide a spectrum of implementation alternatives to the designer. In complex designs, viewing the design space as a smooth area–time curve is overly simplistic [13]. Furthermore, the curve provides evaluation of a design choice *after* it has been made rather than guiding the designer *during* the decision making process. Hebe supports a systematic search of the design space by considering all, or a subset, of the possible resource bindings. The search can be performed either interactively or automatically, using an evaluation of the possible design tradeoffs.

In our paradigm, 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.

4. Sequencing graph model

The **sequencing graph** model is a concise way of capturing the partial order among a set of operations. This model captures the precedence relationship among the operations and defines the execution flow in implementing a given behavior. To be more exact, a sequencing graph is a polar, hierarchical, vertex-

weighted, directed acyclic graph, denoted by $G_s(V, E_s, \delta)$. The vertices $V = \{v_0, \ldots, v_N\}$ represent operations to be performed, where v_0 and v_N denote the source and sink vertices, respectively, of the polar (single-source and single-sink) graph. Directed edges E_s represent sequencing dependencies among the operations. An integer weight $\delta(v_i)$ is associated with each vertex $v_i \in V$ representing its execution delay.

Sequencing dependencies can arise due to data-flow dependencies extracted from the behavioral model (i.e. a value must be written before it can be referenced), explicit sequencing that is specified in the input description (i.e. detect the rising edge of a control signal before reading a bus), or resource sharing restrictions that are introduced during structural synthesis (i.e. operations sharing the same hardware resource are serialized to avoid resource conflicts). A directed edge $s_{ij} \in E_s$ from vertex v_i to v_j means that v_j can begin executing only after the completion of v_i; v_i is called a *predecessor* of v_j, and v_j is called a *successor* of v_i.

Vertices are classified into different types according to the operations they perform. Vertices are further categorized as either *simple* or *complex*: simple vertices are primitive computations that do not involve other operations (i.e. arithmetic or logic operations and message passing commands), and complex vertices allow groups of operations to be performed. They include *model calls*, *conditionals*, and *loops*, and are analogous to structured control-flow constructs in most programming and hardware description languages. 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 over the body of the loop until its exit condition is satisfied, where the body of the loop is also a sequencing graph. The sequencing graph is *acyclic* because only structured control-flow constructs are assumed (i.e., no `goto`'s) and loops are broken through the use of hierarchy. All forms of conditional branching are represented as complex vertices in the graph model.

We separate the sequencing graph hierarchy into two components: *calling hierarchy* and *control-flow hierarchy*. Calling hierarchy refers to the nesting structure of procedure and function calls in the model. Control-flow hierarchy refers to the nesting structure of conditionals and loops in the sequencing graph. An example of control-flow hierarchy is shown in Fig. 3. Let M be a model which is represented in general by a hierarchy of sequencing graphs. The sequencing graph at the root of the hierarchy is called the *root graph* of M, denoted by G_M. The *cf-hierarchy* of G_M, denoted by G_M^*, is the control-flow hierarchy of G_M. In Fig. 5, the cf-hierarchy of model M consists of all four graphs in the figure.

The semantic interpretation of the sequencing graph is as follows. A vertex *executes* by performing its corresponding operation. For example, to execute a conditional vertex, operations in the selected branch are executed. Executing a sequencing graph is equivalent to executing the vertices according to the precedence relations implied by the graph starting from the source vertex. A vertex can

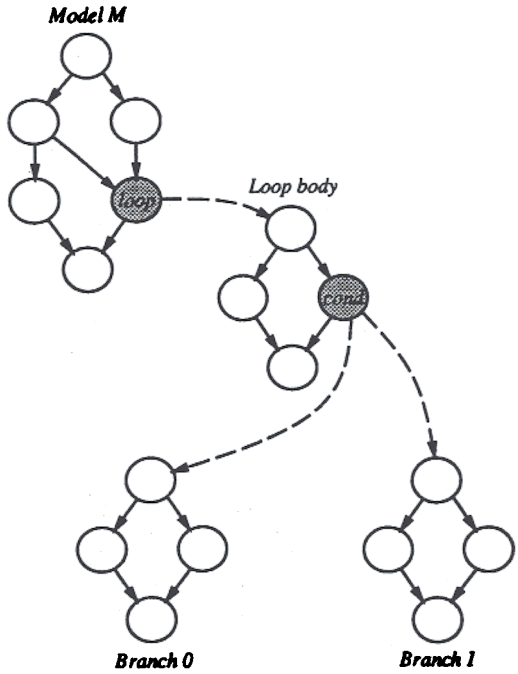


Fig. 3. Example of the control-flow hierarchy for model M containing a loop vertex, which in turn contains a conditional vertex with two branches.

execute only when its predecessors have completed execution. Since a vertex can have multiple predecessors and multiple successors, the model supports *multiple threads of concurrent execution flow*.

Unbounded delay operations. Each vertex represents an operation requiring an integral number of control steps (clock cycles), possibly zero, to execute. The *execution delay* of a vertex v_i represents the number of cycles it takes to execute. Execution delays are defined by the mapping $\delta: V \rightarrow Z^+$ from the set of vertices to non-negative integers, where $\delta(v_i) \geqslant 0$ denotes the execution delay of vertex v_i. These delays are derived either from the operation type, i.e. loading a register takes one clock cycle, or from estimates obtained through logic synthesis, i.e. delay is obtained by computing the critical delay through the logic expressions normalized to the cycle time.

A problem arises for conditionals and loops because their execution delays depend on external signals and events that are not known statically. We further categorize the vertices based on this observation by saying that a vertex has **bounded delay** if the time required to execute its operation is fixed for all input data sequences; otherwise, it has **unbounded delay** and is called an **anchor** of the graph. The delay associated with a bounded delay vertex depends solely on the

nature of the operation. Examples include addition and register loading. On the other hand, the time to execute an unbounded delay is data-dependent. Loops whose exit condition depends on some signal value, or message passing commands that synchronize between two concurrent processes are examples of unbounded delay vertices. Unbounded delay vertices are important to specify interfaces and handshaking protocols.

Constraint graph model. The sequencing edges represent the precedence relationships that are due to data-flow and control-flow dependencies. We describe now the derivation of a *constraint graph* model from the sequencing graph model with timing constraints. The constraint graph captures the timing behavior and timing requirements of a given sequencing graph.

Consider a sequencing graph $G_s(V, E_s, \delta)$. Let $T(v_i)$ represent the *start time* of v_i, i.e. the time at which v_i begins execution with respect to the source vertex of G_s. Detailed timing constraints consist of the following:

- *Minimum* timing constraints $l_{ij} \geq 0$ from v_i to v_j, requiring that $T(v_j) \geq T(v_i) + l_{ij}$. This constraint implies that v_j should be activated at least l_{ij} cycles after the activation of v_i.
- *Maximum* timing constraints $u_{ij} \geq 0$ from v_i to v_j, requiring that $T(v_j) \leq T(v_i) + u_{ij}$. This constraint implies that v_j should be activated no more than u_{ij} cycles after the activation of v_i.

Timing constraints are defined only between vertices of the same sequencing graph; constraints across the graph hierarchy is not permitted.

The timing behavior of a sequencing graph $G_s(V, E_s, \delta)$ under timing constraints is captured by a polar, edge-weighted, directed **constraint graph** $G(V, E, \omega)$. A constraint graph is an alternate representation of the sequencing graph which emphasizes its timing requirements. Vertices of the constraint graph are identical to the vertices of the sequencing graph; they represent the *activation* of the corresponding operations. Edges capture the minimum and maximum timing relationships between the activation of operations. They are categorized into *forward* (E_f) and *backward* (E_b) edges, i.e. $E = E_f \cup E_b$. Weights are associated with the edges by the mapping $\omega: E \rightarrow Z$, which assigns to each edge e_{ij} a weight $\omega(e_{ij})$ that corresponds to the following inequality constraint between v_i and v_j:

$$T(v_i) + \omega_{ij} \leq T(v_j)$$

Forward edges have positive weights and represent minimum timing constraints; backward edges have negative weights and represent maximum timing constraints. The derivation of edges and weights from the sequencing graph and timing constraints is described below.

- *Sequencing edge* $s_{ij} \in E_s$: Create a forward edge $e_{ij} \in E_f$ with weight $\omega(e_{ij}) = \delta(v_i)$, modeling a minimum timing constraint equal to the execution delay of v_i.
- *Minimum timing constraint* l_{ij}: Create a forward edge $e_{ij} \in E_f$ with weight $\omega(e_{ij}) = l_{ij}$.

- *Maximum timing constraint* u_{ij}: Create a backward edge $e_{ji} \in E_b$ with weight $\omega(e_{ji}) - u_{ij}$, because $T(v_j) \leqslant T(v_i) + u_{ij}$ can be rewritten as $T(v_j) - u_{ij} \leqslant T(v_i)$.

The length of the longest path between two vertices v_i and v_j is denoted by $length(v_i, v_j)$, where all unbounded delays are set to zero; it is the minimum timing separation between v_i and v_j for all input data sequences.

5. Hardware resources and the design space

The data-path in the final hardware implementation consists of three types of elements: functional units, registers, and multiplexers. Functional units correspond to arithmetic operations (e.g. + or *) or to generic models (e.g. a procedure describing some application-specifc functions). Registers are introduced either by the input behavioral specification or as required to implement hardware sharing. Multiplexers form the interconnect logic to steer appropriate signals between functional units and registers.

