Self-leaning Activity 2
(IEEE paper report)

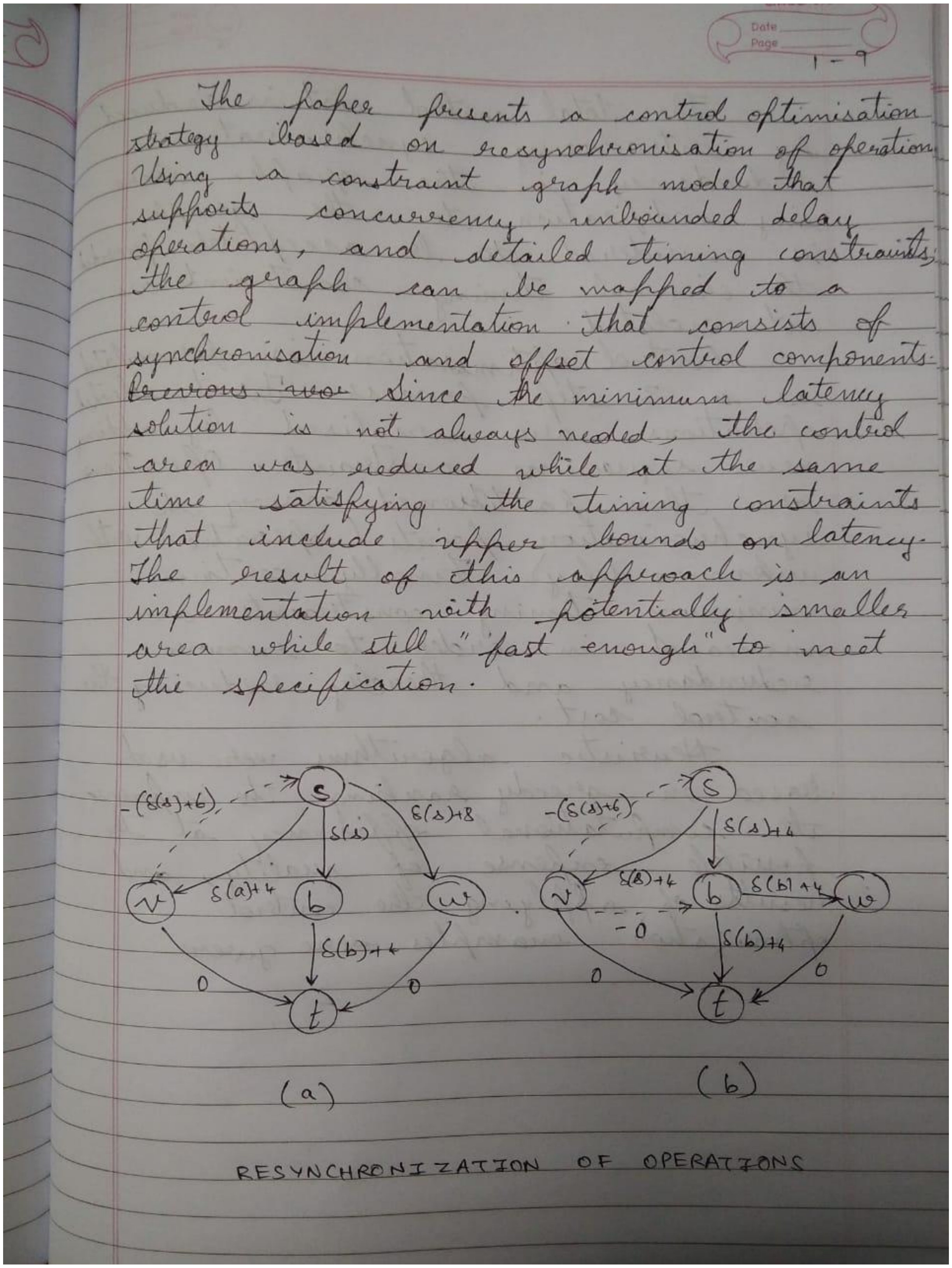
Name : Shawn Louis
Batch : B
Roll No : 31

Title : <u>Optimizing the Control-unit Through the Resynchronization of Operations</u>

Focus on : Control Optimization

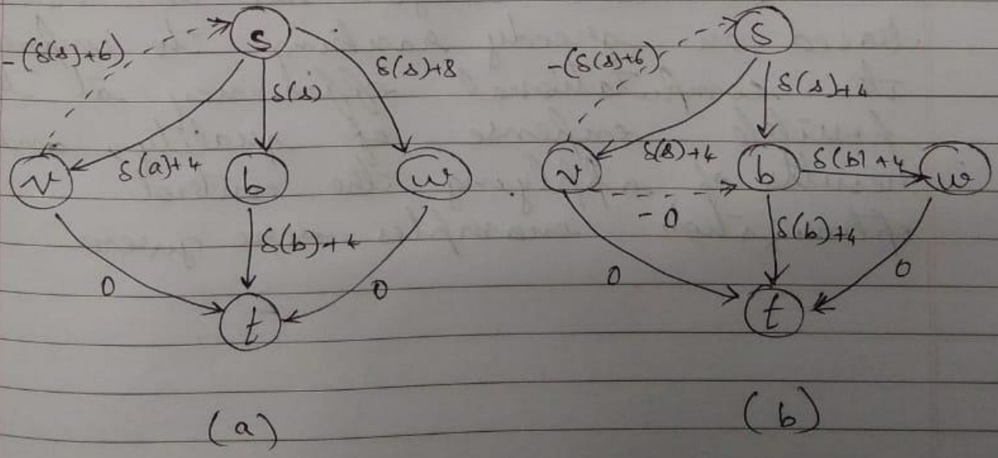
(Unit 5 in paper) (pg. no. 18)

Report:



Date
Page 1 – 9

The paper presents a control optimisation strategy based on resynchronisation of operation. Using a constraint graph model that supports concurrency, unbounded delay operations, and detailed timing constraints, the graph can be mapped to a control implementation that consists of synchronisation and offset control components. ~~Previous wor~~ Since the minimum latency solution is not always needed, the control area was reduced while at the same time satisfying the timing constraints that include upper bounds on latency. The result of this approach is an implementation with potentially smaller area while still "fast enough" to meet the specification.



