Control Optimization Based on Resynchronization of Operations

David C. Ku Dave Filo Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

Most approaches to control optimization use a finite state machine model, where operations are bound to control states. However, when synthesizing circuits from a higher, more abstract level of hardware specification that supports concurrency and synchronization, these approaches may be overly restrictive. We present a strategy for optimizing control circuits based on *resynchronization of operations* such that the original specification under *timing constraints* is still satisfied, but with a lower control implementation cost. We use a general *constraint graph model* to capture the high level specification; the model supports unbounded delay operations, detailed timing constraints, and concurrency. We introduce the notion of synchronization *redundancy*, and formulate the optimization problem as the task of mapping operations to synchronization points. We present algorithms to find a minimal control cost implementation. Results of applying the technique within the framework of the *Hercules/Hebe* High-level Synthesis system are presented.

1 Introduction

We consider the synthesis of synchronous digital systems from behavioral descriptions that include the specification of timing constraints [1]. We address the problem of finding a *minimal-area control implementation*, such that the overall hardware is a valid implementation of its behavioral model.

Control optimization can be performed either at the logic level, by using a *finite-state machine* model [2], or at a higher level, by using a hardware model described in terms of constraints on the sequencing and timing of the operations [3]. In the former case, the operations are bound to control states. This implies that the cycle-per-cycle behavior of the control cannot be altered without changing the control specification. Techniques such as sequential logic synthesis and microcode compaction can be used to reduce the cost of the control implementation [4]. In contrast, in the latter approach, hardware behavior is modeled as a set of sequencing and timing constraints on the operations; the activation of an operation is synchronized to the completion of a set of operations. Since operations are not bound to control states, it is possible to modify the activation time of an operation provided the sequencing dependencies and timing constraints of the original specification are still satisfied. The wider latitude in choosing among a set of possible implementations can lead to a more efficient control implementation in terms of area, which can be further improved by logic synthesis techniques.

In this paper, we present algorithms for control optimization based on *resynchronization of operations* that supports concurrency, detailed timing constraints and external synchronizations (unbounded delay operations). After an overview of the hardware and control model, we present the concept of *synchronization redundancy* in Section 3. We show how redundancies can be introduced by delaying operations through lengthening and serializing a graph-based hardware model. Section 4 describes the optimization algorithms. We conclude with results of applying the technique in the framework of the *Hercules/Hebe* High-level Synthesis system.

Hardware Model. Most existing high-level synthesis systems model hardware behavior as control/data flow graphs [5]. We model hardware timing behavior as a partial order among a set of operations, and we represent it as a polar directed edge-weighted *constraint graph* $G(V, E)$. The vertices V represent the operations, and the edges E capture the precedence and timing relationships among the operations. Each operation $v \in V$ is synchronous and therefore takes an integral number of cycles to execute, called its execution delay, which is denoted by $\delta(v)$. The execution delay may not be known a priori, as in the case of external synchronization and data-dependent loops. In this case, we say the execution delay is *unbounded*. The model supports concurrency, hierarchy, and *detailed timing constraints* that specify bounds on the minimum/maximum time separation between the activation of operations. We refer the interested reader to [6] for details of the constraint graph model. We now describe properties of the model that are important in the control formulation.

A weight w_{ij} associated with each edge $e_{ij} = (v_i, v_j) \in E$ represents the requirement that the start time of v_j (denoted by $T(v_j)$) must occur later than w_{ij} after the start time of v_i, i.e. $T(v_j) \geq T(v_i) + w_{ij}$. For example, a sequencing dependency from v_i to v_j is represented by a forward edge from v_i to v_j with weight $\delta(v_i)$. The edges are categorized into *forward* (E_f) and *backward edges* (E_b). The forward (backward) edges have positive (negative) weights and represent minimum (maximum) timing requirements among the operations. Both forward and backward edges may have unbounded weights, i.e. an unbounded forward edge represents a minimum sequencing constraint whereas an unbounded backward edge represents a maximum sequencing constraint between two operations. Without loss of generality, we assume the graph induced by the forward edges is acyclic, and that all cycles in the graph have bounded length. The graph model serves to determine the extent to which the activation of operations can be modified in optimizing the control implementation. We assume that the mapping of operations to resources has been

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-395-7/91/0006/0366 $1.50

performed, and that resource conflicts have been resolved prior to control synthesis by serializing between the conflicting operations.