In our approach, a *model* in the input description corresponds to a *resource* that can be allocated and shared among the calls to that model. Each different implementation of the called model represents a particular *resource type*, which has its own area and performance characteristics. Predefined operators, such as + or −, are either converted into calls to the appropriate library models or implemented, by default, as combinational logic. Therefore, the only operations whose implementing hardware can be allocated and controlled by the designer are calls to procedure or function models. This model of resources implies that resource sharing is possible only for *call* vertices in the sequencing graph model. We assume that the calling hierarchy is traversed bottom-up, where all called models in the control-flow hierarchy of a model have already been synthesized before the given model can be considered for synthesis. Hebe also assumes the resource type implementing each call vertex is specified prior to synthesis; it performs tradeoffs in the *number* of resource that are allocated, not in the *types* of resource implementing the operations.

There are several motivations for treating models and resources in this manner. First, since many complex ASIC designs use application-specific logic functions to describe hardware behavior, the delay and area attributes of these modules are not known *a priori* since they depend on the particular details of the logic functionality. Having the ability to synthesize each model in a bottom-up fashion according to its distinct needs allows the calling models to more accurately estimate their resource requirements. Second, the granularity of resource sharing can be controlled by the designer by modifying the calling hierarchy in the high level specification. Finally, instead of relying on parametrized and predefined modules, logic synthesis techniques applied hierarchically to each model can significantly improve the quality of the resulting design.

5.1. Design space formulation

For a sequencing graph $G_s(V, E_s, \delta)$, we focus our attention on the subset of vertices in G_s^* (cf-hierarchy of G_s) whose corresponding resources can be shared.

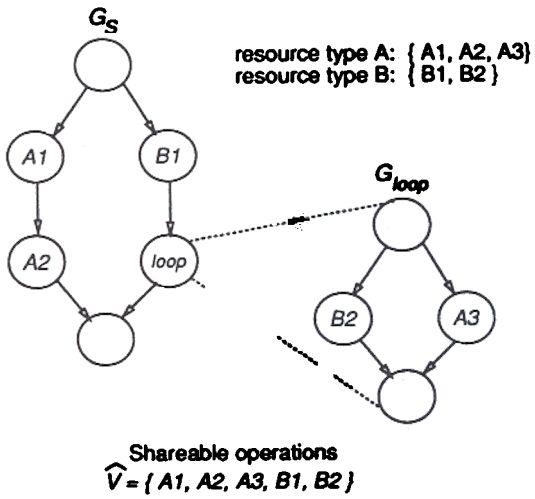


Fig. 4. Example of sequencing graph with 3 calls to model A and 2 calls to model B, where $\hat{V} = \{A1, A2, A3, B1, B2\}$.

These operations are called *shareable operations* and are denoted by $\hat{V} \subseteq V^*$, where V^* represents all vertices belonging to graphs in G_s^*. Shareable operations define the scope within which resource allocations and bindings are defined. Consider, for example, the sequencing graph hierarchy of Fig. 4. The root graph G_s contains 2 calls to model A and 1 call to model B. It also contains a loop, the body of which is a sequencing graph containing two call vertices: one to A and the other to B. The set of shareable operations is $\hat{V} = \{A1, A2, A3, B1, B2\}$.

Let $\mathcal{T}$ denote the set of resource types in $\hat{V}$. For example, the set $\mathcal{T} = \{$model A, model $B\}$ represents the $\mathcal{T}$ resource types for the example in Fig. 4. The *operation set* for type $t \in \mathcal{T}$, denoted as $O(t) \subseteq \hat{V}$, consists of shareable operations with resource type t. A *resource allocation* is formally defined as follows.

Definition 5.1. *Given a sequencing graph $G_s(V, E_s, \delta)$ and resource types $\mathcal{T}$, a* **resource allocation** *is the mapping $\alpha: \mathcal{T} \rightarrow Z^+$ from the set of resource types to positive integers, where $\alpha(t)$ denotes the number of resources allocated for resource type $t \in \mathcal{T}$.*

Each resource type in $\mathcal{T}$ must have at least one resource, i.e. $\alpha(t) \geq 1 \ \forall t \in \mathcal{T}$, since otherwise an implementation is not possible. A resource instance in an allocation α is described by a pair (t, i), where $t \in \mathcal{T}$ denotes the type of the resource instance and i $(1 \leq i \leq \alpha(t))$ denotes the specific allocated instance. For examples, Fig. 5 shows a resource allocation for the sequencing graph example in Fig. 4: 2 instances of model A $(\alpha(A) = 2)$ and 1 instance of model B $(\alpha(B) = 1)$. The range of possible allocations for model A is $1 \leq \alpha(A) \leq 3$ and for model B it is $1 \leq \alpha(B) \leq 2$.

Given a resource allocation α, a *resource binding* for a sequencing graph G_s is an assignment of shareable operations $\hat{V}$ to specific instances of the allocated resources. It is defined as follows.

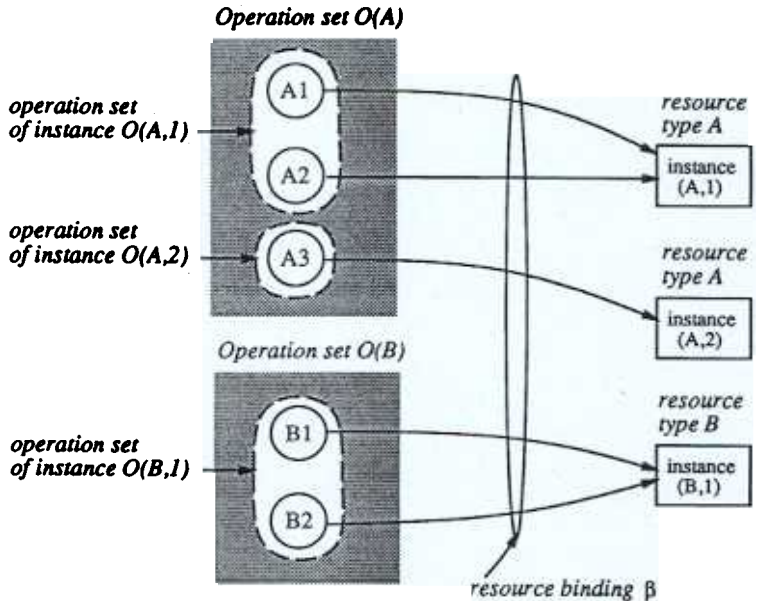


Fig. 5. Illustrating the relationship between shareable operations and allocated resources. The allocation is $\alpha(A) = 2$ and $\alpha(B) = 1$, and the arcs represent the resource binding β.

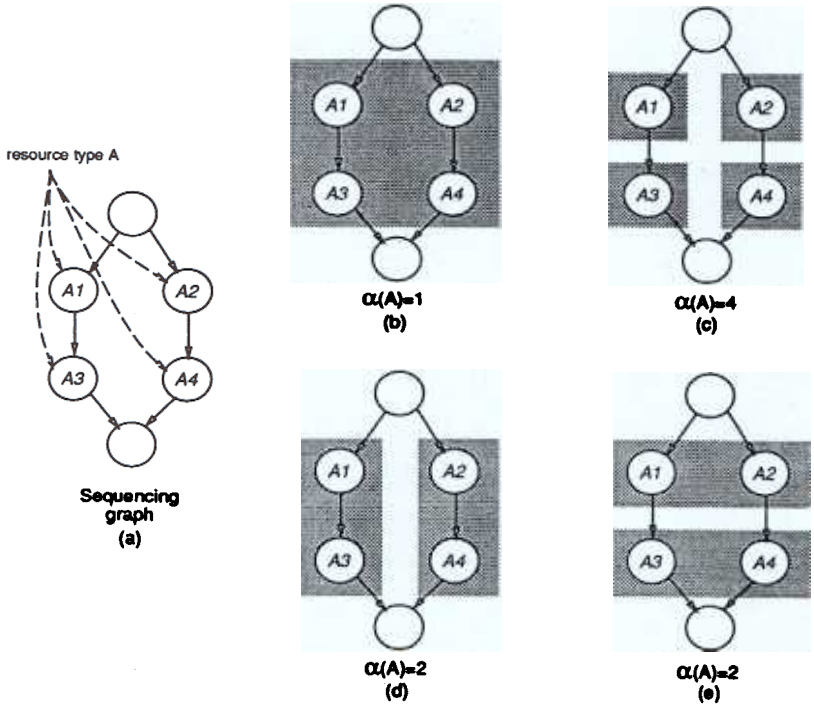


Fig. 6. Examples of different resource bindings, where operations within a shaded block are bound to the same resource instance.

Definition 5.2. *A* **resource binding** *of a sequencing graph G_s given a resource allocation α is a mapping $\beta: \hat{V} \rightarrow (\mathcal{T} \times Z^+)$, where $\beta(v) = (t, i)$ if operation $v \in \hat{V}$ is being implemented by the* ith *instance of resource type $t \in \mathcal{T}$, $1 \leqslant i \leqslant \alpha(t)$; otherwise, $\beta(v)$ is undefined.*

A vertex for which β is defined is called a *hardware-bound* vertex; otherwise it is called an *hardware-unbound* vertex. If there are no hardware-unbound vertices in $\hat{V}$, then β is a *complete binding*; otherwise, it is a *partial binding*. Figure 5 shows a binding β defined on the sequencing graph example of Fig. 4 for the allocation $\{\alpha(A) = 2, \alpha(B) = 1\}$.