RESYNCHRONIZATION OF OPERATIONS

The total control cost is reduced by introducing synchronisation redundancy in the graph, where any modification to the graph is considered to be acceptable provided the original timing constraints are not violated.

Control optimisation is formulated as a 3 step process. First, selected operations are serialised by adding new sequencing constraints. Operations are then partitioned among the synchronisation points (operations with unknown delay). Finally, certain minimum timing constraints are increased in order to remove redundancy and thereby reducing the control cost.

Heuristic algorithms were used based on greedy ranking to improve the computational efficiency at the possible expense of quality, and results of applying the control optimization examples were given.

Optimizing the Control-unit Through the Resynchronization of Operations

David Filo David C. Ku Giovanni De Micheli

Center for Integrated Systems

Stanford University

Stanford, CA 94305

Short title: “Optimizing Control Via Resynchronization”

Correspondence: Professor Giovanni De Micheli

Center for Integrated Systems, Room 129

Stanford University

Stanford, California, 94305-4055

(415) 725-3632

Abstract

Most approaches to control-unit 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.

Keywords: control synthesis, synchronization, relative scheduling, timing constraints.

1 Introduction

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

The minimization of the control-unit area, referred to as control optimization, can be performed either at the logic level by using a *finite-state machine* model [2, 3] or at a higher level by using a hardware model described in terms of constraints on the sequencing and timing of the operations [4]. 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 [5]. 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*. The use of concurrency, detailed timing constraints, and external synchronizations (unbounded delay operations) is supported. After an overview of the hardware and control model, we present the concept of *synchronization redundancy* in Section 4. We show how redundancies can be introduced by delaying operations through lengthening and serializing a graph-based hardware model. Section 5 describes the optimization algorithms. We conclude with results of applying the technique in the framework of the *Hercules/Hebe* High-level Synthesis system.

2 Hardware Model

Most existing high-level synthesis systems model hardware behavior as control/data flow graphs [6, 7]. We model hardware timing behavior as a partial order among a set of operations, and we represent it as a polar, hierarchical *constraint graph* $G(V; E)$. The edges, used to represent timing and sequencing constraints, are directed with edge weights. The graph is hierarchical to support the description of nested *model calls*. Conditional constructs are represented by conditional model calls where separate models are associated with each condition. Iteration is expressed as repeated calls to conditional constructs through the use of hierarchy. Modeling iteration in this manner results in the important property that the graph induced by the forward edges of the constraint graph is *acyclic*.

In this paper, we focus on one level of the hierarchy because control synthesis and optimization can be performed by traversing the hierarchy in a bottom-up fashion. The reader is referred to [8] for further details on the constraint graph model. We now describe the properties of the model that are important in the control formulation.

The vertices V of the graph represent the operations, and the edges E capture the precedence and timing relationships among the operations. Each vertex $v\ 2\ V$ has an associated operation which is synchronous and takes an integral number of cycles to execute. This is called its execution delay and is denoted by (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*.

A weight w_{ij} is associated with each edge $e_{ij} = (v_i; v_j)\ 2\ E$. A positive (negative) weight represents a requirement that the *start time* of v_j, denoted by $T(v_j)$, must occur at least (at most) w_{ij} cycles after the start time of v_i. Positive (negative) weights are associated with forward (backward) edges and represent minimum (maximum) timing constraints. Edges with bounded weight represent minimum and maximum delays between the *start* of two operations. Both forward and backward edges may have unbounded weights. The edge weight w_{ij} for an unbounded forward edge e_{ij} is of the form $(v_i) + k;\ k\ 0$, where k is the minimum number of cycles between the *completion* of v_i and the start of v_j. The edge weight w_{ji} for an unbounded backward edge e_{ji} is of the form $(\ (v_i) + k);\ k\ 0$, where k is the maximum number of cycles between the *completion* of v_i and the start of v_j. A backward unbounded edge e_{ji} is required to have a corresponding forward unbounded edge e_{ij}.

For example, consider the constraint graph in Figure 1, where s is the source, t is the sink, and double circles represent operations with unbounded delay. The operation c must begin at least 3 cycles and at most 5 cycles after operation b begins execution, which is specified by the equation $T(b)+3\ \ T(c)\ \ T(b)+5$. Similarly, operation d must begin at least 3 cycles and at most 5 cycles after the *completion* of operation b, given by the equation $T(b) + (b) + 3\ T(d)\ T(b) + (b) + 5$. In this case (b) represents the execution delay of operation b, where the delay is not fixed. For example, b could represent a *wait* operation on an external synchronization signal.

The graph model used here 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 performed and that resource conflicts have been resolved prior to control synthesis by serializing between the conflicting operations [9].

3 Control Generation

To define the cost function that will drive the control optimization, we first describe the mapping from a constraint graph model of hardware behavior to a control implementation. The mapping involves two tasks: scheduling and control generation. Given a constraint graph, *scheduling* determines the start times of the operations such that the timing constraints are satisfied. These start times are then used by *control generation* to derive an FSM specification of the control.

3.1 Scheduling the Operations

Traditionally, the scheduling problem assigns operations to control states. It can be formulated as an integer labeling problem that assigns to each operation an integer value representing the time at which it begins execution from the beginning of the schedule. The presence of *unbounded delay operations* in our hardware model invalidates this traditional scheduling formulation since an absolute schedule satisfying the timing constraints no longer exists. Instead, a formulation called *relative scheduling* is used 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. We refer those unfamiliar with relative scheduling to [10, 11].

We define a subset of the vertices called **anchors**. These vertices serve as reference points in specifying the activation of operations. The anchors of a constraint graph *G(V; E)*, denoted by *A V* , consist of the source vertex and all vertices corresponding to operations with unbounded delay. The **anchor set** of a vertex *v* is the subset of anchors *A(v) A* such that a path of forward edges exists from an anchor *a 2 A(v)* to *v* containing at least one unbounded edge weight equal to *(a)*. In other words, the anchor set contains all of the unbounded delay operations that may affect the activation of the operation associated with *v*.

