Relative Scheduling under Timing Constraints

David Ku Giovanni De Micheli

Center for Integrated Systems
Stanford University

Abstract

Scheduling techniques are used in high-level synthesis of integrated circuits. Traditional scheduling techniques assume fixed execution delays for the operations. For the synthesis of ASIC designs that interface with external signals and events, operations with *unbounded delays*, i.e. delays unknown at compile time, must also be considered. We present a *relative scheduling* technique that supports operations with fixed and unbounded delays. The technique satisfies the timing constraints imposed by the user, which places bounds between the activation of operations. We analyze a novel property called *well-posedness* of timing constraints that is used to identify consistency of constraints in the presence of unbounded delay operations, and present an approach to relative scheduling that yields a *minimum schedule* that satisfies the constraints, or detects if no schedule exists, in polynomial time.

1 Introduction

High-level synthesis of digital hardware from behavioral specifications has been shown to be a practical and efficient means of design. Many tasks need to be performed in high-level synthesis to transform an abstract hardware representation into an interconnection of modules and a corresponding control unit. *Scheduling* and *module binding* are among the most important tasks in order to synthesize circuits that are efficient in terms of area and performance. These two problems can be modeled as *scheduling under resource constraints*, which unfortunately is an intractable problem [1]. For this reason, most high-level synthesis system either separate the two tasks or use heuristic approaches. Some systems perform module binding before scheduling, e.g. Caddy/DSL [2] and BUD [3]; some systems perform scheduling before module binding, e.g. Facet [4], DAA [5], YSC [6]. Combined heuristic scheduling and binding are performed in other synthesis systems, such as MAHA [7], Elf [8], Slicer/Splicer [10], Chippe [11], Hal [12], and GENIE-S [9]. It is important to remark that most of these approaches assume that each module is characterized *a priori* in terms of area and execution time.

We consider in this paper the scheduling problem for the high-level synthesis of digital Application-Specific Integrated Circuits (ASICs). This class of circuits has two important characteristics. First, ASICs often interface with, and synchronize on, external signals. Therefore, ASIC modeling in terms of high-level specifications require synchronization primitives and data-dependent iterations. These operations have execution delays that are not known at compile time, or equivalently, their delays are *unbounded*. Second, real-time ASIC applications require the specification of *timing constraints* in the hardware model and their enforcement in the synthesis process [13] [14]. Timing constraints specify upper and lower bounds on the time separation between two operations. They can be applied, for example, to control the time gap between a read and a write of an external bus, or to synchronize two write operations.

We present in this paper a scheduling algorithm under timing constraints that supports operations with *unbounded* delay. We assume that scheduling follows module binding as in Caddy/DSL [2] and BUD [3]. We extend the traditional formulation of scheduling to support unbounded delay operations by introducing the *relative scheduling* problem. We analyze the properties of timing constraints in the presence on unbounded delays by introducing the notion of *well-posedness* of the constraints. We present an algorithm called *iterative incremental scheduling* that finds a minimum schedule which satisfies a set of timing constraints, or detects if no schedule exists, both in polynomial time. We comment then on the implementation of the algorithm in the framework of the HERCULES high-level synthesis system [15].

2 Hardware Model

We model hardware behavior as a set of operations and a partial order among the operations. Each operation is synchronous and therefore it takes an integral number of cycles to execute, called its execution delay. The execution delay may not be known *a priori*, as in the case of external synchronization and data-dependent iteration. In this case, we say that the execution delay is *unbounded*. The partial order represents the *sequencing* dependencies among operations that arise due to *data-flow* restrictions or *module-sharing* limitations. An important assumption made in relative scheduling is that module binding has been performed prior to scheduling. Furthermore, any conflicts due to the assignment of multiple operations to a single module have already been resolved by introducing sequencing dependencies between these operations. This is in contrast to heuristic approaches that combines scheduling with module binding [7, 10, 12].