Examples of different resource bindings for a sequencing graph containing 4 calls are shown in Fig. 6 (b) through (e). All operations grouped by the shaded rectangle share the same hardware resource in the final implementation, e.g. the binding of (b) utilizes one resource, the binding of (c) utilizes four resources.

A partial binding can be defined for more than one allocation, where it is assumed that the number of resources required by the partial binding is satisfied by these allocations. This leads to the concept of compatible bindings, defined below.

Definition 5.3. *A complete binding β_c is* **compatible** *with a partial binding β_p for a resource allocation α, denoted by $\beta_c \overset{\alpha}{\prec} \beta_p$, if for all hardware-bound vertices $v \in \hat{V}$, the implementing resource instance is identical: $\beta_p(v) = \beta_c(v)$.*

In other words, a compatible binding can be derived from a given partial binding by mapping all hardware-unbound vertices to resource instances. Obviously, if β_p is already a complete binding (i.e. all operations are pre-assigned to resources), then there is a single compatible binding.

Each resource instance (t, i) is bound to a subset of vertices $O_{(t,i)} \subseteq \hat{V}$ called the instance operation set of (t, i), i.e. $O_{(t,i)} = \{v \mid \beta(v) = (t, i)\}$. The cardinality of $O_{(t,i)}$ is denoted by $|O_{(t,i)}|$. Instance operation sets partition $\hat{V}$ into groups, each of which is implemented by a particular allocated resource instance. Obviously, an instance operation set of (t, i) is a subset of the operation set of t, i.e. $O_{(t,i)} \subseteq O(t)$, and the union of the instance operation sets for all allocations of t is equal to the operation set of t, i.e. $\bigcup_{i=0}^{\alpha(t)} O_{(t,i)} = O(t)$.

If there is a single instance allocated for a particular resource type t, then all operations with resource type t are automatically bound to that instance. If there is a single resource type t in the graph, then t is implied and the instance operation set is abbreviated as O_i. For example, instance operation sets for a binding β are shown in Fig. 5. In Fig. 6(b), there is a single instance operation set $\{A_1, A_2, A_3, A_4\}$; in Fig. 6(c), there are four operation sets $\{A_1\}$, $\{A_2\}$, $\{A_3\}$, $\{A_4\}$. The definitions presented in this section are summarized in Table 1.

Given resource constraints in the form of a partial binding β_p and a set of resource allocations $\{\alpha_1, \ldots, \alpha_k\}$, the design space of a sequencing graph $G_s(V, E_s, \delta)$ is defined as follows.

Table 1
Summary of resource allocation and binding terminology

Symbol	Name	Description
	root graph	Root of sequencing graph hierarchy
	cf-hierachy of G_s	Control-flow hierarchy of G_s
	operation domain of G_s	All vertices in cf-hierarchy G_s^*
	shareable operations of G_s	All call vertices in V_s^*
	resource type set	Set of all resource types for G_s
	operation set of t	Subset of $\hat{V}$ with type $t \in \mathcal{T}$
	resource allocation for t	# of allocated instances of type $t \in \mathcal{T}$
	resource instance	ith instance of type $t \in \mathcal{T}$
	partial resource binding	Partial mapping of $\hat{V}$ to α
	complete resource binding	Complete mapping of $\hat{V}$ to α
	instance operation set of (t, i)	All vertices bound to resource instance (t, i)

Definition 5.4. *For a set of resource allocations* $\{\alpha_1, \ldots, \alpha_k\}$ *and a partial binding* β_p, **design space** S *of* G_s *is the entire set of possible compatible bindings, i.e.* $S = \{\beta_c \mid \beta_c \overset{\alpha_i}{\prec} \beta_p, \forall \alpha_i\}$.

The design space of possible resource bindings for Fig. 6 with allocation $\alpha = 2$ is illustrated in Fig. 7. There are seven different resource bindings in the design space.

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 alloc-

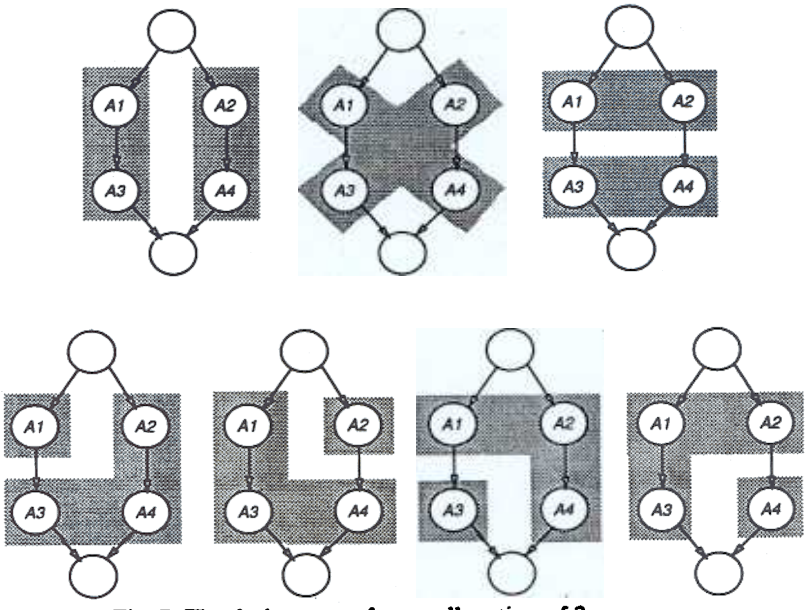


Fig. 7. The design space for an allocation of 2 resources.

ation of resources. This formulation allows partial binding information to be uniformly incorporated, where the partial binding is used to limit the design space so that the synthesis system focuses on the remaining unmapped operations. At the extreme, if all operations are bound initially, then the design space trivially reduces to a single point.

5.2. Design space exploration

With the design space formulated as a set of resource bindings for a given resource allocation, Hebe explores the design space to find a favorable implementation with respect to a particular design goal, such as minimal area or minimal latency. Any valid implementation must satisfy both resource and timing constraints.

A set of resource allocations $\{\alpha_1, \ldots, \alpha_k\}$ is specified either by the user manually or by the system automatically [1]. Hebe supports both *exact* and *heuristic* strategies to explore the design space; they are summarized below.

- *Exact design space exploration*: Exact exploration finds an optimum hardware implementation for a given design. This strategy synthesizes a logic-level implementation for each point in the design space. For many ASIC designs, this is an appropriate strategy because of the restricted size of the design space that stems from the few number of shareable operations and resources.
- *Heuristic design space exploration*: For designs with a large design space, exhaustive synthesis may be prohibitive. To address this difficulty, two heuristic strategies are supported by Hebe. The first strategy constructs only a portion of the design space and the second strategy evaluates and ranks the design space according to a set of cost criteria. The resource bindings with the most favorable cost are synthesized first to determine if they are valid under timing constraints.

Details of the design space exploration strategy are described in [10] and [12].

6. Resource conflict resolution

A resource binding implies a certain degree of hardware sharing. It is necessary in general to resolve resource conflicts present in the binding. Resource conflicts arise when two operations bound to the same resource execute in parallel. Most synthesis approaches formulate conflict resolution as a *scheduling* problem where operations are assigned to fixed control steps. Resource conflict occurs if two operations bound to the same resource are assigned to the same control step and they are not in mutually exclusive conditional branches. Consider for example the

[1] Note that an allocation of $\alpha_i(t)$ means that *exactly* $\alpha_i(t)$ resources are used. Therefore, allocating up to 3 resources is represented by the allocations $\{1, 2, 3\}$.

force-directed scheduling technique [14]. Operations with similar resources are first scheduled to reduce their concurrency, then they are bound to hardware resources subject to this schedule. The binding step ensures that no resource conflicts will arise. This approach is, however, restricted to bounded delay operations.

The support for *unbounded delay* operations in the sequencing graph model invalidates this formulation because operations can no longer be assigned to fixed control steps. Furthermore, detailed timing constraints impose bounds on the activation of operations. These constraints must be analyzed for consistency. To address these issues, the *relative scheduling* formulation [5] was developed in which operations are activated with respect to time offsets from the completion of a set of anchors, i.e. unbounded delay operations. Resource conflict resolution is formulated as the task of serializing the graph model so that operations bound to the same resource cannot execute in parallel.

This section presents a technique called *constrained conflict resolution* that takes as input as sequencing graph with timing constraints and a resource binding. It serializes the sequencing graph to resolve the resource conflicts such that the timing constraints are still satisfied after the serialization. In addition to the support for unbounded delay operations and detailed timing constraints, this technique uses the topology of the timing constraints to improve the computation time of the resolution algorithm. Resource sharing among mutually exclusive conditional branches is also supported in this formulation. Once the graph model has been appropriately serialized, relative scheduling is performed and the corresponding control logic is generated. If the resource conflicts cannot be resolved under timing constraints, then another resource binding is selected as candidate for synthesis. The consistency of timing constraints is based on the relative scheduling formulation.