2 Control Model

To define the cost function that will drive the control optimization, we describe first the mapping from a constraint graph to a control implementation. The mapping involves two tasks – scheduling and control generation. Given a constraint graph, *scheduling* finds the start times of the operations satisfying the timing constraints, which are then used by *control generation* to derive a FSM specification of the control.

2.1 Scheduling the Operations

Traditionally, the scheduling problem assigns operations to control states, and it can be formulated as an integer labeling problem that assigns to each operation an integer value representing the time from the beginning of the schedule at which it begins execution. The presence of *unbounded delay operations* in our hardware model invalidates this traditional scheduling formulation since an absolute schedule satisfying timing constraints no longer exists. We use a formulation called *relative scheduling* that schedules an operation with respect to the completion of a set of unbounded delay operations. We now review some terminology and results that are pertinent to our presentation, and we recommend to those unfamiliar with relative scheduling to first read [6].

We define a subset of the vertices, called **anchors**, that serve as reference points in specifying the activation of operations. The anchors (denoted by $A \subseteq V$) of a constraint graph consist of the source vertex v_0 and all vertices with unbounded delay. The **anchor set** of a vertex v (denoted by $A(v)$) is the subset of anchors that are in the transitive fan-in of v. In other words, the anchor set consists of all unbounded delay vertices which affect the activation of a given operation. The **start time** of a vertex v (denoted by $T(v)$) is defined as offsets $\sigma_a(v)$ from the completion of each anchor in the anchor set $a \in A(v)$, i.e. $T(v) = \max_{a \in A(v)}\{T(a) + \delta(a) + \sigma_a(v)\}$.

A constraint graph is **feasible** if its constraints can be satisfied when the unbounded delays are equal to zero. If there are no unbounded delay operations, then the concept of feasibility is sufficient to guarantee that a schedule exists. It can be shown that a graph is feasible if it contains no positive length cycles. In the presence of unbounded delays, however, we extend the analysis by defining a graph to be **well-posed** if its constraints can be satisfied *for all values* of unbounded delays. The concept of well-posedness is important to ensure the resulting synthesized hardware is valid for all possible input conditions. Note that well-posedness implies feasibility, and non-feasibility implies ill-posedness; therefore, we assume that we are given a well-posed constraint graph as input to the control optimization.

2.2 Generating the Control Circuit

Given a schedule, the task is to generate the corresponding control logic. We model the control in terms of a modular interconnection of synchronous FSMs; the FSM abstraction decouples the control generation from a particular style of logic-level implementation. The task of control generation is abstracted as generating enable and done signals for each vertex v, such that its activation (completion) is indicated by the assertion of the signal $enable_v$ ($done_v$). The completion of the unbounded delay operation corresponding to each anchor $a \in A$ is indicated by the assertion of a signal $done_a$, which stays asserted until all operations in the graph have completed. Details of generating $done_a$, along with the support for conditional branching and looping, are described in [7]. We now describe control generation for the remaining signals.

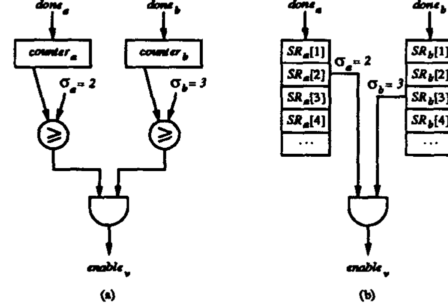

Figure 1: Alternate implementation styles for control generation: (a) counter-based, and (b) shift-register based.

We illustrate two possible alternatives for control generation in Figure 1, where we show the circuit that enables an operation v. For the sake of illustration, we assume the anchor set of v is comprised of a and b, with offsets $\sigma_a(v) = 2$ and $\sigma_b(v) = 3$, respectively. That is, operation v must start at least 2(3) cycles after the completion of operation $a(b)$. In the first case, we implement the offset control as a counter and the synchronization control as a set of comparisons between the counter values and appropriate offsets. For the second case, we implement the offset control as a shift register and the synchronization control as logic conjunctions of the appropriate shift register outputs. We see that the control circuit consists of two types of components: an *offset control circuit* for each anchor and a *synchronization control circuit* for each vertex. The offset control circuit generates signals that indicate the time offset from the completion of an anchor, and the synchronization control circuit coordinates the activation of an operation with respect to offsets in its start time.