Several high-level synthesis systems use variations of this general hardware model [2, 6, 7, 18]. In particular, the Hercules high-level synthesis system [15] represents the hardware model by a polar hierarchical acyclic graph, where the vertices represent operations to perform and the edges represent the dependencies among the operations. The hierarchy supports *procedure call*, *conditional branching*, and *iteration*[1] constructs of the hardware description language. We use this model as the basis for scheduling. In Hercules, scheduling is applied hierarchically in a bottom-up fashion.

[1] It is important to note that hardware descriptions with structured iterative constructs may still be modeled by acyclic graphs through the use of hierarchy, i.e. the body of a loop is a separate graph.

For the sake of simplicity, we consider only a non-hierarchical model in this paper. The extension to hierarchical scheduling is straight-forward.

3 Problem Formulation

We model the scheduling problem under timing constraints by means of a polar *constraint graph* $G(V, E)$. The vertices of the constraint graph represent the operations. There are $|V| = n + 1$ vertices in the graph, where v_0 and v_n denote the source and sink vertices, respectively. The edge set represents the dependencies, where a weight w_{ij} is associated with each edge (v_i, v_j) that is equal to the execution delay of the operation v_i, denoted by $delay(v_i)$. Let us assume first that the weights are known; this assumption will be removed in the next section. In the case where no timing constraints are specified, the graph is acyclic, and the scheduling problem may be defined as follows:

Definition 3.1 *A* **schedule** *of a constraint graph $G(V, E)$ is an integer labeling $\sigma : V \rightarrow Z^+$, such that $\sigma(v_j) \geq \sigma(v_i) + w_{ij}$ if there is an edge from v_i to v_j with weight w_{ij}. A* **minimum schedule** *is a schedule such that $(\sigma(v_i) - \sigma(v_0))$ is minimum for all $v_i \in V$.*

The integer label $\sigma(v_i)$ associated with a vertex v_i represents the time (or equivalently the cycle) with respect to the beginning of the schedule ($\sigma(v_0)$) in which the operation modeled by v_i may begin execution, i.e. $\sigma(v_i)$ is the *start time* of v_i. The start time of an operation is used by the control to determine when the operation can begin execution. The acyclic nature of the constraint graph guarantees the existence of a minimum schedule.

We introduce now timing constraints to define upper and lower bounds between the start times of two operations:

- A *minimum* timing constraint $l_{ij} \geq 0$ requires that: $\sigma(v_j) \geq \sigma(v_i) + l_{ij}$
- A *maximum* timing constraint $u_{ij} \geq 0$ requires that: $\sigma(v_j) \leq \sigma(v_i) + u_{ij}$

We incorporate timing constraints into the constraint graph as follows. For every minimum timing constraint l_{ij}, we add a *forward* edge (v_i, v_j) in the constraint graph with weight equal to the minimum value $w_{ij} = l_{ij} \geq 0$. For every maximum timing constraint u_{ij}, we add a *backward* edge (v_j, v_i) in the constraint graph with weight equal to the negative of the maximum value $w_{ij} = -u_{ij} \leq 0$, because $\sigma(v_j) \leq \sigma(v_i) + u_{ij}$ implies $\sigma(v_i) \geq \sigma(v_j) - u_{ij}$. An example of a constraint graph is shown in Figure 1. The double-circled vertices v_0 and a are vertices with unbounded delays $d(v_0)$ and $d(a)$. The execution delays for v_1, v_2, and v_3 are fixed, are are equal to 2, 2, and 5, respectively.

In the resulting constraint graph $G(V, E)$, the edge set $E = E_f \cup E_b$ consists of *forward* E_f and *backward* E_b edges. The forward edges have positive weights and represent minimum timing constraints and operation dependencies; the backward edges have negative weights and represent maximum timing constraint, as shown in Figure 1. The subgraph $G_f = (V, E_f)$ containing only the forward edges is called the *forward* constraint graph. Without loss of generality we assume that $G_f = (V, E_f)$ is acyclic, i.e.