6.1. Review of relative scheduling

Before describing the conflict resolution strategy, we first briefly describe the main results in relative scheduling as necessary background for the conflict resolution formulation. The interested reader is referred to [5] for further details. Given a constraint graph $G(V, E, \omega)$, we use the set of anchors A as reference points for specifying the start times of the operations, where the anchors consist of the source vertex v_0 and the set of unbounded delay vertices in G. We define the **anchor set** $A(v_i)$ of a vertex v_i as the set of anchors that are predecessors to the vertex, representing the *unknown* factors that affect the activation time of the vertex. In particular, for each anchor $a \in A(v_i)$, the following condition holds for all values of unbounded delay $\delta(a)$: $\text{length}(a, v_i) \geqslant \delta(a)$. The **start time** $T(v_i)$ of a vertex v_i is then generalized in terms of fixed time offsets $\sigma_a(v_i)$ from the completion of each anchor $a \in A(v_i)$ in its anchor set. The expression for the start time $T(v_i)$ is defined recursively as follows:

$$T(v_i) = \max_{a \in A(v_i)} \{T(a) + \delta(a) + \sigma_a(v_i)\}$$

A *minimum relative schedule* of a constraint graph is the set of minimum offsets for all vertices of the graph.

An important consideration during scheduling is whether the timing constraints can be satisfied for any value of the unbounded delay operations. We have introduced the concept of *feasible* and *well-posed* constraints in the presence of unbounded delays [5]. A constraint graph is feasible if the constraints are consistent assuming all unbounded delays are set to zero. A constraint graph is well-posed if the constraints are satisfied for all values of unbounded delays. A relative schedule is guaranteed to exists if and only if the graph is well-posed. We state without proof in this paper that a constraint graph is well-posed if and only if (i) it is feasible, and (ii) no unbounded length cycle exists in the graph [5]. 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 conflict resolution.

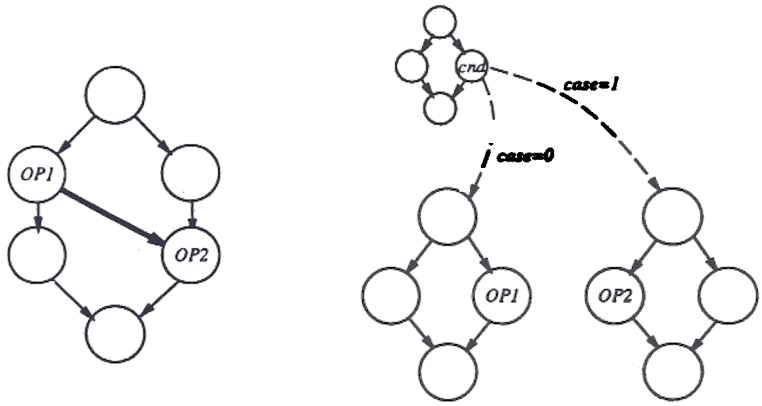
6.2. Conflict resolution formulation

A resource binding is valid if it is possible to resolve its resource conflicts and still satisfy the required timing constraints. For a given resource binding β, recall that an *instance operation set* $O_{(t,i)}$ of β is a subset of vertices that are bound to an allocated resource instance (t, i). Obviously, resource conflicts will occur if the vertices in $O_{(t,i)}$ can execute in parallel. An *implementable* binding is defined as follows.

Definition 6.1. *An instance operation set $O_{(t,i)}$ is* **implementable** *if the elements of $O_{(t,i)}$ are disjoint in time, i.e. they do not execute concurrently. Given a binding β of a constraint graph $G(V, E)$, G is* **implementable** *if every instance operation set in β is implementable.*

Two operations op1 and op2 are disjoint in time if one of the two following conditions holds: (1) op1 is serialized with respect to op2 in the graph, such that op1 can execute only if op2 has completed execution or vice versa, and (2) op1 and op2 each belong to different mutually exclusive branches of a conditional. The two cases are illustrated in Fig. 8. Since the conditional branching structure cannot be arbitrarily altered without changing the algorithmic flow of the model, we resolve resource conflicts by serializing operations. The example in Fig. 8 illustrates the hierarchical control-flow of the sequencing graph model. In particular, elements of an instance operation set $O_{(t,i)}$ may not all belong to the same sequencing graph.

To address this issue, conflict resolution for an instance operation set $O_{(t,i)}$ in a sequencing graph G_M is performed hierarchically in a bottom-up manner. A *candidate operation* set $O_{(t,i)}(G)$ for each graph G in the cf-hierarchy G_M^* is identified as candidates to be serialized. A vertex v is a candidate if v belongs to the instance operation set $O_{(t,i)}$ or if one or more elements of $O_{(t,i)}$ belong to graphs in the cf-hierarchy induced by v.



OP1 serialized w.r.t. OP2 (a)

OP1 and OP2 mutually exclusive (b)

Fig. 8. Two cases when op1 and op2 are *implementable*: (a) when they are serialized with each other, or (b) when they reside in mutually exclusive conditional branches.

We consider in the rest of this section a single constraint graph G that is derived from a sequencing graph with timing constraints, where conflict resolution has been performed on all graphs in its cf-hierarchy G^*. Therefore, the term "instance operation set" in the sequel refers to the candidate operation set of $O_{(t,i)}$ with respect to G. The objective in conflict resolution is to resolve the conflicts among elements of the set $O_{(t,i)}(G)$. An *ordering* of the instance operation set is defined as follows.

Definition 6.2. *An* **ordering** *of an instance operation set* $O_{(t,i)}(G)$, *denoted by* $\langle o_1, o_2, \ldots, o_k \rangle$ *where* $k = |O_{(t,i)}(G)|$, *is a serialization of the vertices of* $O_{(t,i)}(G)$ *in* $G(V, E)$ *such that in the resulting constraint graph,* o_j *is a predecessor to* o_{j+1}, $1 \leqslant j \leqslant k-1$.

The activation of the vertex $o_j \in O_{(t,i)}$ in an ordering must depend on the *completion* of the preceding vertex $o_{j-1} \in O_{(t,i)}$ in the ordering. An ordering for an instance operation set $O_{(t,i)}(G)$ is a sufficient condition to ensure that $O_{(t,i)}(G)$ is implementable. It is a *valid ordering* if the resulting serialized graph G satisfies the timing constraints, i.e. the graph is well-posed [5].

6.3. Constraint topology

This section analyzes the topology of timing constraints in a constraint graph $G(V, E)$. We describe several concepts that are used in the conflict resolution formulation. Let the target instance operation set $O_{(t,i)}(G)$ be denoted by $O \subseteq V$, where we dropped the terms (t, i) and G for conciseness. The instance operation set O consists of $k = |O|$ vertices, denoted by o_i, $i = 1, \ldots, k$. Each vertex $o_i \in O$ has an associated execution delay $\delta(o_i)$ that can be bounded or unbounded. In

the simplistic case of flat graphs, all elements of O are call vertices to the same model; therefore, they have identical execution delays. However, for hierarchical graphs, unequal execution delays may result.

A cycle in the constraint graph represents a cyclic timing relationship among a set of vertices. It has been shown that a violation of timing constraints can occur if the constraint graph is *unfeasible* or if the constraint graph is *ill-posed*. A constraint graph is feasible if and only if no positive cycle exists in G assuming unbounded delays are set to zero; it can be made well-posed if and only if no unbounded length cycles exist in G. In both cases, timing constraint violation occurs in the presence of cycles in the graph model. Based on this observation, we partition the elements of the instance operation set by introducing the concept of operation clusters, defined below.

Definition 6.3. *An* **operation cluster** $\mathscr{C}$ *of an instance operation set O is the maximal subset of vertices in O that is strongly connected, i.e. there exists a directed path between every pair of vertices in the operation cluster.* $|\mathscr{C}|$ *denotes the cardinality of* $\mathscr{C}$.

Theorem 6.1. *A partial order exists among the operations clusters of an operation set.*

Proof. Elements of an instance operation set are strongly connected in the constraint graph. Since strong connectivity is an equivalence relation, two operation clusters cannot be connected by a cycle. This is the definition of partial order. □

The set of operation clusters is denoted by $\Pi = \{\mathscr{C}_i, i = 1, \ldots, |\Pi|\}$, where $|\Pi|$ is the number of operation clusters in O. The operation clusters form a partition over the elements of O because the property of strong connectivity is an equivalence relation. We illustrate the concept in Fig. 9, where the dotted arcs represent backward edges with negative weights and the solid arcs represent forward edges with positive weights. There are two operation clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E\}$ in the example. A partial order is formed over the two clusters, i.e. from C_1 to C_2.

This partial order over the operation clusters provides the basis for a conflict resolution strategy based on decomposition. Specifically, the problem of finding a valid ordering for an instance operation set is divided into two steps:

- *Ordering among the operation clusters*: Find a linear order of operation clusters that is compatible with the induced partial order in Π, and
- *Ordering within each operation cluster*: Find a valid ordering for the vertices within each operation cluster.

We state the following theorem.