Given a specification of control logic in terms of FSMs, we estimate the total **control cost** $COST_{area}$ of the control implementation as:

$$COST_{area} = \sum_{\forall a \in A} COST_{off}(a) + \sum_{\forall v \in V} COST_{sync}(v)$$
$$= \alpha \cdot \sum_{\forall a \in A} f_{off}(\sigma_a^{max}) + \beta \cdot \sum_{\forall v \in V} f_{sync}(|A(v)|)$$

The first term $COST_{off}(a)$ is related to the cost due to the length of the schedule corresponding to an anchor $a \in A$; it is a function f_{off} of the maximum offset value for a (σ_a^{max}) that yields the number of registers implementing the offset FSM for anchor a. The second term $COST_{sync}(v)$ is related to the cost of the synchronization logic for $v \in V$; it is a function f_{sync} of the cardinality of the anchor set of v ($|A(v)|$). The values α and β represent appropriate weight factors related to the actual cost of the logic implementation.

Alternative strategies to implement the control logic exist, i.e. it is possible to specify the control as a finite state machine. In all formulations, the control complexity can be reduced by either minimizing the *maximum offsets* and/or by reducing the size of the *anchor sets*. This can be achieved by *modifying* the graph topology and/or the edge weights as shown in the following sections.

3 Redundancy in Synchronization

It is often the case that some anchors in the anchor set of a vertex are not needed in the computation of its start time, i.e. $T(v)$ is unchanged if offsets from these anchors are not used in its computation. Intuitively, redundancy arises due to the cascading effect of the synchronization dependencies. Consider the example in Figure 2(b) with two anchors a and b (represented by double circles),

a vertex v, and arcs that represent minimum timing requirements between the operations. Vertex v can execute only when 4 cycles have elapsed after the completion of a and 2 cycles after the completion of b. In contrast, v in Figure 2(c) no longer depends directly on the completion of a; instead this dependency exists implicitly through the dependency on anchor b. Therefore, a is redundant with respect to v and can be ignored in computing the start time $T(v)$.

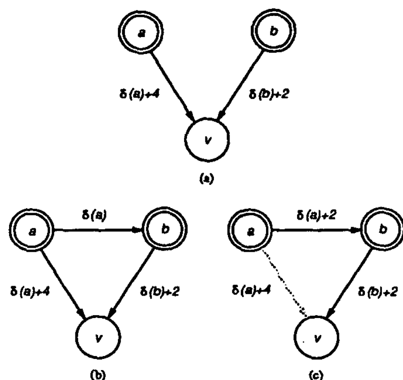

Figure 2: Example of redundancy: (a) original graph (b) serialization edge e_{ab} added (c) edge e_{ab} lengthened to make a redundant w.r.t. v

More formally, let $lp(v, w)$ denote the length (sum of edge weights) of the longest path in the constraint graph from v to w such that all unbounded execution delays are set equal to zero. We define an anchor a to be **redundant** with respect to vertex v if there exists an anchor q that lies on a path from a to v, i.e. $q \in A(v)$ and $a \in A(q)$, such that $lp(a, v) \leq lp(a, q) + lp(q, v)$; otherwise it is **irredundant**. The irredundant anchor set of a vertex v, denoted by $IR(v)$, is the minimum set of synchronizing points affecting the activation of v. It can be shown that the start time computed with only irredundant anchors is identical to the start time computed with the full anchor set, for well-posed constraints and minimum offsets.

By using only irredundant anchors in computing the start time, the control cost can be reduced significantly by (1) reducing the size of the anchor sets, translating to lower synchronization costs, and by (2) reducing the maximum offset values, translating to fewer number of states in the corresponding FSM.

3.1 Making Anchors Redundant

Consider an irredundant anchor a with respect to a vertex v. It is sometimes possible to make it redundant either by *lengthening* an existing path, or by *serializing* to introduce new paths in the constraint graph, or by a combination of both. Figure 2 illustrates how the anchor a can be made redundant with respect to vertex v by the two techniques.

The delaying of operations must be carried out with care to avoid violating the constraint graph, i.e. make the resulting graph ill-posed. Both lengthening and serializing can be modeled as adding a forward edge $e_{av} = (a, v)$ representing a sequencing constraint from anchor a to vertex v, with the edge weight $w_{av} = \delta(a) + k$, $k \geq 0$. If a path already exists from a to v, then the new edge implies lengthening; otherwise, it implies serializing. We state the following theorem that provides the basis for determining the validity of a given lengthening or serialization.