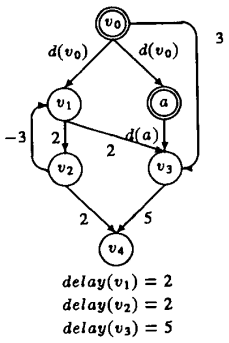


Figure 1: Example of a constraint graph, with a maximum timing constraint from v_1 to v_2 and a minimum timing constraint from v_0 to v_3. v_0 and a are anchors in the graph.

we do not consider a minimum timing constraints l_{ij} to be valid if there is already a path of dependencies from v_j to v_i. In particular, if $l_{ij} > 0$, then the constraint violates the dependencies among the operations; otherwise, if $l_{ij} = 0$, then it can be modeled equivalently by a maximum timing constraint $u_{ji} = 0$ from v_j to v_i. Cycles in the forward constraint graph can be detected by using Dijkstra's algorithm. Note that the values of the execution delays are irrelevant for this check. With this assumption, we say that a vertex v_i is a successor of vertex v_j ($v_i \in succ(v_j)$) if there is a directed path from v_j to v_i in $G_f(V, E_f)$.

This scheduling problem bears similarity to the constrained layout *compaction* problem [19] [20]. Both problems involve finding the spacing relationships for a set of elements to meet a set of upper and lower bound constraints. In the case of compaction, the elements are objects to be placed on a layout, whereas for scheduling, the elements are operations to be ordered in time. A common goal in both problems is to minimize the total spacing among the elements.

3.1 Relative Scheduling

Scheduling problems are defined and solved on graphs with fixed delay operations. We extend this notion to graphs with unbounded delay vertices. For an unbounded delay vertex v_i, the execution delay $delay(v_i)$ is not known statically, and can assume any integer value from 0 to ∞. For this reason, we define a subset of the vertices, called *anchors*, that serve as reference points for specifying the start times of operations.

Definition 3.2 *The* **anchors** *$A \subseteq V$ of a constraint graph $G(V, E)$ are the source vertex v_0 and all vertices with unbounded delay.*

The source vertex v_0 is treated as an anchor since the activation of a sequencing graph is analogous to the completion of the source vertex, which is not known a priori. Therefore, all outgoing edges from v_0 has weight equal to $delay(v_0)$, which is unbounded.

We extend the scheduling problem in the presence of unbounded delay vertices by introducing the concept of *offsets* with respect to

the anchors of the graph. Let $V_a \subseteq V$ be the subset of the vertices including a and all its successors. Let $G_a(V_a, E_a)$ be the subgraph induced by V_a, where the execution delays of all unbounded delay vertices assume the minimum value of zero.

Definition 3.3 *The* **offset** *of a vertex $v_j \in V_a$ with respect to an anchor a is an integer value $\sigma_a(v_j)$ such that $\sigma_a(v_j) \geq \sigma_a(v_i) + w_{ij}$ if there is an edge of weight w_{ij} from v_i to v_j in $G_a(V_a, E_a)$, and $\sigma_a(a)$ is normalized to zero. If $\sigma_a(v_i)$ is the minimum value, then it is the* **minimum offset** *of v_j w.r.t. a, and it is denoted by $\sigma_a^{min}(v_i)$.*

Finding the set of offsets is identical to scheduling $G_a(V_a, E_a)$, where the constraint graph models both operation dependencies and timing constraints. If no such set exists, then the constraints are said to be *inconsistent*. Since the execution delay of an unbounded delay vertex can be any integer greater than or equal to zero, a minimum offset $\sigma_a(v_i)$ is the minimum time after the completion of the anchor a before v_i can begin execution.

We relate now the offsets to the start time of a vertex. Let us consider first the anchors that affect the activation of a vertex v_i.