Theorem 6.2. *If valid orderings exist for the vertices inside each operation cluster* $\mathscr{C}_i \in \Pi$, $i = 1, \ldots, |\Pi|$, *then any ordering of operation clusters that is compatible with the partial order induced by Π is a valid ordering for O.*

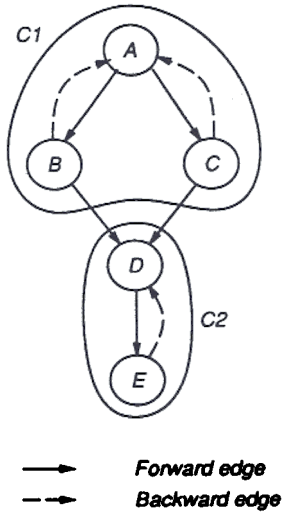


Fig. 9. Example of an instance operation set with 5 vertices $\{A, B, C, D, E\}$. Two operations clusters are formed: $C_1 = \{A, B, C\}$ and $C_2 = \{D, E\}$.

Proof. Assume each operation cluster has a valid ordering. Since clusters are not connected by a cycle, the serialization of one cluster does not affect any cyclic constraints of the other operation clusters. Since each cluster is ordered and no constraints are violated by ordering among the clusters, the resulting ordering is valid for the entire instance operation set. □

With Theorem 6.2, the problem of finding a valid ordering for an instance operation set O has been reduced to the problem of finding a valid ordering for the elements of an operation cluster $\mathscr{C}_i \in \Pi$. The formation of operation clusters is strongly dependent on the extent to which operations are related by timing constraints. By linking the complexity of the resolution effort with the complexity of operation clusters, we take advantage of the *topology* of constraints in finding an efficient serialization.

6.4. Orientation and polarization

We introduce in this section the concepts of *orientation* and *polarization* of an operation cluster. For conciseness, we consider one operation cluster $\mathscr{C}$ that contains $|\mathscr{C}|$ vertices, i.e. $\mathscr{C} = \{c_i \mid i = 1, \ldots, |\mathscr{C}|\}$.

We make the following assumptions. First, the cardinality of the operation cluster must be greater than one ($|\mathscr{C}| > 1$), since otherwise the ordering is trivial. Second, each vertex $c_i \in \mathscr{C}$ must either be an unbounded delay operation (i.e. anchor) or have non-zero bounded execution delay, i.e. $\delta(c_i) > 0$. Note that registers have already been introduced prior to conflict resolution to latch the outputs of the shared resource. For example, the execution delay for shared calls to a combinational adder is 1 cycle because of the latching delay.

$c_i \in \mathscr{C}$ is a predecessor to another vertex $c_j \in \mathscr{C}$ if there exists a pair $(c_i, c_j) \in \mathscr{P}_\varphi$; successors are defined in a similar manner.

Based on this predecessor-successor relation, the **leaves (roots)** of an orientation $\mathscr{P}_\varphi$ are the subset of elements of $\mathscr{C}$ that have no successors (predecessors). They are denoted by $\mathscr{P}_\varphi^{\text{leaf}}$ and $\mathscr{P}_\varphi^{\text{root}}$, respectively. Returning to the example in Fig. 10, the roots of the orientation are $\{A, B\}$ and the leaves are $\{D, E\}$.

6.4.2. Polarization

It is straightforward to show that any valid ordering of $\mathscr{C}$ must be compatible with the partial order induced by the orientation $\mathscr{P}_\varphi$. This observation implies that the first element of any valid ordering must be a root, and similarly the last element must be a leaf. For a root–leaf pair (r, l): $r \in \mathscr{P}_\varphi^{\text{root}}$, $l \in \mathscr{P}_\varphi^{\text{leaf}}$, $r \neq l$, we can make the orientation *polar* (single-source and single-sink) by serializing from r to all other vertices and from all vertices to l. We formalize this observation in defining a *polarization*.

Definition 6.5. *A* **simple polarization** *with respect to* $r \in \mathscr{P}_\varphi^{\text{root}}$ *and* $l \in \mathscr{P}_\varphi^{\text{leaf}}$ *of an orientation* $\mathscr{P}_\varphi$, $r \neq 1$, *denoted by* $\mathscr{P}_\varphi(r, l)$, *is the relation that is derived from union of* $\mathscr{P}_\varphi$ *with the relations* (r, v), $\forall v \neq r$ *and* (w, l), $\forall w \neq l$. *An* **(extended) polarization**, *denoted by* $\mathscr{P}_\varphi^*(r, l)$, *is* $\mathscr{P}_\varphi(r, l)$ *extended with the all pairs* (v, w) *such that* $\text{length}(w, v) + \delta(v) > 0$.

The reason for disallowing the serialization from w to v if the condition $\text{length}(w, v) + \delta(v) > 0$ holds is to avoid creating a positive cycle. Figure 11 shows an operation cluster of 5 vertices $\{v_1, v_2, v_3, v_4, v_5\}$. Vertices v_2, v_3 and v_4 are connected with one another by negatively weighted edges representing maximum timing constraints, i.e. $w_{v_2,v_3} = -3$ means that v_2 can start no more than 3 cycles after the activation of v_3. The orientation $\mathscr{P}$ is the subgraph induced by the positive weighted edges, where the roots consist of $\mathscr{P}^{\text{root}} = \{v_1, v_2, v_3, v_4\}$ and the leaves consist of $\mathscr{P}^{\text{leaf}} = \{v_1, v_3, v_5\}$. A simple polarization $\mathscr{P}(v_1, v_5)$ adds edges from v_1 to all remaining vertices and from all non-leaf vertices to v_5. The extended polarization $\mathscr{P}^*(v_1, v_5)$ considers in addition the value of the negatively weighted edges. For example, $(v_2, v_3) \in \mathscr{P}^*(v_1, v_5)$ becuase a positive cycle is formed if v_3 with execution delay of 4 is serialized to v_2.

Theorem 6.3. *If a cycle exists in a polarization* $\mathscr{P}_\varphi^*(r, l)$, *then no valid ordering exists that is compatible with the polarization.*

Proof. A pair (x, y) in the polarization implies a precedence relationship between x and y, i.e. x must be serialized *before* y. Assume the presence of a cycle in the graph, denoted by $(x, y_1), (y_1, y_2), \ldots, (y_k, x)$. By transitivity of the precedence relationship, the cycle implies that x must be serialized with respect to x, which is inconsistent. Therefore, since any serialization must be compatible with the polarization, no valid ordering exists if a cycle exists in the graph. □

Any valid ordering of an operation cluster must be compatible with one of its polarizations. There is a finite number of polarizations for a given orientation. The total number of possible polarizations for an orientation $\mathcal{P}_{\mathcal{G}}$ is given by the expression:

$$\#\text{polarization} = \left|\mathcal{P}_{\mathcal{G}}^{\text{root}}\right| \cdot \left|\mathcal{P}_{\mathcal{G}}^{\text{leaf}}\right| \quad \left|\mathcal{P}_{\mathcal{G}}^{\text{root}} \cap \mathcal{P}_{\mathcal{G}}^{\text{leaf}}\right|$$

where the $|\mathcal{P}_{\mathcal{G}}^{\text{root}} \cap \mathcal{P}_{\mathcal{G}}^{\text{leaf}}|$ term corresponds to the isolated vertices in the orientation.

Figure 12 shows an operation cluster with 5 vertices. The bold arcs are due to the orientation and the shaded vertices denote root and leaf vertices in a polarization. There are $2 \cdot 2 - 0 = 4$ polarizations for this cluster. The concept of polarizations allows us to prune the search for a valid ordering. Since the simple polarization $\mathcal{P}_{\mathcal{G}}(r, l)$ is a restriction of the polarization $\mathcal{P}_{\mathcal{G}}^{*}(r, l)$, we use strictly $\mathcal{P}_{\mathcal{G}}^{*}(r, l)$ in the rest of the section.

6.5. Properties of polarizations

This section describes two theorems related to polarizations that are used as filters to speed the search for a valid ordering. The first theorem is related to the presence and position of anchors in a given polarization.

Theorem 6.4. *For a polarization $\mathcal{P}_{\mathcal{G}}^{*}(r, l)$, if any non-leaf vertices is $\mathcal{G}$ in an anchor, then no valid ordering exists for the polarization.*

Proof. Assume there exists a non-leaf vertex $v \neq l$ with unbounded execution delay. A valid ordering of $\mathcal{P}_{\mathcal{G}}^{*}(r, l)$ implies that v must be serialized with respect to l. Since v has unbounded execution delay, the serialization requires introducing an edge with unbounded weight to the constraint graph. The vertices in $\mathcal{G}$ are however strongly connected, meaning that an unbounded length cycle has been formed. This means the timing constraints cannot be satisfied, and no valid ordering exists. □

The following theorem provides an effective and exact pruning measure that is used in the exact conflict resolution algorithm, described in Section 6.6.2. The theorem states that the sum of the execution delays of the vertices to be serialized ($\sum_{v \in \mathcal{G}, v \neq l} \delta(v)$) must not exceed the allowed maximum timing constraint from l to r, i.e. length(l, r).

Theorem 6.5. *Consider a polarization $\mathcal{P}_{\mathcal{G}}^{*}(r, l)$. If the following condition holds:*

$$\text{length}(l, r) + \sum_{v \in \mathcal{G}, v \neq l} \delta(v) > 0$$

then no valid ordering exists for the polarization.