Theorem 3.1 *Consider a well-posed constraint graph $G(V, E)$. If a forward edge e_{av} from anchor a to vertex v with weight $w_{av} = \delta(a) + k$, $k \geq 0$ is added from anchor a to vertex v, then the resulting graph $\tilde{G}$ can be made well-posed if and only if:*
(1) $\tilde{G}$ is feasible, and
(2) $\tilde{G}$ does not contain unbounded length cycles. [1]

The two conditions described in Theorem 3.1 are used to ensure that the control optimization constructs valid solutions.

3.2 Prime versus Non-prime Anchors

If an operation is delayed by *lengthening* existing paths in the constraint graph without introducing new serializations, then the full anchor sets of the vertices remain unchanged. The reason is because the transitive fan-in relation for the vertices is not affected by lengthening paths. It is useful to identify the subset of anchors in the anchor set which can be made redundant by lengthening alone. We state the following definition and theorem.

Definition 3.1 *An anchor $p \in A(v_i)$ of a vertex v_i is* **prime** *if for all paths of forward edges from p to v_i, no unbounded delays other than $\delta(p)$ are encountered. Otherwise, the anchor is* **non-prime**. *The set of prime (non-prime) anchors of a vertex v is the* **prime (non-prime) anchor set** *of v, denoted by $PA(v)$ ($NPA(v)$).*

Theorem 3.2 *A prime anchor $p \in PA(v_i)$ of a vertex v_i is always irredundant with respect to v_i.*

Consider for example Figure 2(a), since there exists no other anchors on any path from anchors a and b to vertex v, both are prime anchors of v. On the other hand, anchor a in Figure 2(b) is *not* a prime anchor of v because it is possible to lengthen the graph to make a redundant with respect to v (as in Figure 2(c)).

We observe that an irredundant non-prime anchor can be made redundant by lengthening a path from the non-prime anchor to the vertex, provided that no timing constraints are violated. A constraint graph is called **taut** if all non-prime anchors of a vertex are made redundant with respect the vertex, for all vertices. We state the following theorem.

Theorem 3.3 *Given a well-posed constraint graph $G(V, E)$, there always exists a lengthening of G, denoted by $\tilde{G}$, such that $\tilde{G}$ is taut and well-posed.*

It is important to point out that the prime anchor sets are *fixed* for a given graph topology, and they remain unchanged even if the graph is lengthened. However, it is possible to change the prime anchor sets by serializing the graph. Consider the example in Figure 2. In (a), the prime anchor set of v consists of $\{a, b\}$. By serializing between a and b, anchor a has been made non-prime, as shown in (b). Since a is non-prime, it can be made redundant with respect to v by lengthening, as shown in (c).

4 Control Optimization Approach

From the previous section, we see that synchronization redundancies can be used to reduce the control cost. Since redundant anchors do not affect the start time of an operation, the irredundant anchors of an operation represent its *synchronization points*, i.e. they synchronize the execution of the operation with respect to multiple concurrent execution flows.

We now formulate the task of control optimization as *minimizing the control cost $COST_{area}$ by modifying the constraint graph*, where the modification is modeled as either graph lengthening, graph serialization, or a combination of both. We consider any modification to the constraint graph to be acceptable as long

[1] The proofs can be found in [8] and are not presented here for brevity.

as the resulting graph satisfies all the constraints in the original specification, and remains well-posed.

The two factors in the control cost - synchronization and offset costs, are tightly coupled. The reduction of one may result in an increase of the other. A globally minimum solution does not necessarily imply minimum values in both synchronization and offset control costs. Since simultaneous minimization of both factors may be computationally hard to solve exactly, we use the following three step heuristic strategy. The strategy is based on the observation that non-prime anchors can always be made redundant, and a prime anchor can be made non-prime by serializing it with respect to another prime anchor.

1. *Minimize the prime anchor sets* – by serializing among the anchor.
2. *Resynchronize operations with respect to new synchronization points* – by serializing among the operations to minimize the offset control cost.
3. *Remove redundancy* – by lengthening the graph to make it taut, in order to minimize the synchronization control cost.