The relative scheduling formulation can be used to derive an implementation with minimum execution delay. This is achieved by constructing an *as soon as possible* (ASAP) schedule for the graph. For each vertex $v\ 2\ V$, a *start time* $T(v)$ is defined in terms of the anchor offsets $f_a{}^{min}(v)\ j\ a\ 2\ A(v)g$, where ${}_a^{min}(v)$ represents the minimum time at which v can begin execution after the completion of a.

$$T\,(v) = \max_{a2A(v)} fT\,(a) + \ (a) + {}_a^{min}(v)g$$

For example, in Figure 1 the start time of vertex d is given by the expression $T(d) = \max fT(s) + \ (s) + 3; T(b) + \ (b) + 3g$. Defining the start time in this manner results in a *minimum relative schedule*, which is guaranteed to minimize the source to sink latency for all profiles of execution delays $f\,(a); 8a\,2\,Ag$ [10].

For a particular anchor a, ${}_a^{max}$ represents its maximal offset and is given by the equation:

$${}_a^{max} = \max f\,{}_a^{min}(v)\ j\ a\ 2\ A(v);\ 8v\ 2\ V\,g$$

This number represents the largest offset for anchor a needed by some vertex v to compute $T\,(v)$. For example, in Figure 1 the maximal offset of anchor b is ${}_b^{max} = \max f\,{}_b^{min}(d); {}_b^{min}(e)\ ; {}_b^{min}(t)g = \max f3; 5; 5g = 5$, which is needed for the computation of $T(e)$ and $T(t)$. The maximal offset of an anchor is of interest because it is used to estimate hardware costs as described later.

In some cases the use of an ASAP schedule is not necessary in order to satisfy the given constraints. The use of such a schedule can sometimes lead to a higher control cost when compared to a non-ASAP schedule. The goal of our work is to find a schedule such that the control-unit area is minimized while still satisfying all of the constraints. In order to take advantage of the work already done in deriving ASAP schedules, we minimize area by first modifying the original constraint graph and then applying relative scheduling techniques to obtain a circuit implementation. Modification of the constraint graph is achieved through the *delaying* and *resynchronizing* of some operations. This results in hardware minimized for area that is not necessarily an ASAP implementation of the specification.

A constraint graph is **feasible** if its constraints can be satisfied when the unbounded delays are equal to zero. Figure 2(a) shows an example of a feasible constraint graph. If the unbounded delays are assumed to be equal to zero (i.e. $\ (s) = \ (b) = \ (c) = 0$), the constraints can be satisfied by assigning the following start times: $T\,(s) = T\,(b) = T\,(c) = 0;\ T\,(d) = 3;\ T\,(e) = 4$. 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 and only if it contains no positive length cycles [1, 10].

In the presence of unbounded delays, however, we extend the analysis. A graph is **well-posed** if a schedule based on the start times defined above exists such that the constraints can be satisfied *for all values* of unbounded delays; otherwise, it is called **ill-posed**. The graph in Figure 2(a) is clearly ill-posed, because for all schedules there exists values for the unbounded delay *(c)* that would violate the maximum timing constraint e_{ec}, namely *(c)* > 3.

A graph is well-posed provided the following three conditions hold:

No positive cycles exist with unbounded delays set to zero.

No positive unbounded length cycles exist.

$A(v_i)$ $A(v_j)$ for all edges e_{ij} 2 E.

The third condition, *anchor containment* is necessary to ensure that an ASAP schedule, based on start times as defined above, is valid for all profiles of execution delays. If a specification violates one of the first two conditions, then there does not exist a valid solution. However, violation of the *anchor containment* property can be fixed by adding new edges to the constraint graph. For example, in Figure 2(b) anchor containment is violated because $A(e) = fs; cg$ 6 $A(d) = fs; bg$; however, this problem is solved in Figure 2(c) by the addition of e_{cd}, which results in $A(e) = fs; cg$ $A(d) = fs; b; cg$. The process of adding these edges to satisfy anchor containment is done by a procedure called *make-well-posed* described in [10]. The following theorem implies that the anchor containment property can always be satisfied provided the first two conditions hold.

Theorem 3.1 *A feasible constraint graph G(V; E) can be made well-posed if and only if no unbounded length cycles exist in G.*

Proof: In [10]. *k*

The concept of well-posedness is important to ensure that the resulting synthesized hardware is valid for all possible input conditions. Note that well-posedness implies feasibility, and non-feasibility implies ill-posedness. We assume that the input to the control optimization process is a well-posed constraint graph. Since all transformations presented here preserve the three conditions described above, our optimization process maintains the well-posed property of the constraint graph. The first two conditions must be checked any time an edge is added or an edge weight is changed. Since the anchor set of a vertex can only be changed as a result of the addition of an edge in the graph, the anchor containment property need only be checked when forward edges are added.

A block diagram of our approach is shown in Figure 3. The diagram in (a) represents the previous approach, which minimizes the execution delay of the resulting implementation. The approach taken in this paper, shown in (b), is to add an optimization step to reduce the control area. This additional step takes a well-posed graph as input, performs graph transformations, and produces a well-posed graph, which is then implemented using an ASAP schedule.

3.2 Generating the Control Circuit

Given a schedule, the task is to generate the corresponding control logic to activate operations according to the schedule. 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. We abstract the task of control generation as generating control signals for each vertex v where the activation (completion) of v is indicated by the assertion of the signal $enable_v$ ($done_v$). We assume that the completion of the unbounded delay operation corresponding to each anchor $a\ 2\ A$ is indicated by the assertion of a signal $done_a$, which stays asserted until all operations in the graph have completed.

We illustrate two possible alternatives for control generation in Figure 4 with 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 $_a(v) = 2$ and $_b(v) = 3$. That is, operation v must start at least 2 and 3 cycles after the completion of operations a and b respectively. In the first case, binary counters, which are initially cleared, are used to count the number of cycles since the completion of the anchors a and b. Comparators are then used to determine when the operation v should be activated. For the second case, shift registers are used instead of binary counters. This results in more registers but the need for comparators has been eliminated.