Definition 3.4 *The* **anchor set** *of a vertex v_i is the subset of anchors $A(v_i) \subseteq A$, such that $a \in A(v_i)$ if there exists a path in $G_f(V, E_f)$ from a to v_i containing at least one unbounded weight edge with weight equal to $delay(a)$.*

In other words, an anchor a is in the anchor set of a vertex if the vertex can begin execution only *after* the completion of a. Note that since the graph is polar, the source vertex is contained in the anchor set of every vertex. The anchor set represents the *unknown* factors that affect the activation time of an operation. If we generalize the definition of the *start time* of a vertex in terms of fixed time offsets from the *completion* time of each anchor in its anchor set, then it is possible to completely characterize the temporal relationships among the operations. In particular, the offsets of a vertex can be related to its start time when the execution delays $\{delay(a) | a \in A\}$ of the anchors are known. The **start time** of a vertex v_i, denoted by $T(v_i)$, is defined recursively as follows:

$$T(v_i) \equiv \max_{a \in A(v_i)} \{T(a) + delay(a) + \sigma_a(v_i)\}$$

Note that if there are no unbounded delay vertices in the graph, then the start times of all operations are specified in terms of time offsets from the source vertex, which reduces to the traditional scheduling formulation. We define the relative scheduling problem as follows.

Definition 3.5 *A* **relative schedule** *Ω of a constraint graph $G(V, E)$ is the set of offsets of each vertex $v_i \in V$ with respect to each anchor in its anchor set $A(v_i)$, i.e. $\Omega = \{\sigma_a(v_i) | a \in A(v_i), \forall v_i \in V\}$. A* **minimum relative schedule** *Ω^{min} is the set of corresponding minimum offsets, i.e. $\Omega^{min} = \{\sigma_a^{min}(v_i) | a \in A(v_i), \forall v_i \in V\}$.*

A minimum relative schedule for a constraint graph $G(V, E)$ guarantees that, for all profiles of execution delays $\{delay(a)\}$, the delay from the source vertex to the sink vertex is minimum.

Vertex v_i	*Anchor Set* $A(v_i)$	*Offsets* σ_{v_0}	σ_a
v_0	$\emptyset$	-	-
a	$\{v_0\}$	0	-
v_1	$\{v_0\}$	0	-
v_2	$\{v_0\}$	2	-
v_3	$\{v_0, a\}$	3	0
v_4	$\{v_0, a\}$	8	5

Figure 2: Illustrating anchor sets and minimum offsets for constraint graph in the previous example.

This can easily be shown from the expression for $T(v_i)$ above by noting that if $\sigma_a(v_i)$ is minimum for all v_i, then $T(v_i)$ is also minimum for all v_i. Consider the constraint graph in Figure 1. The anchor sets and minimum offsets of the vertices are given in Figure 2. For example, vertex v_4 has two anchors v_0 and a with corresponding offsets $\sigma_{v_0} = 8$ and $\sigma_a = 5$; the start time of v_4 is given as:

$$T(v_4) = \max\{T(v_0) + delay(v_0) + 8, T(a) + delay(a) + 5\}$$

3.2 Well-Posedness of Timing Constraints

An important consideration during scheduling is whether a schedule exists under the required timing constraints. An analysis of the consistency of timing constraints was presented in [21]. However, the approach does not consider unbounded delay operations. We extend the analysis by introducing the concept of *well-posed* verses *ill-posed* timing constraints in the presence of unbounded delay vertices.

Intuitively, the unbounded delay vertices create time gaps that cannot be resolved statically. Depending on the execution profile of these operations, a timing constraint *may* or *may not* be satisfied by a given schedule. Consider the examples in Figure 3. Both graphs contain an *ill-posed* maximum timing constraint u_{ij} from v_i to v_j, represented by a backward edge (v_j, v_i) with weight $-u_{ij}$. In Figure 3(a), an unbounded delay vertex exists on the path from v_i to v_j. Depending on how long it takes to complete execution, the constraint may or may not be satisfied. Similarly for Figure 3(b), the activation of v_i depends on the completion of a_1, and the activation of v_j depends on the completion of a_2, both of which are unbounded. Therefore, the determination of whether the constraint is satisfiable depends on unbounded execution delays, and hence is ill-posed. More formally, we define the following.