Although alternative optimization strategies exist, this approach has the advantage of being able to yield a globally minimum solution for a subclass of constraint graphs.

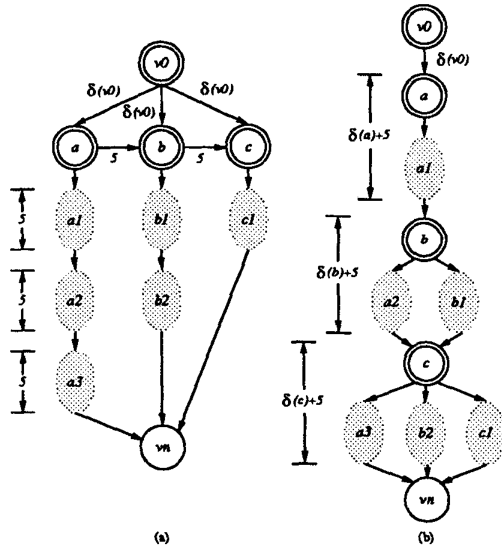

Figure 3: Illustrating the serializing of anchors, and assignment of operations (represented by shaded blocks) to segments of the ordering. Each shaded block requires 5 clock cycles to execute. The offset cost is reduced from $15+10+5 = 30$ to $5+5+5 = 15$.

Before describing the details of each step in subsequent sections, we illustrate our strategy with the example in Figure 3 containing four anchors. Each shaded block represents a group of operations requiring 5 cycles to execute; there is a minimum timing constraint of 5 from a to b, and another minimum constraint from b to c. Originally, the offset control cost is proportional to the sum of the maximum offsets $\sigma_a^{max}+\sigma_b^{max}+\sigma_c^{max} = 15+10+5 = 30$. If we serialize the anchors to form a chain, and then partition the vertices to each segment of the chain as in Figure 3(b), the sum of the maximum offsets is reduced, from 30 to $5+5+5 = 15$. Furthermore, the synchronization cost for v_n has been reduced because its activation no longer depends on the completion of anchors a and b.

4.1 Minimize Prime Anchor Sets

The motivation for serializing anchors is based on the observation that non-prime anchors can always be made redundant by graph lengthening. Therefore, serialization is performed to reduce the prime anchor sets of the vertices as much as possible, which reduces the synchronization cost of the final control implementation. This is important because the synchronization cost for control-dominated machines in certain implementation styles may dominate the overall control cost.

The lower bound on the synchronization control cost corresponds to the case where every vertex excluding the source has a single synchronization point, i.e. $\sum_{v\in V}|IR(v)| = |V|-1$, where $|IR(v)|$ is the number of irredundant anchors of v. This implies that the anchors are *completely* serialized with respect to one another. However, in the presence of maximum timing constraints, the constraint graph may be cyclic; it is therefore not always possible to arbitrarily serialize two anchors without violating timing constraints. To address this issue, we define an *anchor cluster* as follows.

Definition 4.1 *An* **anchor cluster** *denoted by λ_i is a maximal subset of strongly connected anchors in the constraint graph.*

The set of anchor clusters is denoted by Λ, where λ_0 is the cluster containing the source vertex. A constraint graph is called **elementary** if all anchor clusters contain a single anchor, i.e. $|\lambda_i| = 1, \forall i$.

Since strong connectivity is an *equivalence relation*, the anchor clusters form a partition over the set of anchors A. Furthermore, the set of anchor clusters form a *partial order*, and it is possible to find a serialization of the anchor clusters such that the clusters are completely ordered. In the case where all clusters contain a single anchor, an ordering results in a chain of anchors from source to sink. More formally, we define a *cluster ordering* as follows.

Definition 4.2 *A* **cluster ordering** *of a constraint graph G is a complete serialization of the anchor clusters of G, such that for every pair of clusters λ_i and λ_j, every anchor $a \in \lambda_i$ is serialized with respect to every anchor $b \in \lambda_j$. The graph G with a cluster ordering is called an* **ordered** *graph.*

Theorem 4.1 *For a well-posed, ordered, elementary constraint graph, the sum of the cardinality of the prime anchor sets is equal to $|V|-1$.*

The reason that the sum is equal to $|V|-1$ is because $|PA(v)| = 1, \forall v \in V$ except $PA(v_0) = 0$. Cluster ordering reduces the synchronization requirement of a vertex. Since we can make a graph taut, the theorem above states that it is possible to achieve the lower bound in synchronization costs for elementary graphs. We note that imposing a cluster ordering in a graph will not affect the property of well-posedness. The reason is because by definition the clusters are not connected by a cycle in the constraint graph; therefore, no cycles are formed by serializing among the clusters, and hence the resulting graph remains well-posed.