The control circuit can be modeled as consisting of two components: an *offset control* for each anchor $a\ 2\ A$ and a *synchronization control* for each vertex $v\ 2\ V$. The offset control for a generates the time offsets with respect to the completion of a. We abstract the offset control as an FSM that is activated by the assertion of $done_a$. The FSM generates a set of signals $C_a(i);\ 1 \quad i \quad {}^{max}_{a}$, where $C_a(i)$ is asserted when *at least i* cycles have elapsed after the completion of a. The synchronization control for v synchronizes the activation of v, denoted by $enable_v$, to offsets from the completion of its anchors.

Specifically, the enable signal is defined as $enable_v = \prod_{a \in A(v)} C_a(\ _a(v))$. Figure 5 shows the block diagram of the offset control. The number of states in the offset control for a is equal to the maximum value of the offsets of anchor a, ${}^{max}_{a}$. The details of generating the signal $done_a$ as well as the control logic for conditional branching and looping are described in [11].

3.2.1 Cost Function

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

$$\begin{aligned} COST_{area} &= \sum_{a \in A} COST_{off}(a) + \sum_{v \in V} COST_{sync}(v) \\ &= \sum_{a \in A} f_{off}({}^{max}_{a}) + \sum_{v \in V} f_{sync}(|A(v)|) \end{aligned}$$

The first term $COST_{off}(a)$ is the cost due to the offset control for an anchor $a \in A$ and is related to the cost due to the length of the schedule corresponding to a. The cost is a function f_{off} of the maximal offset value ${}^{max}_{a}$ for a that yields the number of registers implementing the offset FSM for a. The second term $COST_{sync}(v)$ is the synchronization control cost for $v \in V$ and depends on the number of signals that must be synchronized to activate v. The cost is a function f_{sync} of the cardinality $|A(v)|$ of the anchor set of v. The values and represent appropriate weight factors related to the actual cost of the logic implementation.

For the two alternate implementation styles shown in Figure 4, is the cost of a register, and is the cost of a literal. In the counter-based implementation of Figure 4(a), the complexity of the offset control is a logarithmic function of the maximal offset values, i.e. $f_{off}(n) = \lceil \log_2(n) \rceil$ represents the number of registers. The complexity of the synchronization control is a linear function of the number of comparators, i.e. $f_{sync}(n) = n$ represents the number of literals in the required comparators. In the shift-register based implementation of Figure 4(b), f_{off} is now a linear function of the maximal offset values, i.e. $f_{off}(n) = n$. Since comparisons are no longer needed in the synchronization, $f_{sync}(n)$ represents the number of literals in a n-input AND gate.

In addition to the two examples shown here, there exists many other possible implementations of the control logic. Specifically, it is possible to specify the control as finite state machines with different state encodings. The examples shown in Figure 4 correspond to the two limiting cases of binary and one-hot encodings.

In all formulations, the control complexity can be reduced by either minimizing the *maximal offsets* and/or by reducing the size of the *anchor sets*. Our approach to control optimization focuses on these two aspects. Such reductions can be achieved by *modifying* the graph topology and/or the edge weights. These techniques will be described in the following sections.

4 Redundancy in Synchronization

It is often the case that some anchors in the anchor set of a vertex v are not needed in the evaluation 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 synchronization dependencies. Consider the example in Figure 6(b) with two anchors a and b, a vertex v, and arcs that represent minimum timing requirements between the operations. Vertex v can execute 3 after the completion of a and 1 cycle after the completion of b. Using the previous definition for start time, these requirements result in the following start time for v:

$$T(v) = \max f T(a) + (a) + 3; T(b) + (b) + 1 g$$

It is important to emphasize that this start time can only be dynamically determined because the delays (a) and (b) are not known at synthesis time. Substituting the value of $T(b)$ in the equation above, we get:

$$T(v) = \max f T(a) + (a) + 3; T(a) + (a) + 2 + (b) + 1 g$$

This can then be reduced to:

$$T(v) = T(a) + (a) + 2 + (b) + 1 = T(b) + (b) + 1$$

Thus we conclude that the constraint corresponding to edge e_{av} is not necessary to compute $T(v)$. This is because the remaining edges ensure that v will not start before 3 cycles after the completion of a. The dependency on a exists implicitly through the dependency on anchor b. In this case the anchor a is said to be redundant with respect to the vertex v, which implies time offsets from anchor a are not needed in computing the start time $T(v)$.

In contrast, all edges are necessary in Figure 6(c). Removal of any edge would result in a violation of the timing constraints for some values of (a) and (b). In this case, we say that anchor a is irredundant with respect to v, which means that offsets from a must be used to compute $T(v)$.

More formally, let $(u; w)$ denote a path from u to w in G, and let $(u; \dots ; v; \dots ; w)$ denote a path from u to w containing v. The length of $(u; w)$, denoted by $j\ (u; w)j$, is the sum of the edge weights on the path where all unbounded execution delays are set equal to zero. A *longest path* $lp(u; w)$ is defined to be a path from u to w whose length is at least as long as all paths $(u; w)$ from u to w.

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, where $q\ 2\ A(v)$ and $a\ 2\ A(q)$, such that $jlp(a; v)j =$[1] $jlp(a; q)j + jlp(q; v)j$; otherwise it is **irredundant**. The irredundant anchor set $IR(v)$ of a vertex v consists of the irredundant anchors in the anchor set of v. It is the minimum set of synchronizing points affecting the activation of v. In Figure 6(b) the irredundant anchor sets are $IR(v) = b$ and $IR(b) = a$. The constraints on the start time of a vertex v are identical whether or not redundant anchors are included in the start time computation. The proof for this can be found in [10].

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

4.1 Making Anchors Redundant

Consider an anchor a that is irredundant 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 a new sequencing dependency in the constraint graph. We define a serialization of a graph G to be the creation of a new sequencing dependency that did not exist in the original graph. Lengthening a graph G is defined to be the increase of a minimum timing constraint between two vertices.