Definition 3.6 *A timing constraint is* **well-posed** *if its satisfiability does not depend on the execution delay of any unbounded delay vertex.*

Conversely a timing constraint is said to be *ill-posed* if it cannot be satisfied for some values of the unbounded delays. A constraint graph $G(V, E)$ is well-posed if every constraint implied by the edges E is well-posed. Note that minimum timing constraints are always well-posed, because the check for their validity does not depend on the values of the execution delays, as explained in the previous section.

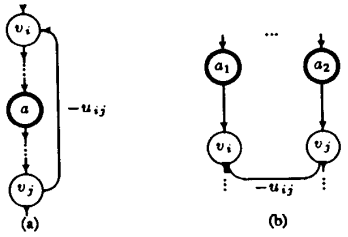


Figure 3: Examples of ill-posed timing constraints, where the doubly-circled vertices represent unbounded delay vertices.

On the other hand, a maximum timing constraint defines an upper-bound between the activation of two operations. If its satisfiability depends on the completion time of an unbounded delay vertex, then the constraint cannot be met in general because it is possible that an input data sequence exists such that the execution delay of the unbounded delay vertex exceeds the upper-bound imposed by the constraint. We state without proof the following theorem as a necessary and sufficient condition for checking if a constraint graph is well-posed. The proof is given in [17].

Theorem 3.1 *Assuming $G(V, E_f)$ is acyclic, a constraint graph $G(V, E)$ is well-posed if and only if $A(v_i) \subseteq A(v_j)$ for all edge $e_{ij} \in E$.*

3.3 Existence of Relative Schedule

For a constraint graph containing only fixed delay vertices, the scheduling problem *may* or *may not* have a solution, depending on the consistency of the imposed timing constraints. A well-known theorem states that a schedule of a constraint graph exists if and only if there are no positive cycles in the graph, where a *positive cycle* is a cycle whose sum of the edge weights is a strictly positive integer [19]. This condition can be checked by the Bellman-Ford algorithm, or more efficiently, by specialized algorithms [19] [20].

We extend the analysis in order to consider graphs with unbounded delay vertices. A positive cycle in the presence of unbounded delay weights is a cycle whose length is strictly positive. Since an unbounded delay weight can be any value from 0 to infinity, a postive cycle can also be unbounded. We state the existence condition of a relative schedule as follows.

Theorem 3.2 *Given a well-posed constraint graph $G(V, E)$, a relative schedule for $G(V, E)$ exists if and only if there are no positive cycles in $G(V, E)$.*

For a well-posed constraint graph, there are no cycles with unbounded weight edges. Therefore, the computation of positive cycle can be made, for well-posed graphs, by setting the unbounded delay weights to zero.

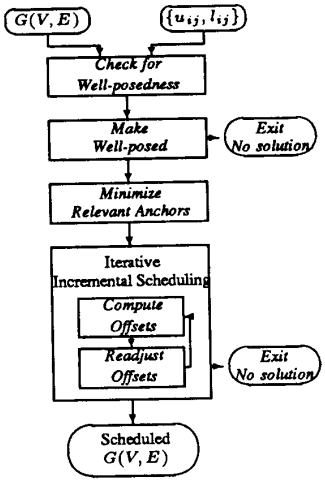


Figure 4: Block diagram of *Relative Scheduling* approach.

4 Relative Scheduling Approach

Given a sequencing graph and a set of minimum and maximum timing constraints, we first generate a constraint graph $G(V, E)$ consisting of forward edges E_f and backward edges E_b. We approach the relative scheduling problem in four steps, as shown in Figure 4.