Proof. A valid ordering within an operation cluster implies that all vertices are serialized to form a chain. Given a polarization (r, l), r is the first element of the

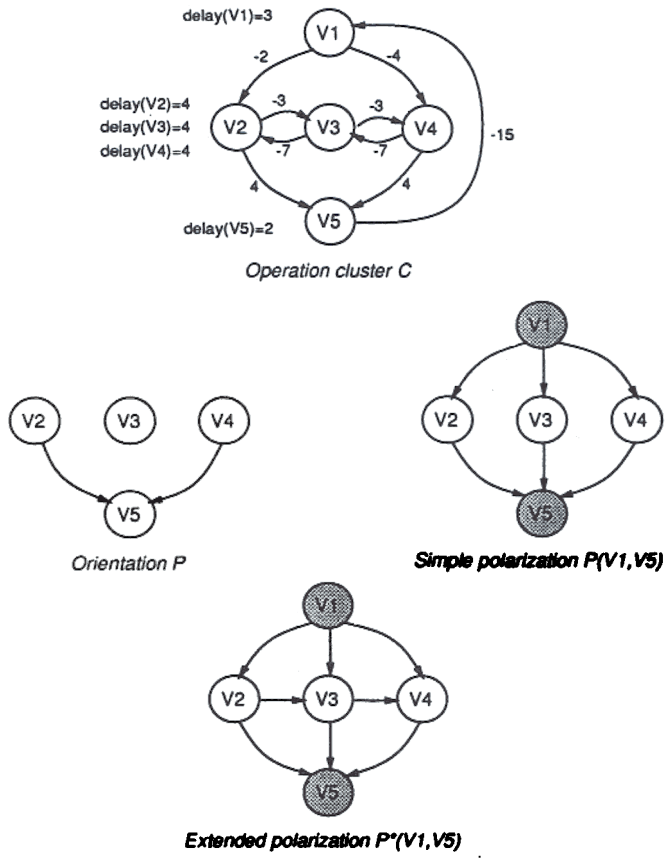


Fig. 11. Illustrating an operation cluster and its orientation $\mathcal{P}$, a simple polarization $\mathcal{P}(v_1, v_5)$, and the extended polarization $\mathcal{P}^*(v_1, v_5)$.

chain and l is the last element of the chain. The minimum length of such a chain is equal to the sum of the execution delays of the vertices excluding the leaf l, i.e. $\Sigma_{v \in \mathcal{C}, v \neq l} \delta(v)$. A necessary condition for a valid ordering is that no positive cycles are formed in the resulting constraint graph. Consider the cycle formed by the chain and the backward path from l to r, the length of the latter is denoted by length(l, r). If the cycle has positive length, then the resulting graph is invalid and no valid ordering exists. □

6.6. *Algorithms for conflict resolution*

Two algorithms for conflict resolution are presented in this section. The input is a resource binding β consisting of a number of instance operation sets. The instance operation sets in β are selected in turn. For a given instance operation set O, its operation clusters are first identified using standard graph techniques such as cycle detection or path tracing. The following steps are then performed for each operation cluster $\mathcal{C}_i$ in O:

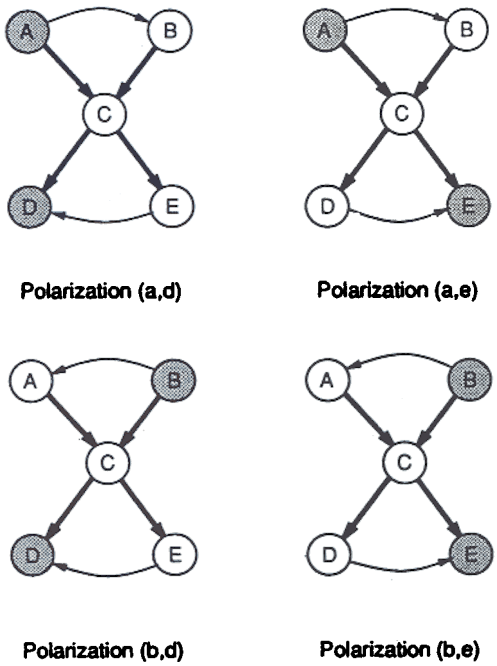


Fig. 12. Four possible polarizations for an operation cluster where bold arcs represent the orientation.

1. *Identify the orientation* $\mathscr{P}_{\varphi}$. The orientation is obtained by categorizing the edges based on the sign of their weights. The roots $\mathscr{P}_{\varphi}^{\text{root}}$ and leaves $\mathscr{P}_{\varphi}^{\text{leaf}}$ of the orientation are identified.
2. *Select a polarization* $\mathscr{P}_{\varphi}^{*}(r, l)$. A particular polarization with root r and leaf l is selected. If a cycle exists in the polarization or if the polarization violates the condition in Theorem 6.4, then it is discarded and another polarization is selected. If all polarizations are invalid, then the resource conflicts for the given operation cluster cannot be resolved under timing constraints.
3. *Apply heuristic ordering algorithm.* A polynomial-time complexity heuristic algorithm is performed to find a valid ordering with the goal of minimizing the latency of the resulting hardware. If a solution is found, another cluster is selected as candidate and the steps are repeated until all clusters have been ordered. Otherwise, the exact ordering algorithm in the next step is performed.
4. *Apply exact ordering algorithm.* A branch-and-bound ordering algorithm is applied if the heuristic fails to find a solution. The exact algorithm is guaranteed to find a solution if one exists. Theorem 6.5 is used in the cost function to prune the branch-and-bound search.

For designs with a large number of possible orderings, the exact ordering algorithm (Step 4) can be skipped. In this case, no guarantee on the existence of a solution is possible if the heuristic (Step 3) fails.

After the operations within each clusters have been serialized, the clusters are linearly ordered compatible with the original partial order. This linear order can

```
  current = l; / * construct ordering upwards * /
  while (unordered candidates exist) {
    Candid = compatible(Ord);
    / * select most constrained candidate * /
    v_y = Select arg min_{z ∈ Candid}{ f(z)};
    Add v_y to the ordering Ord and serialize graph;
    Recompute all-pairs longest path;
    / * check timing constraints * /
    if (positive cycle formed)
      return no valid ordering found;
    current = v_y;
  }
  return valid ordering Ord;
}
```

serializing v_y with respect to current, the length of the longest path between them is the maximum of the length $\delta(v_y)$ of the serializing edge and the previous longest path length. Note that by definition of clusters the longest path is defined between every pair of vertices in a cluster.

Intuitively, the slack is a measure of the length of the longest cycle that would be formed if v_y is selected and serialized as the next element in the ordering Ord. It must always be positive since otherwise the serialization is not valid. Among the possible candidates, the one with *minimum* slack is selected as the next element in the ordering. The procedure to incrementally construct an ordering $\text{Ord} = \langle \ldots \rangle$ is described in the heuristic ordering Procedure Heuristic-order.

The routine compatible(Ord) returns a set of candidates with respect to a partial ordering Ord. To define it, we first augment the original polarization $\mathscr{P}^*_{\mathscr{G}}(r, l)$ with the partial ordering $\text{Ord} = \langle \text{Ord}_i, \ldots, \text{Ord}_{|\mathscr{G}|} \rangle$ by adding the relations $\{(\text{Ord}_j, \text{Ord}_{j+1}),\ i \leqslant j \leqslant |\mathscr{G}| - 1\}$ to $\mathscr{P}^*_{\mathscr{G}}(r, l)$. An unordered element v_c is in compatible(Ord) if there exists no other candidate $w_c \in$ compatible(Ord) such that the relation (v_c, w_c) is in the augmented polarization.

At each iteration of the loop, the graph is serialized with respect to the constructed partial ordering. This ordering is constructed incrementally until the root r is reached. At each iteration, the serialized graph is checked for consistency. Consistency analysis involves computing the longest path lengths between pairs of operations, which using Floyd's algorithm requires complexity $O(|\mathscr{G}|^3)$. Therefore, the overall procedure has $O(|\mathscr{G}|^4)$ complexity.