In practice, lengthening a constraint between two vertices w and t can be achieved in two different ways, as shown in Figure 7. The obvious method, illustrated in Figure 7(b), is to increase existing edge weights on edges (shown in bold) of a path $(w; t)$. An alternative method, shown in Figure 7(c), consists of the following two steps. First an edge e_{wt} that has an associated edge weight $w_{wt} = jlp(w; t)j$ is added to the constraint graph. This edge weight is then increased by the appropriate amount to achieve the desired lengthening. The second method is used at times to eliminate the unwanted side effect of lengthening other paths unnecessarily. For example, the path $(v; t)$ is lengthened as a side effect in

[1]Intuitively, it may seem that this relation should be . However, $<$ is not possible, because a longest path $lp(a; v)$ must be at least as long as every path $(a; v)$.

Figure 7(b).

We illustrate in Figure 8 how an anchor *a* can be made redundant with respect to vertex *t* using the serializing and lengthening techniques. The original graph in Figure 8(a) is modified by adding a serialization edge e_{ab} from *a* to *b* in Figure 8(b). The addition of this edge makes the edge e_{sb} redundant. The edge e_{ab} is then lengthened in Figure 8(c) to make the edge e_{at} redundant. Since *a* is now redundant with respect to *t*, the edge e_{at} can be removed from the graph without changing the constraints on $T(t)$. Because $T(t)$ no longer explicitly depends on the completion of anchor *a*, the control cost is reduced in two ways. For the graph in (a), the start time is given by:

$$T(t) = \max f T(a) + (a) + 4; T(b) + (b) + 2g$$

However, for the graph in (c), the start time is reduced to:

$$T(t) = T(b) + (b) + 2$$

Since this calculation is done dynamically during execution, the synchronization logic for *t* is reduced. Furthermore, the maximal offset $^{max}_{a}$ is reduced from 4 to 2 which results in a smaller number of registers needed by the offset generation for *a*.

The delaying of operations via serialization and lengthening must be carried out with care to avoid violating the constraint graph, i.e. making the resulting graph ill-posed. Both lengthening and serializing can be viewed as adding a forward edge e_{av} representing a sequencing constraint from anchor *a* to vertex *v*. This edge has a non-negative edge weight $w_{av} = (a) + k, k \quad 0$. If a path already exists from *a* to *v* containing (a), then the new edge implies lengthening; otherwise, it implies serializing.

Before proceeding, we state the following theorem, which provides the basis for determining the validity of transformations. They relate the concept of feasibility (satisfaction of constraints when $(a) = 0; 8a\ 2\ A$) to the concept of well-posedness (satisfaction of constraints for any $(a); 8a\ 2\ A$).

Theorem 4.1 *Consider a well-posed constraint graph $G(V;E)$. If a forward edge e_{av} from anchor a to vertex v with weight $w_{av} = (a) + k; k \quad 0$ is added from anchor a to vertex v, then the resulting graph $\hat{G}$ can be made well-posed if and only if:*
(1) $\hat{G}$ is feasible, and
(2) $\hat{G}$ does not contain unbounded length cycles.

Proof: This follows directly from Theorem 3.1. For the sufficient condition, if $\mathcal{G}$ is feasible and does not contain unbounded length cycles, then from Theorem 3.1 we know that $G(V; E)$ can be made well-posed. Now for the necessary condition. If $\mathcal{G}$ can be made well-posed, then it is feasible. Furthermore, Theorem 3.1 implies that no unbounded length cycles exist in $\mathcal{G}$. *k*

This theorem ensures that a constraint graph can be made well-posed provided no positive cycles and no unbounded length cycles are created during control optimization. Furthermore, if it can be shown that the anchor containment property holds, then it follows that the resulting graph *is* well-posed.

4.2 Prime versus Non-prime Anchors

If an operation is delayed by *lengthening* existing paths in the constraint graph without introducing new serializations, then the anchor sets of the vertices remain unchanged. The reason is because the transitive sequencing dependencies are not affected by lengthening paths. It is useful to identify the subset of anchors in the anchor set of a vertex which can be made redundant by lengthening.

Definition 4.1 *An anchor a 2 $A(v)$ of a vertex v is* **prime** *if a 2̸ $A(q)$; 8q 2 $A(v)$. Otherwise, the anchor is* **non-prime**. *The set of prime anchors of a vertex v is the* **prime anchor set** *of v, denoted by $PA(v)$.*

This means that for a prime anchor a 2 $PA(v)$, all paths of forward edges from a to v contain no unbounded delay weights other than the delay of the prime anchor (a). From the definition of anchor redundancy, it follows that a prime anchor is irredundant. For example, in Figure 8(a), both a and b are prime anchors of vertex t, because there exists no other anchor on any path from a or b to t. On the other hand, anchor a in Figure 8(b) is *not* a prime anchor of t, because there exists a forward path from a to t that contains an unbounded edge weight other than (a), namely (b). In this case it is possible to lengthen the graph to make a redundant with respect to t, as shown in Figure 8(c).

Given a vertex v and an irredundant anchor a 2 $A(v)$, if a is non-prime, it can be made redundant by lengthening a path from a to v such 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 to the vertex, for all vertices. We state the following lemma.

Lemma 4.1 *Consider a vertex v 2 V of a well-posed graph, a non-prime anchor a 2 $A(v)$, and a prime anchor b 2 $A(v)$. The anchor a can always be made redundant with respect to v by lengthening the minimum timing requirements from a to b and from b to v. Furthermore, the resulting constraint graph remains well-posed.*

Proof: We prove by construction. First, let P denote the set of all paths from a to v containing an edge with weight (a). If a longest path $lp(a; v)$ 2 P contains an unbounded delay weight other than (a), then a is already redundant. Otherwise, a path $(a; b; v)$ 2 P containing (b) will be lengthened to make a redundant. This lengthening will be done in two steps. First the edges e_{ab} and e_{bv} are added with the weights $w_{ab} = (a)$ and $w_{bv} = (b)$. Then these weights are increased so that j $(a; b; v)j = jlp(a; v)j$, thus making a redundant with respect to v.