1. *Checking Well-posed* – The constraint graph is checked for well-posedness using the criterion of Theorem 3.1.
2. *Making Well-posed* – If the constraint graph is ill-posed, then no schedule can satisfy the constraints for all input sequences. We can however attempt to make it well-posed by adding sequencing dependencies to selectively serialize the graph. We describe an algorithm *makeWellposed* that is guaranteed to yield a well-posed graph with *minimum* serialization, if one exists. If the graph cannot be made well-posed, then we regard the set of constraints as inconsistent and exit the algorithm.
3. *Minimize Relevant Anchor Sets* – At this point, the constraint graph is guaranteed to be well-posed. We take advantage of the *cascading effect* of anchors on the start times to minimize the size of the anchor sets. Consider a graph with a single path of forward edges from an anchor a to a vertex v_i containing an anchor b. Since anchor b can begin execution only after a completes, and since v_i can begin execution only after b completes, it is necessary to define the start time $T(v_i)$ with respect to the completion of b **only**. We formalize the observation by introducing the concept of *relevant anchor set* that consists of the set of anchors that may *directly* affect the start time. By using relevant anchor sets in the computation of the relative schedule, we improve the efficiency of the scheduling algorithm and the complexity of the resulting

control because the start times depend on fewer offsets.

4. *Iterative Incremental Scheduling* – Finally, we use an algorithm called *iterative incremental scheduling* that finds the relative schedule by solving the constraint graph. The algorithm is an extension of the technique used by Liao and Wong [19] for layout compaction to support vector solutions, and it is guaranteed to find the minimum relative schedule, or detect the presence of inconsistent timing constraints, in polynomial time.

4.1 Making Well-posed

An ill-posed constraint graph $G(V, E)$ can in some cases be made well-posed by adding sequencing dependencies to G. Consider for example Figure 3(b). The ill-posed constraint can be made well-posed if one adds a sequencing dependency from a_2 to v_i. Although this forces v_i to be serialized with respect to a_2, it is necessary to make the constraint well-posed, i.e. if we are looking for a solution valid under all input sequences. Note that it is not always possible to make an ill-posed constraint well-posed. In particular, if the added sequencing dependency induces a cycle in the forward constraint graph G_f, as in Figure 3(a), then the constraint cannot be transformed into a well-posed constraint.

We describe an algorithm called *makeWellposed* that minimally serializes a constraint graph to make it well-posed. For every backward edge $e_{ij} \in E_b$, the algorithm first checks if there is an anchor a, such that $a \in A(v_j)$ but $a \notin A(v_i)$. If no such a exists, then the constraint is well-posed. Otherwise, it attempts to make the constraint well-posed by adding a forward edge from a to v_i. Procedure *addEdge* adds a forward edge from anchor a to all vertices reachable by a path of backward edges from v.

```
makeWellposed( G(V, E) ) {
    for each ( e_ij ∈ E_b ) do {
        D = {a | a ∈ A(v_i) and a ∉ A(v_j)};
        for each ( a ∈ D )
            addEdge( a, v_j );
    }
}

addEdge( a, v ) {
    if ( a ∉ A(v) ) {
        if ( v is predecessor of a )
            stop with ill-posed constraints;
        Add forward edge (a, v);
        Set weight on (a, v) = delay(a);
        A(v) = A(v) ∪ {a};
        for each ( backward edge (v, b) ∈ E_b )
            addEdge( a, b );
    }
}
```

The worst-case complexity of the *makeWellposed* algorithm is $O(|A| \cdot |E_b|^2)$, where $|A|$ is the number of anchors in G.

4.2 Iterative Incremental Scheduling

The scheduling algorithm is performed by iteratively applying two tasks. The first is *incrementally computing the offsets*. The offsets are initially set to zero, and increased incrementally until all the minimum timing constraints implied by the forward edges are satisfied. This is followed by *readjusting offsets* to meet the maximum timing constraints implied by the backward edges. The scheduling algorithm is described below.

```
IncrementalScheduling( G(V, E) ) {
    for (c = 1 to |E_b| + 1) do {
        IncrementalOffset(G_f, v_0);
        E_violate = {e_ij ∈ E_b | violate constraint};
        if ( E_violate = ∅)
            return minimum relative schedule;
        ReadjustOffsets(G(V, E));
    }
    return no schedule;
}
```

Since the forward constraint graph $G_f(V, E_f)$ is acyclic, the set of offsets satisfying the minimum timing constraints can be found using the longest path calculation from the anchors to their successors. The edge weights in the constraint graph corresponding to the execution delays of unbounded delay vertices are set to 0. The two steps are described below.