Example. We illustrate the application of procedure *Heuristic_order* in Fig. 13 to an operation cluster consists of 7 vertices $\{v_1, \ldots, v_7\}$, starting with the polarization $\mathscr{P}^*(v_1, v_7)$. The partial order Ord is constructed from the leaf v_7 upwards to the root v_1. At step 1, the candidates based on the partial order of the

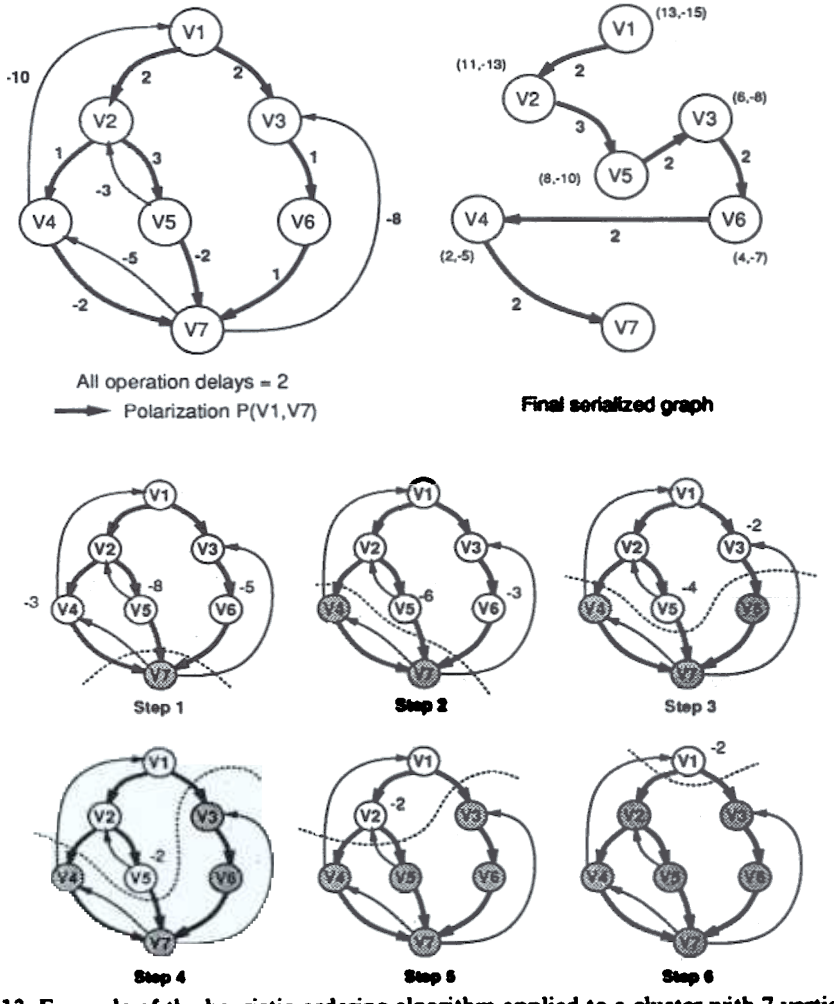


Fig. 13. Example of the heuristic ordering algorithm applied to a cluster with 7 vertices.

polarization (represented by bold arcs) are $\{v_4, v_5, v_6\}$. The slacks for these candidates are:

$$\int(v_4) = -(\max(2, -2) + 0 + (-5)) = 3$$

$$\int(v_5) = -(\max(2, -2) + 0 + (-10)) = 8$$

$$\int(v_6) = -(\max(2, 1) + 0 + (-7)) = 5$$

Vertex v_4 has minimum slack and hence is added to the partial ordering

Ord = $\langle v_4, v_7 \rangle$. The graph is serialized accordingly. At step 2, the candidates are $\{v_5, v_6\}$. The slacks for these candidates are:

$$\int(v_5) = -(\max(2, -2) + 2 + (-10)) = 6$$

$$\int(v_6) = -(\max(2, -4) + 2 + (-7)) = 3$$

Vertex v_6 has minimum slack and is added to Ord = $\langle v_6, v_4, v_7 \rangle$. The algorithm repeats until the root vertex v_1 is reached. The final order $\langle v_1, v_2, v_5, v_3, v_6, v_4, v_7 \rangle$ results in a valid constraint graph.

6.6.2. Exact ordering search

The heuristic ordering strategy in the previous section may fail to find a valid ordering in some cases. This section presents an *exact* ordering algorithm based on branch-and-bound called *Exact_order* that is performed to find a valid ordering for a given polarization $\mathscr{P}^*_{\mathscr{G}}(r, l)$. If a valid ordering is not found for this polarization, the algorithm is applied to another polarization. If a valid ordering is not found for any polarization, then it is not possible to resolve the conficts in the operation cluster $\mathscr{G}$.

This recursive algorithm constructs an ordering incrementally, starting from the leaf l of the polarization. The partial ordering that is being constructed is denoted by Ord = $\langle \text{Ord}_i, \ldots, \text{Ord}_{|\mathscr{G}|} \rangle$; the index i is the index of the current element in the partial ordering, $1 \leqslant i \leqslant |\mathscr{G}|$. Note that the first and last elements of Ord are the root and leaf vertices, respectively. The procedure is described in the exact ordering algorithm *Exact_order*, where the routine compatible(Ord) is the same as in the previous section.

The ordering is complete when $i = 1$, whereupon the procedure records the valid ordering Ord and returns true. Otherwise, one of the candidates in the set returned by compatible(Ord) is added to Ord. For each candidate, pruning is performed to filter out candidates that will violate timing constraints. The pruning strategy is based on defining for the subsequence $\langle \text{Ord}_i, \ldots, \text{Ord}_{|\mathscr{G}|} \rangle$ a cost function, denoted by cost(Ord_i) representing a bound on the *length of the longest path* from the root vertex r to the leaf vertex l, assuming the partial ordering is applied. Specifically, the cost function for a subsequence $\langle \text{Ord}_i, \ldots, \text{Ord}_{|\mathscr{G}|} \rangle$ is given as follows:

$$\text{cost}(\text{Ord}_i) = \left\{ \sum_{v \in \mathscr{G} \text{ s.t. } v \notin \text{Ord}} \delta(v) \right\} + \text{length}(\text{Ord}_i, l)$$

The first term is a lower bound to the longest path length of the remaining unordered vertices after they have been serialized. The second term length(Ord_i, l) represents the longest path length from the first element Ord_i of the subsequence to the leaf l. Theorem 6.5 guarantees that cost(Ord_i) is always a lower bound to length(r, l) in the serialized graph. The branch-and-bound strategy terminates when the first valid ordering is found.

```
Procedure Exact_order(partial ordering Ord, current index i)
{
  if (i = 1){
    Record valid orderig Ord;
    return TRUE;
  }
  / * try each compatible candidates * /
  foreach (z ∈ Compatible(Ord) {
    Append z to the ordering Ord;
    / * prune based on cost * /
    if (cost(Ord_i) + length(l, r) < 0) {
      Serialize graph subject to Ord;
      if (resulting graph valid)
        if (Exact_order(Ord, i − 1) = TRUE)
          return TRUE;
    }
  }
  / * backtrack * /
  return FALSE;
}
```

7. Implementation and design experiences

Hercules and *Hebe* have been implemented in C, with approximately 140000 lines of code. Several digital ASIC designs were synthesized using this system, including an Ethernet controller [17], a digital audio input–output chip [18], and a bi-dimensional discrete cosine transform chip [19]. The functionality and synthesis results are summarized below.

- Ethernet controller—manages transmitting and receiving data frames over a network under CSMA/CD protocol. The purpose is to off-load the host processor from managing the communication activities. Its capabilities include data framing and deframing, network and link operations, address sensing, error recovery, data encoding, direct memory access, and collision detection. The entire design is modeled by 13 concurrent processes, described in over 1200 lines of HardwareC code. Port read and write, as well as message passing send/receive commands, are used extensively in the design to specify handshaking protocols. The logic-level implementation was mapped to 11000 complex gates in LSI Logic's LCA10K library. The controller was designed for operation frequency of 20 MHz.
- Digital Audio input/output (DAIO) ship—controls the transfer of data between a microprocessor and a compact disc player or a digital audio tape player. Bit-serial synchronous line transmission is defined by the Audio Engineering Standard (AES) protocol. The design is described in 650 lines of

HardwareC code. The resulting implementation was 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.

- Bi-dimensional Discrete Cosine Transform (BDCT) chip—performs coding to remove redundant video information in low bit-rate transmission channels and video compression for image storage and retrieval. An 8×8 BDCT architecture was synthesized by Hercules and implemented in a compiled macro-cell design style [20] as a 9×9 mm^2 image in $2\mu_m$ CMOS technology.

Each design was described completely in HardwareC and synthesized to a gate-level implementation. Extensive logic-level simulation demonstrated the correctness of the specification and implementation. Other ASIC designs include a Multi-anode Microchannel array (MAMA) detector for the space telescope [21], a pixel line drawing design, and an error correcting code design [12].

In addition, the system has been applied to the synthesis of benchmark circuits from the High-Level Synthesis Workshop [22]. Most of these examples have been rewritten in HardwareC and synthesized to logic-level implementations. Three widely used and compared examples are the example used in Facet (Tseng) [23], the differential equation solver (Diffeq) [24], and the 5th-order elliptic waveform filter (Elliptic) [25]. Although these examples do not contain detailed synchronization and timing constraints, they serve to demonstrate the use of our approach on general synchronous designs. The statistics on the SIF models of these three benchmark designs are given in Table 2. The size of the multiplication is double the size of the addition in these examples. For example, Elliptic requires 32-bit multiplications and 16-bit additions. To evaluate and compare the reuslts of resource binding and scheduling in Hebe with existing systems, we make the following assumptions: additions (both 8-bit and 16-bit) and subtractions take 1 cycle to execute, and multiplications (both 8-bit and 16-bit) and divisors take 2 cycles to execute, multiplications are non-pipelined, and no detailed timing constraints are present.