Assume first that there do not exist paths $(b; a)$ and $(v; b)$. Then the corresponding edge is *unconstrained* and its weight can be made arbitrarily large. This allows for $(a; b; v)$ to be lengthened to $jlp(a; v)j$ and the proof is complete. On the other hand, if both edges are constrained, then their weights can be increased to the values $w_{ab} = (a) + jlp(b; a)j$ and $w_{bv} = (b) + jlp(v; b)j$ so that two zero length cycles are created. These cycles have zero length because forward paths $(a; b)$ and $(b; v)$ exist, which implies that both $jlp(b; a)j$ and $jlp(v; b)j$ are negative.

For the sake of contradiction, assume that this lengthening does not make a redundant. Since a is not redundant, it follows that $l = j$ $(a; b; v)j < jlp(a; v)j$. However, a zero length cycle exists between a and v which implies that there exists a path of length l from v to a. By combining this path of length l from v to a along with the path $lp(a; v)$, a positive cycle exists since $l < jlp(a; v)j$. Since no positive length cycles are created, it follows that the original graph contains a positive cycle. By contradiction a must be redundant. Hence, a lengthening does exist. Furthermore, since no positive cycles are created and the anchor sets for all vertices remain the same, it follows that the resulting graph is well-posed. k

The above lemma implies that any non-prime anchor can be made redundant with respect to its associated vertex. Lengthening can be repeatedly applied until all of the non-prime anchors of a vertex v 2 V have been made redundant. We now show that there exists an ordering of the anchors such that a lengthening to make one anchor redundant does not cause a previous anchor in the ordering to become irredundant.

Theorem 4.2 *Given a well-posed constraint graph $G(V; E)$, there always exists a lengthening of G, denoted by $\overset{\ell}{G}$, such that $\overset{\ell}{G}$ is taut and well-posed.*

Proof: We prove by construction. The graph is traversed in topological order starting from the source vertex. When an anchor $a\ 2\ A$ is traversed it is made redundant to all vertices to which is non-prime and irredundant. We claim that making a redundant in this manner will not cause a previous anchor $c\ 2\ A$ in the ordering to become irredundant with respect to some vertex to which it is non-prime.

Suppose anchor a is to be made redundant with respect to a vertex v. According to Lemma 4.1, this can be done by increasing unbounded edge weights of the edges e_{ab} and e_{bv} where $b\ 2\ A(v)$ is a prime anchor of v. We assume for the sake of contradiction that some previous anchor c in the ordering has inadvertently been made irredundant with respect to some vertex w. That is, the length of a longest path $lp(c; w)$ from c to w has increased. Since c is now irredundant with respect to w, it follows that the path $lp(c; w)$ contains exactly one unbounded delay weight (c). Every path that has been increased must contain at least one of the unbounded weights (a) or (b) because lengthening is accomplished by increasing the weight on an edge from one these anchors. Hence, this implies that $c = a$ or $c = b$. However, this means that c has not yet been traversed. Thus, by contradiction of the assumption, the proof is complete. It follows from Lemma 4.1 that the resulting graph is well-posed. *k*

We point out that the prime anchor sets are *fixed* for a given graph topology. Furthermore, they remain unchanged when the graph is lengthened. However, it is possible to change the prime anchor sets by serializing the graph, which alters the transitive sequencing dependencies. We use a combination of graph serialization and graph lengthening (as shown in Figure 8) to introduce redundancy, which can be removed to optimize the control. This approach is described next.

5 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, only the irredundant anchors of an operation represent its *synchronization points*, i.e. they synchronize the execution of an 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 as the resulting graph remains well-posed. Note that since we never remove any dependencies or constraints in the original graph, the resulting graph still satisfies the original behavior specification.

The two factors in the control cost, synchronization and offset costs, are tightly coupled. The reduction of one may result in the increase of the other. Furthermore, 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 heuristic strategy. This strategy is based on the observation that non-prime anchors can always be made redundant, and that a prime anchor of a vertex can be made non-prime by serializing it with respect to another prime anchor.

1. *Minimize the prime anchor sets* – by serializing the anchors.
2. *Resynchronize operations with respect to new synchronization points* – by serializing the operations with respect to the anchors to minimize the offset control cost.
3. *Remove redundancy* – by lengthening the graph to make it taut, which minimizes 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. The three steps in this approach are now described in the following sections.

5.1 Minimize Prime Anchor Sets

We first serialize the anchors to make as many anchors of a vertex non-prime as possible. Since non-prime anchors can be made redundant by lengthening, this has the effect of reducing the synchronization cost of the final control implementation. This is particularly important for control-dominated designs because the synchronization cost may dominate the overall control cost in certain implementation styles.

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. In this case, the anchors are *completely* serialized with respect to one another. In the presence of maximum timing constraints, however, it is not always possible to arbitrarily serialize two anchors without violating timing constraints. Specifically, two anchors connected by a cyclic timing relationship cannot be serialized because an unbounded length cycle would be formed. This is illustrated in Figure 9 where anchors $a, b \in PA(v)$ cannot be serialized and, therefore, cannot be made non-prime with respect to t. To address this issue, we define an *anchor cluster* as follows.

Definition 5.1 *An* **anchor cluster**, *denoted by* $_i$, *is a maximal subset of strongly connected anchors in the constraint graph.*

The set of anchor clusters is denoted by $\{ _0, _1, \ldots, _k\}$, where $_0$ is the cluster containing the source vertex. A constraint graph is called **elementary** if all anchor clusters contain a single anchor, i.e. $| _i| = 1, \forall i$.

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 Figure 9, there are two anchor clusters $_0 = \{s\}$ and $_1 = \{a, b\}$, and they 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 5.2 *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 $_i$ and $_j$, every anchor $a \in _i$ is serialized with respect to every anchor $b \in _j$. The graph G with a cluster ordering is called an* **ordered** *graph.*

Since only the irredundant anchors are necessary in the start time computation of a vertex, serializing the clusters can reduce the synchronization requirement by decreasing the number of irredundant anchors for a vertex. In order to define a lower bound on the synchronization cost, we present the following theorem. It applies to those graphs that are ordered.