1. *IncrementalOffset*. The offsets are computed by successive approximations. Initially, the offsets are set to 0. We then incrementally find the longest path from the anchors to their successors in the forward constraint graph G_f. Specifically, an offset $\sigma_a(v_i)$ is updated as:

$$\sigma_a(v_i) \quad \Leftarrow \quad \max_{v \in V'}\{\sigma_a(v) + LP(v, v_i)\}$$

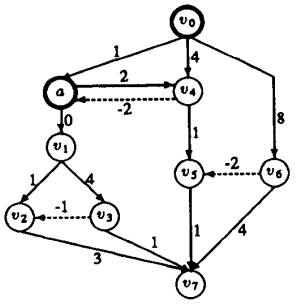
   where $V' \subseteq V$ is the subset of vertices on any path from a to v_i in $G_f(V, E_f)$, and $LP(v, v_i)$ is the length of the longest path from v to v_i in G_f. Note that $LP(v_i, v_i) = 0$.

2. *ReadjustOffset*. After applying *IncrementalOffset*, the resulting offsets satisfy all the minimum constraints implied by the forward edges in G_f. If all the inequalities implied by the backward edges (maximum timing constraints) are satisfied, then the current offsets are the minimum relative schedule, and the algorithm terminates. Otherwise, the algorithm successively accesses each backward edge in E_b to test if the maximum timing constraint implied by the edge is violated. Let $\Omega(v_i) = \{\sigma_a(v_i) | a \in A(v_i)\}$ and $\Omega(v_j) = \{\sigma_a(v_i) | a \in A(v_i)\}$ represent the offsets for two vertices v_i and v_j, respectively. There is a constraint violation on a backward edge (v_i, v_j) with weight $w_{ij} \leq 0$ if, there exists an anchor a common to both anchor sets $a \in A(v_i) \cap A(v_j)$ such that $\sigma_a(v_i) < \sigma_a(v_j) + w_{ij}$. If the constraint is violated, then the offset $\sigma_a(v_j)$ is increased by the minimum amount to satisfy the inequality constraint,

$$\sigma_a(v_j) \Leftarrow \sigma_a(v_i) + w_{ij}$$

   It is important to note that in the case of well-posed timing constraints, $A(v_i) \subseteq A(v_j)$. After the readjustments, *IncrementalOffset* is reapplied, and the process repeats until all maximum timing constraints due to the backward edges are satisfied.

We prove in [17] that the algorithm has polynomial time complexity by finding the minimum relative schedule, or detects inconsistent timing constraints by executing at most $(|E_b|+1)$ iterations. We illustrate the application of the algorithm on the graph of Figure 5. There are two anchors v_0 and a, the dashed-arcs represent backward edges. The offsets for each step of the algorithm are given.



Vertex	Iteration 1		Iteration 2		Final
	Compute σ_{v_0}, σ_a	Readjust σ_{v_0}, σ_a	Compute σ_{v_0}, σ_a	Readjust σ_{v_0}, σ_a	Compute σ_{v_0}, σ_a
v_0	-,-		-,-		-,-
a	1,-	2,-	2,-		2,-
v_1	1,0		2,0		2,0
v_2	2,1	4,3	4,3	5,3	5,3
v_3	5,4		6,4		6,4
v_4	4,2		4,2		4,2
v_5	5,3	6,3	6,3		6,3
v_6	8,-		8,-		8, -
v_7	12,5		12,6		12,6

Figure 5: Trace of offsets in the scheduling algorithm.