The search for a cluster ordering that is compatible with the original partial order can be carried out using branch-and-bound techniques. Alternatively, heuristic algorithms can be applied to limit the search for a good solution. We use a heuristic that finds an ordering based on ranking the clusters with respect to increasing lengths of the longest path from the source vertex, and serializing the clusters according to the ranking.

4.2 Partition to Resynchronize Operations

A cluster ordering can be viewed as a *chain* of clusters; each link in the chain represents a set of synchronization points along with a set of operations that depend on these points. Given that the prime anchor sets for the vertices have been minimized by imposing the cluster ordering, the idea is to *partition* the operations to

the links of this chain so as to minimize the offset control cost for a given cluster ordering. The process of assigning operations to links involves serializing among operations, and is called **resynchronization** since the activation of operations now depend on a possible new set of anchors (i.e. synchronization points). We formalize this idea with the following definition.

Definition 4.3 *Given a well-posed, ordered graph with two consecutive anchor clusters λ_i and λ_{i+1}. The subgraph that is induced by the vertices on all paths from the anchors in λ_i to but excluding the anchors in λ_{i+1} is called the* **cluster link** $\mathcal{L}_i$ *corresponding to λ_i. A* **segmented** *graph is a graph for which all vertices belong to cluster links.*

A vertex v belonging to a cluster link $\mathcal{L}_i$ satisfies the condition that it is a successor of at least one anchor in λ_i, and therefore its prime anchor set is equal to a subset of the anchors in λ_i, i.e. $PA(v) \subseteq \lambda_i$. A segmented graph is a special form of ordered graph where all operations belong to links.

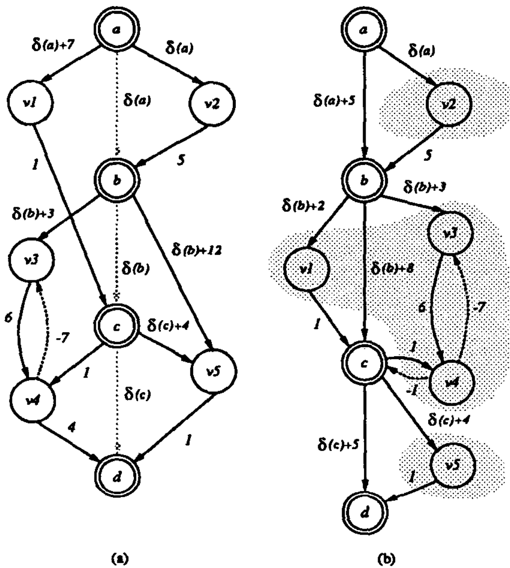

Figure 4: Example of (a) an un-segmented constraint graph, where v_2 is the only c-opset belonging to a link, (b) a segmented constraint graph. The shaded arcs represent the cluster ordering that is derived based on ranking of longest path lengths from the source.

In general, not all vertices of an ordered graph belong to cluster links. For example, a vertex may not fall between consecutive clusters, but rather lie on a path between non-consecutive clusters. Consider the ordered, elementary graph in Figure 4. In (a), only v_2 belongs to a link, whereas in (b), all vertices belong to links. The objective is to partition the vertices by assigning them to the links in such a way as to minimize the overall offset control.cost. The assignment is made by serializing a vertex with respect to the cluster corresponding to the assigned link.

Due to the presence of maximum timing constraints, the vertices cannot be assigned arbitrarily. We define a **constrained operation-set** (c-opset) as a maximal subset of strongly connected vertices, so called because the execution of these vertices are constrained with respect to each other. We state without proof that all elements of a c-opset must be assigned to the *same* segment in the chain if the resulting graph is to be well-posed.