Theorem 5.1 *For a well-posed, ordered, elementary constraint graph, the sum of the cardinality of the prime anchor sets for all vertices is equal to $|V| - 1$.*

Proof: $|PA(v)| = 1, \forall v \in V$ except $PA(v_0) = 0$. This implies that $\sum_{v \in V} |PA(v)| = |V| - 1$. ∎

Since we can make a graph taut, the theorem above implies 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 that, by definition, anchor clusters are not connected by any cycle in the constraint graph. Therefore, no cycles can be formed by serializing between anchors in different clusters.

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 orders the anchor clusters by ranking them with respect to increasing lengths of the longest path from the source vertex. This ranking is then used to convert the original partial order to a complete order by serializing the clusters. Cluster serialization is done by add sequencing dependencies from the anchors in one cluster to those anchors in the following cluster. As noted above, this serialization step does not create any cycles. Furthermore the anchor containment property is preserved because edges are added only between anchors. For example, in Figure 13(a) the longest paths from the source anchor *a* to the three anchors *b*, *c*, and *d* are computed. This results in the cluster ordering indicated by the shaded arrows.

5.2 Partition to Resynchronize Operations

Informally 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 a cluster ordering, we *partition* the operations and assign them to the links of this chain in order to minimize the offset control cost. The process of assigning operations to links involves serializing the operations with respect to the anchor clusters, and it is called **resynchronization** since the activation of operations will depend on a possibly new set of synchronization points. We formalize this idea with the following definition.

Definition 5.3 *Given a well-posed, ordered constraint graph with two consecutive anchor clusters* $_i$ *and* $_{i+1}$*, consider all paths from the anchors in* $_i$ *to the anchors in* $_{i+1}$ *such that each path has a single unbounded edge weight corresponding to an anchor in* $_i$*. The subgraph induced by the vertices on these paths is defined to be a* **cluster link** L_i *corresponding to* $_i$*. A* **segmented** *graph is a graph for which all vertices belong to cluster links.*

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

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 the anchors of non-consecutive clusters. Consider the ordered, elementary graph in Figure 13. In (a), only v_2 and v_5 belong 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.

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. For example, in Figure 13(a), the vertices $f v_3; v_4 g$ form a c-opset. A trivial c-opset is one containing a single vertex. We state without proof that all elements of a c-opset must be assigned to the *same* link in the chain if the resulting graph is to be well-posed. If two operations in a c-opset belonged to different links, a positive unbounded cycle would exist.

We now present a heuristic algorithm, called *ResyncGraph* shown in Figure 10, that assigns c-opsets to the links of the cluster chain. The anchor clusters are traversed in topological order. For each cluster $_i$, all of the vertices v are examined that have a prime anchor in the cluster (i.e. $a\ 2\ P\ A(v)$ and $a\ 2\ _i$). If there exists a path from v to any anchor in the following cluster $_{i+1}$, then v must be assigned to the cluster $_i$; otherwise, an unbounded cycle would result. This assignment is done by adding a backward edge from v to each anchor in the cluster $_{i+1}$. Note that a forward path from $_i$ to v already exists because the prime anchors $PA(v)$ are contained in $_i$. Appropriate weights are given to the added backward edges so that no positive cycles are created. The reason for adding these backward edges is to delay the cluster $_{i+1}$ as much as possible without increasing the maximal offsets for the anchors in the previous cluster $_i$. Consider the graph in Figure 11(a), which has two clusters $_1 = f s g$ and $_2 = f b g$. Vertex v must be assigned to cluster $_1$ because there exists a path from v to b in (a). Therefore, the backward edge e_{vb} is added, which has the effect of increasing the edge weight w_{sb} from $(\ (s) + 0)$ to $(\ (s) + 4)$. The maximal offset for s does not change and remains at 4.

If there exists no paths from v to $_{i+1}$, then serialization edges are added from the anchors in $_{i+1}$ to v. This *resynchronizes* v to a different anchor cluster by changing its prime anchor set so that

P $A(v)$ $_{i+1}$. This has the effect of amortizing the offsets needed for T (v) over consecutive anchor clusters. For example, in Figure 11(a) no path exists from w to s so the edge e_{bw} is added. This edge is later lengthened to make e_{sw} redundant. The combination of adding backward edges and resynchronizing operations has reduced the maximal offset $^{max}_{s}$ from 8 to 4. This in turn reduces the register requirement needed by the offset generation for anchor s. Neither of these steps by itself would have achieved such a result. In both of these cases edges are added, but the type and weight of an added edge is such that no bounded or unbounded positive cycles are created. Furthermore, forward edges are added in a way that preserves the anchor containment property.

5.3 Making the Graph Taut

The final step in the optimization strategy is to remove redundancies by making the constraint graph *taut*. This is done by making the non-prime anchors of a vertex redundant by lengthening the graph appropriately. However, it is possible to increase the maximal offset values as a result of lengthening. It was shown in Section 3.2.1 that such an increase has a direct effect on the hardware control cost. Therefore, an important criterion in lengthening is to minimize the *amount* of increase. The algorithm for lengthening, called *LengthenTaut*, is shown in Figure 12.

The algorithm for lengthening visits each anchor a in the graph and lengthens the paths to vertices v, where a is a non-prime irredundant anchor of v, in order to make a redundant. For a vertex v and a non-prime anchor a 2 $A(v)$, a forward path $(b_0;$ $; b_n)$ is found where $b_0 = a$, $b_n = v$, and b_i 2 $A(v)$ for 0 $i < n$. Furthermore the path contains the unbounded edge weights (b_i) for 0 $i < n$. This path is lengthened by traversing the anchors of the path starting with a. Edges are added between consecutive anchors, and their weights are increased as much as possible until either a maximum constraint limit is reached, or until a is made redundant with respect to v. Lengthening in this way does not introduce any bounded or unbounded positive cycles. Since lengthening does not introduce a forward path between two vertices where such a path did not previously exist, it follows that the anchor containment property is preserved by this operation.