5 Implementation

The relative scheduling approach has been integrated in the *Hercules* and *Hebe* high-level synthesis system [15, 16]. Hercules performs behavioral optimizations on a *HardwareC* hardware description, and generates a maximally parallel sequencing graph that is the basis for *Hebe*'s structural optimizations. The objective of *Hebe* is to explore design tradeoffs in meeting the required timing and resource constraints. First, a binding of operations to specific resource components is selected to meet the resource and interconnect constraints. The selected binding may have resource contentions, e.g. two parallel operations bound to the same resource may simultaneously access the shared resource. In this case, all operations bound to the same resource component are serialized by adding sequencing edges via a branch and bound search to determine the best ordering. Finally, relative scheduling is performed on the graph model. The computed offsets are used to construct a control unit for the resulting hardware, which can be implemented, for example, by a set of look-up tables scanned by counters. More elaborate implementations are possible by using logic synthesis techniques.

6 Summary

We have presented a generalization of the scheduling problem that supports unbounded delay operations. The relative scheduling problem under timing constraints is an important task in the synthesis of ASIC designs that interface to external signals and events. We introduced the property of *well-posed* timing constraints that is used to check the consistency of constraints in the presence of unbounded delay operations. We presented a technique called *iterative incremental scheduling* that finds a provably *minimum* relative schedule, or detect the presence of inconsistent timing constraints, both in polynomial time. The techniques are integrated in the framework of the Hercules synthesis system.

7 Acknowledgments

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 Philips/Signetics.

References

[1] M. Garey, D. Johnson, *Computers and Intractability*, W. Freeman and Co, 1979.

[2] R. Camposano, W. Rosenstiel, *Synthesizing Circuits from Behavioral Descriptions*, IEEE Transactions on CAD, Vol 8, No. 2, Feb 1989, p. 171-180.

[3] M. J. McFarland, *Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions*, Proceedings 23 th Design Automation Conference, June 1986, p. 474-480.

[4] C. Tseng, D. Siewiorek, *Automated Synthesis of Data Paths in Digital Systems*, IEEE Transaction on CAD, Vol CAD-5, pp. 379-395, July 1986.

[5] T. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Boston, 1985

[6] R. K. Brayton, R. Camposano, G. De Micheli, R. Otten, J. van Eijndhoven, *The Yorktown Silicon Compiler System*, in *Silicon Compilers*, D. Gajski (ed.), Addison Wesley 1987, pp. 204-310.

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

[8] E. Girczyc, J. Knight, *An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling*, Proceedings of ICCD, Oct 1984, p. 726-731.

[9] S. Devadas, R. Newton, *Algorithms for Hardware Allocation in Data-Path Synthesis*, Proceedings of ICCD, Oct, 1987, pp. 526-531.

[10] B. Pangrle, D. Gajski, *Slicer: a State Synthesizer for Intelligent Compilation*, Proceedings of ICCD, pp. 42-45, 1987.

[11] F. Brewer, D. Gajski, *Knowledge Based Control in MicroArchitectural Design*, Proceedings of 24 th Design Automation Conference, pp. 203-209.

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

[13] J. Nestor, D. Thomas, *Behavioral Synthesis with Interfaces*, Proceedings ICCAD 86, pp. 112-115.

[14] G. Borriello, R. Katz, *Synthesis and Optimizations of Interface Transducer Logic*, Proceedings of ICCAD 87, pp. 56-60.

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

[16] D. Ku, G. De Micheli, *High-level Synthesis and Optimization Strategies in Hercules and Hebe*, Proceedings of EuroASIC, Paris, France, May 1990.

[17] D. Ku, G. De Micheli, *Relative Scheduling under Timing Constraints*, Stanford CSL Technical Report CSL-TR-402, 1989.

[18] M. J. McFarland, *Value Trace*, CMU Internal report, 1978

[19] Y. Liao, C. Wong, *An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints*, IEEE Transactions on CAD, Vol. CAD-2, No. 2, Apr 1983, pp. 62-69.

[20] A. R. Newton, *Symbolic Layout and Procedural Design*, in *Design Systems for VLSI*, G. De Micheli et. al. (ed.) pp. 65-112.

[21] R. Camposano, A. Kunzmann, *Considering Timing Constraints in Synthesis from a Behavioral Description*, Proceedings of ICCD, pp. 6-9, 1986.