For the elliptic filter example, the filter coefficients are arbitrary 16-bit values and not necessarily powers of two. Therefore, multiplication with these coefficients are implemented as a full multiplication with a set of 16-bit wide coefficient registers instead of shift registers. Our results are compared with the force-directed scheduling (FDS) ad force-directed list scheduling (FDLS) in HAL

Table 2
SIF model statistics for the benchmark examples

Design	Lines		Nodes	Graphs	Calls				Size
	HC	SIF			add	sub	mul	div	
Tseng			11	1					8/16-bit
Diffeq			17	2					8/16-bit
Elliptic			37	1					16/32-bit

Table 3
Comparison of schedule latency for the Elliptic filter example

Resources		Latency (cycles)					
add_{16}	mul_{32}	FDS	FDLS	CSTEP	SLICER	PBS	Hebe
		–	–	–	–	–	
		19	18	19	19	18	
		–	–	–	–	–	
		18	–	–	18	18	
		17	17	17	–	17	

[26], the CSTEP scheduler (CSTEP) [27], SLICER in Chippe (SLICER) [28], percolation based scheduling (PBS) [29]. These results are based on the original Elliptic specification that is distributed in the benchmark suite. Tables 3 summarize the comparison of the schedule latency for different resource allocations in the Elliptic example. The schedule latency is known because there are no unbounded delay operations in the design. A dash (–) in the table means that synthesis result of the corresponding synthesis system is not available in the literature.

Tables 4 summarizes the synthesis results for Tseng's example and the Diffeq example. Comparison with other systems is not available because the published results are in terms of operator cell area. For example, HAL uses relative operator cell area to describe the area cost [24]. The heuristic design space exploration and conflict resolution strategy was used to obtain these results. Note that the latency can be improved if expression tree height reduction is performed during behavioral synthesis. However, this optimization is not implemented in the current version of Hercules.

We now present synthesis results of the Tseng, Diffeq, and Elliptic examples for different allocations of resources, where each logic-level implementation is mapped to LSI Logic's LCA10K library using the technology mapper *Ceres*. Instead of assuming the availability of a 2 cycle multiplier from a given micro-ar-

Table 4
Summary of schedule results for Tseng and Diffeq examples

Design	Resources	Latency
	1*, 1/	
	1*, 1/	
	1*, 1/	
Diffeq	1–	
	1–	
	1–	
	1–	
	1–	
	1–	

Table 5
Synthesis results for the arithmetic library modulus

Module	Size	Latency	Register	Area	Delay
Subtract					
Multiply					
	32 bits				

chitecture library, we have designed and implemented an efficient multiplier unit starting from a HardwareC description. Tradeoffs can be performed on this multiplier just as with other designs. Table 5 gives the statistics on the area and delay costs for arithmetic library modules. Note that a 32-bit multiplier taking two 16-bit operands is significantly larger in area than a 16-bit adder. This is contrary to the assumption made by most synthesis systems that a multiplier is 4 times as large as an adder.

Based on these library resources, the implementation results for different allocations of Tseng, Diffeq, and Elliptic are presented in Table 6. For Elliptic, the 4 cycle 32-bit multiplier was used to implement its multiplication. For Tseng and Diffeq, the 4-cycle 16-bit multiply was used. Therefore, each multiplication and latch operation requires 5 cycles to execute. Combinational logic optimization was not performed on the logic-level implementation prior to technology mapping due to excessive memory usage of MisII [30].

Table 6
Synthesis results for the benchmark examples, assuming 5 cycle 16-bit and 32-bit multiply (4 cycle multiply +1 cycle latch)

Benchmark	LSI Implementation			
	Schedule	Reg	Area	Delay
Elliptic (1 *, 1 +)				
Elliptic (2 *, 1 +)				
Elliptic (2 *, 2 +)				
Tseng (1 +, 1 −, 1 *, 1/)				
Tseng (2 +, 1 −, 1 *, 1/)				
Tseng (3 +, 1 −, 1 *, 1/)				
Diffeq (1 +, 1 −, 1 *)				
Diffeq (1 +, 1 −, 2 *)				

8. Conclusion and future work

Hebe is a system that supports the synthesis of synchronous digital ASIC designs starting from behavioral level specifications. The underlying hardware model is a sequencing graph abstraction that supports concurrency, external synchronizations in the form of unbounded delay operations, and detailed timing constraints. Algorithms were presented to resolve resource conflicts for a given resource binding under timing constraints. The algorithms take advantage of graph properties in pruning the search space. The system has been applied to the design of benchmarks and some ASIC designs.

Future work includes optimizing control under timing and area constraints. This is based on the observation that control is often an important component in the overall hardware cost. Extensions to consider synthesis of multiple processes are also important in supporting system level designs.

Acknowledgements

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

References

[1] M. McFarland, A. Parker and R. Camposano, The high-level synthesis of digital systems, *Proc. IEEE* **78** (2) (Feb. 1990).

[2] A. de Geus, Logic synthesis speeds ASIC designs, *IEEE Spectrum Magazine* **26** (8) (Aug. 1989) 27–31.

[3] G. De Micheli and D.C. Ku, HERCULES—a system for high-level synthesis, *Proc. Design Automation Conf.* (June 1988) 483–488.

[4] D.C. Ku and G. De Micheli, High-level synthesis and optimization strategies in Hercules and Hebe, *Proc. European ASIC Conf.*, Paris, France, May 1990.

[5] D. Ku and G. De Micheli, Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits, CSL Technical Report CSL-TR-477, Stanford, June 1991.

[6] R. Camposano and W. Rosenstiel, Synthesizing circuits from behavioral descriptions, *IEEE Trans. CAD/ICAS* **8** (2) (Feb. 1989) 171–180.

[7] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan and R. Blackburn, *Algorithmic and Register-Transfer Level: The System Architect's Workbench* (Kluwer Academic Publishers, Dordrecht, the Netherlands, 1990).

[8] J. Nestor and G. Krishnamoorthy, "SALSA: A new approach to scheduling with timing constraints," in *Proc. Int. Conf. Computer-Aided Design* (Nov. 1990) 262–265.

[9] D.C. Ku and G. De Micheli, HardwareC—a language for hardware design (version 2.0), CSL Technical Report, Stanford, Apr. 1990.

[10] G. De Micheli, D.C. Ku, F. Mailhot and T. Truong, The Olympus Synthesis System for digital design, *IEEE Design and Test Magazine* (Oct. 1990) 37–53.

[11] R. Camposano and W. Wolf (Ed.), *High-Level VLSI Synthesis* (Kluwer Academic Publishers, Dordrecht, the Netherlands, June 1991).
[12] D.C. Ku, Constrained synthesis and optimization of digital integrated circuits from behavioral specifications, CSL Technical Report (Disseration) CSL-TR-91-476, Stanford, June 1991.
[13] M.J. McFarland, Reevaluating the design space for register-transfer hardware synthesis, *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA (Nov. 1987).
[14] P. Paulin and J. Knight, Force-directed scheduling for the behavioral synthesis of ASICs, *IEEE Trans. CAD/ICAS* (June 1989) 661–679.
[15] M. Garey and D. Johnson, *Computers and Intractability* (Freeman, New York, 1979).
[16] S. French, *Sequencing and Scheduling; Introduction to the Mathematics of the Job Shop* (Ellis Horwood, Chichester, UK, 1982).
[17] R. Gupta and C. Coelho, Ethernet controller design, private communication, 1991.
[18] M. Ligthart, A. Bechtolsheim, G. De Micheli and A.E. Gamal, Design of a Digital Audio Input Output chip, *Proc. Custom Integrated Circuits Conf.* (May 1989) 15.1.1–15.1.6.
[19] V. Rampa and G. De Micheli, The Bi-dimensional DCT chip, *Proc. Int. Symposium on Circuits and Systems* (May 1989) 220–225.
[20] F. Mailhot and G. De Micheli, Automatic layout and optimization of static CMOS cells, *Proc. Int. Conf. Computer Design* (Oct. 1988) 180–185.
[21] D.B. Kasle, High resolution decoding techniques and single-chip decoders for multi-anode microchannel arrays, *Proc. Int. Society of Optical Engineering* **1158** (Aug. 1989) 311–318.
[22] H.-L.S. Workshop, Benchmark suite, *HLSW*, 1989.
[23] C.J. Tseng and D. Siewiorek, Automated synthesis of data paths in digital systems, *IEEE Trans. CAD/ICAS* **CAD-5** (July 1986) 379–395.
[24] P. Paulin, J. Knight and E. Girczyc, HAL: A multi-paradigm approach to Putomatic data-path synthesis, *Proc. Design Automation Conf.* (June 1986) 263–270.
[25] P. Dewilde, E. Deprettere and R. Nouta, Parallel and pipelined VLSI implementation of signal processing algorithms, in: kung and Whitehouse, eds., VLSI and Modern Signal Processing (Prentice-Hall, Englewood Cliffs, NJ, 1985) pp. 258–264.
[26] P. Paulin and J. Knight, Algorithms for high-level synthesis, *IEEE Design and Test Magazine* (Dec. 1989) 18–31.
[27] D. Thomas, E. Dirkes, R. Walker, J. Rajan, J. Nestor and R. Blackburn, The System Architect's Workbench, Proc. Design Automation Conf., June 1990.
[28] B. Pangrle and D. Gajski, Slicer: a state synthesizer for intelligent compilation, *Proc. Int. Conf. Computer Design* (Oct. 1987) 42–45.
[29] R. Potasman, J. Lis, A. Nicolau and D. Gajski, Percolation based synthesis, *Proc. Design* Automation Conf., June 1990.
[30] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, MIS: A multiple-level logic optimization system, *IEEE TRans. CAD/ICAS* **6** (6) (Nov. 1987) 1062–1081.