We present a heuristic algorithm, called *ResyncGraph*, that partitions vertices (c-opsets) to links of the chain. The algorithm is based on a greedy ranking strategy, and is described below. Given a cluster ordering, a vertex is assigned to the link corresponding to the nearest anchor cluster that precedes the vertex in the ranking. The partitioning is subject to the condition that the constraint graph remains well-posed. We know from the previous section that the elements of a c-opset must reside in the same link; therefore, if a c-opset is not completely encapsulated between two clusters in the ranking, then some anchors must be *delayed* by the least amount to ensure full containment of the c-opset within a link. The complexity of the algorithm is limited by the search for the longest path, which is quadratic in the number of vertices/edges [6].

ResyncGraph(G)
  /* *Rank based on longest path lengths* */
  Sort $v \in V$ based on $lp(v_0, v)$
  /* *Resynchronize vertices* */
  $Last$ = cluster containing source vertex v_0
  foreach (unassigned c-opset ψ in increasing ranking order)
    Assign ψ to cluster $Last$ by serializing $\psi \rightarrow Last$
    Update $Last$ if ψ is an anchor cluster
  return modified G

4.3 Making the Graph Taut

The final step in the optimization is to remove redundancies by making the constraint graph *taut*, i.e. the non-prime anchors of a vertex are made redundant by lengthening the graph appropriately. Since the prime anchor set is necessarily irredundant, making the non-prime anchors redundant reduces the synchronization control cost. However, it is possible to increase the maximum offset values as a result of lengthening. Therefore, an important criterion in lengthening is to minimize the *amount* of increase. The algorithm for lengthening, called *LengthenTaut*, is given below.

LengthenTaut(G)
  foreach (vertex $v \in V$ in topological order)
    foreach (irredundant non-prime anchor $a \in IR(v) - PA(v)$)
      find forward path $\rho = path(a, q_1, \cdots, q_k, v) : q_k \in PA(v)$
      lengthen ρ by amount $|lp(a,v)| - |\rho|$

The algorithm for lengthening visits each vertex of the graph and lengthens paths from non-prime irredundant anchors as necessary. From a non-prime anchor a to vertex v, it is possible to find a forward path from a to v containing *all* anchors in the subgraph induced by the forward edges between a and v. This forward path is denoted by $\rho = path(a, q_1, \cdots, q_k, b, v)$. The path is lengthened by visiting the segments of the path starting from (a, q_1), (q_1, q_2), and so on, where a path segment is increased as much as possible until either a maximum constraint limit is reached, or until a is made redundant with respect to v. The complexity of the procedure is $O(|V| \cdot |A|^2)$.

The algorithm is guaranteed to make a constraint graph taut. In addition, the algorithm also guarantees for *elementary* constraint graphs that the control offset costs are reduced, or in the worst case remain the same. We state the following important theorem.

Theorem 4.2 *Given a well-posed, elementary constraint graph G, procedure LengthenTaut can make G taut without increasing the maximum offset values of the anchors of G.*

5 Analysis and Example

In general, the interaction between anchor clusters and c-opsets complicates the analysis of the problem formulation and its solution space. For example, it is not guaranteed that a globally minimum cost graph is always ordered or segmented. However, for the case of an *elementary* and *ordered* constraint graph, we

can show that it is always possible to find a minimum control cost $COST_{area}$ solution using our formulation. Specifically, it can be shown that of all the possible minimum solutions, at least one of them is segmented. Thus, by searching all possible ways to segment the graph, it is guaranteed that a minimum solution will be found.

We illustrate the application of our strategy in Figure 4. The graph contains four anchors $\{a, b, c, d\}$, and five vertices $\{v_1, v_2, v_3, v_4, v_5\}$, where v_3 and v_4 form a c-opset. The graph is elementary since each anchor cluster contains a single element. Based on the ranking of longest path lengths, the algorithm impose a cluster ordering $[a, b, c, d]$ corresponding to the lengths 0, 5, 8, 18. An assignment of c-opsets to links is then performed, resulting in assigning v_1 to $\mathcal{L}_b$, v_2 to $\mathcal{L}_a$, v_3 and v_4 to $\mathcal{L}_b$, and v_5 to $\mathcal{L}_c$. The graph is then lengthened to make it taut, i.e. all non-prime anchors are made redundant. This results in lengthening the distance of a-to-b to 5, b-to-c to 8, and c-to-d to 5. The offset cost is reduced from $\sigma_a^{max} + \sigma_b^{max} + \sigma_c^{max} = 8 + 13 + 5 = 26$ to $5 + 9 + 5 = 19$.

6 Implementation and Results

The control optimization algorithms have been implemented in the framework of the Hercules/Hebe High-level synthesis system. The constraint graph model is derived from a high-level specification of hardware behavior, which is then used as the basis for scheduling and control synthesis. We present in Figure 5 the results of applying the heuristic technique on several benchmark examples, including diffeq, elliptic digital filter, error-correcting encoder and decoder, and the greatest common divisor. The benchmark examples have been extended to support reset and synchronizing sequences, indicated by the RM suffix. For each example, the table gives the number of anchors $|A|$, the number of vertices $|V|$, the sum of the maximum offsets and the sum of the anchor sets for (1) the full anchor set, (2) the irredundant anchor set, and (3) after control optimization has been performed. Recall that anchors include both the source vertex and the set of unbounded delay operations, i.e. data-dependent while loops. In particular, multiplication, modeled as repeated conditional addition and shift, and division operations are anchors. This explains the relatively large number of anchors in some of the examples. The control implementation style is a shift-register based scheme. The mapped control logic cost in terms of Actel cells is also given. Note that for control-dominated designs, such as the ECC encoder and decoder, the reduction in control is significant in terms of the overall reduction in area.

7 Summary

We have presented a control optimization strategy based on resynchronization of operations. Using a constraint graph model that supports concurrency, external synchronization (unbounded delay operations) and detailed timing constraints, we showed how the graph can be mapped to a control implementation that consists of synchronization and offset control components. The total control cost can be reduced by introducing *synchronization redundancy* in the graph, where any modification to the graph is considered to be acceptable as long provided the original timing constraints are not violated. We formulated control optimization as partitioning operations to synchronization points, and we described heuristic algorithms based on greedy ranking to improve the computational efficiency at the possible expense of quality. Results of applying the control optimization strategy to examples are presented. Future work includes investigating properties of multi-anchor clusters, and optimization considering both synchronization cost and offset cost simultaneously.

Example graph	$\|A\|/\|V\|$	*Offset* $\sum \sigma_a^{max}$			*Sync* $\sum \|A(v)\|$		
		Full	Irr	Opt	Full	Irr	Opt
Diffeq	14/41	27	12	8	106	52	36
Elliptic	12/66	370	141	95	247	73	63
ECC encoder	6/47	62	62	42	80	63	43
ECC decoder	6/53	59	59	32	92	76	49
Gcd	16/38	15	7	7	45	28	28

Example graph	*Mapped logic*	
	Before	*After*
Diffeq	248/2848	135/2735
Elliptic	647/2147	525/2025
ECC encoder	246/256	203/213
ECC decoder	316/366	251/301
Gcd	320/434	320/434

Figure 5: Summary of results. The control costs are given for the *full*, *irredundant*, and *optimized* anchor sets. The logic cost is the # of Actel cells for the *control* portion and the *total* area.

8 Acknowledgments

The authors would like to thank Bill Lin for many helpful discussion and comments. 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] R. Camposano and A. Kunzmann, "Considering timing constraints in synthesis from behavioral description," in *Proceedings of the International Conference on Computer Design*, pp. 6–9, Nov. 1986.

[2] S. Malik, E. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," in *Proceedings of the Hawaii International Conference on System Sciences*, (Hawaii), pp. 397–406, 1990.

[3] W. Wolf, "Rescheduling for cycle time by reverse engineering," in *Proceedings of the ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, (Univ. of British Columbia), Aug. 1990.

[4] G. Goosens, J. Rabaey, J. Vanderwalle, and H. DeMan, "An efficient microcode compiler for custom DSP-processors," in *Proceedings of the International Conference on Computer-Aided Design*, (Santa Clara), pp. 24–27, Nov. 1987.

[5] R. Camposano and W. Wolf (Ed.), *High-Level VLSI Synthesis*. Kluwer Academic Publishers, June 1991.

[6] D. C. Ku and G. D. Micheli, "Relative scheduling under timing constraints," in *Proceedings of the Design Automation Conference*, pp. 59–64, June 1990.

[7] D. C. Ku and G. D. Micheli, "Optimal synthesis of control logic from behavioral specifications," *Journal of VLSI Integration*, Mar. 1991.

[8] D. Ku, D. Filo, and G. D. Micheli, "Control optimization based on resynchronization of operations," CSL Technical Report CSL-TR-91, Stanford, 1991.