By Theorem 4.2 the algorithm is guaranteed to make a constraint graph taut. In addition, the algorithm also guarantees for *elementary* constraint graphs, the control offset cost is always reduced, or in the worst case remains the same. In order to determine the effect of these steps on the control cost, we state the following lemma and theorem.

Lemma 5.1 *Consider a well-posed constraint graph $G(V; E)$ and the set of maximal offsets $S =$ $f\ {}^{max}_{a}\ j\ a\ 2\ Ag$. An edge e_{bc} added between two anchors $b; c$ with an unbounded edge weight (b) can modify only the maximal offset ${}^{max}_{b}\ 2\ S$ provided the resultant graph remains well-posed.*

Proof: Only graphs that remain well-posed are considered here because the notion of a maximal offset is not well defined for ill-posed graphs. The maximal offset ${}^{max}_{b}$ of an anchor b can be computed by finding the longest path from b to all vertices v where $b\ 2\ IR(v)$. For the sake of contradiction, assume there exists an anchor $d\ 6{=}\ b$ such that ${}^{max}_{d}$ has increased as a result of adding e_{bc}. This implies that there exists $c\ 2\ A$, $w\ 2\ V$, and a path $(d; w)$ (containing a single unbounded delay (d)) that has increased in length. Since only the edge e_{bc} has been added, any path that has changed length must contain e_{bc}. Hence the increased path $(d; w)$ contains an unbounded delay weight other than (d), namely (b). By contradiction the proof is complete. Only graphs that remain well-posed are considered here because the notion of a maximal offset is not well defined for ill-posed graphs. k

Theorem 5.2 *Given a well-posed, elementary constraint graph $G(V; E)$, G can be made taut without increasing the maximal offset values of the anchors of G.*

Proof: Consider a vertex $v\ 2\ V$ and a non-prime anchor $a\ 2\ A(v)$ that is irredundant. Since a is non-prime, there exists a forward path $(a; b)$ to some anchor $b\ 2\ PA(v)$ containing the unbounded delay weight (a). By definition of irredundancy, a longest path $lp(a; v)$ from a to v contains only the one unbounded delay weight (a). It follows that $jlp(a; v)j > j\ (a; b)j + j\ (b; v)j$. By definition, $jlp(a; v)j\ {}^{max}_{a}$. An edge e_{ab} can be added with weight $w_{ab} = (a) + jlp(a; v)j\ j\ (b; v)j\ {}^{max}_{a}$.
Such an edge can always be added without making the resulting graph ill-posed. This is assured because a positive cycle could result only if a and b were already strongly connected. However, such a cycle cannot exist because the graph is elementary. Therefore, a has been made redundant without increasing its maximal offset. In fact, its maximal offset may actually have been reduced. According to the Lemma 5.1 we know that no other maximal offsets can possibly change, hence the proof is complete. k

According to the above theorem, given a well-posed, elementary constraint graph G, procedure *LengthenTaut* can make G taut without increasing the maximal offset values of the anchors of G.

5.4 Example

We illustrate the application of our strategy in Figure 13. The graph contains four anchors $fa; b; c; dg$, and five vertices $fv_1; v_2; v_3; v_4; v_5g$, where v_3 and v_4 form a c-opset. The graph is elementary since each anchor cluster contains a single element. The first step is to find the longest path from the source vertex a to all other vertices in the graph. Based on increasing length of these longest paths, a cluster ordering is imposed. This results in the cluster ordering $(a; b; c; d)$ corresponding to the longest path lengths 0, 5, 8, 18 as seen in Figure 13(a). This ordering is realized by adding serialization edges e_{ab}, e_{bc}, and e_{cd} with the respective edge weights of (a), (b), and (c).

The next step is to resynchronize the operations. The vertices are grouped into c-opsets, which are then assigned to cluster links. In this example there is only one non-trivial c-opset $fv_3; v_4g$. Resynchronization results in the c-opset assignments of fv_1g to L_b, fv_2g to L_a, $fv_3; v_4g$ to L_b, and fv_5g to L_c. Assignment of a vertex v to a link L_i is achieved by adding a forward edge from $_i$ to v and a backward edge from v to $_i$. For example, v_4 was assigned to its link by adding the edges $e_{b;v_4}$ and $e_{v_4;c}$ with edge weights $w_{b;v_4} = 0$ and $w_{v_4;c} = 1$.

Finally, redundancy is removed from the graph by lengthening it to make it taut. This results in the increased values of the edge weights $w_{ab} = (a)+5$, $w_{b;v_1} = (b)+2$, $w_{b;c} = (b)+8$, and $w_{c;d} = (c)+5$. Comparing the original graph with redundancies removed to the optimized graph, we see that the offset cost is reduced from $^{max}_{a} + {}^{max}_{b} + {}^{max}_{c} = 8+13+5 = 26$ to $5+9+5 = 19$. The synchronization cost is also reduced in the optimized graph, were all vertices depend only on a single synchronization point. So in this example, both components of the control cost have decreased. The graphic in Figure 14 represents both the offset generation and synchronization costs for an implementation based on shift registers as described in Section 3.2. The number of registers needed for offset generation and the synchronization logic are shown in the figure. The first row of numbers corresponds to the original graph. The costs in the second row have been reduced by removing the original redundancies. Finally, applying control optimization results in the final row of numbers.

6 Analysis

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, we now show that of all the possible minimum solutions for an elementary, ordered graph, 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.

Formally, we say a graph $\acute{G}$ *implements* another graph *G* if it is derived from *G* by serializing and lengthening, and we state the following key theorem.

Theorem 6.1 *Given a well-posed, ordered, elementary graph G with control cost COST^G. A well-posed* **segmented** *graph* $\acute{G}$ *can always be found such that* $COST^{\acute{G}}$ $COST^G$.

Proof: Consider a vertex *v 2 V; v 6= v*$_0$*;* of the graph *G*. By Theorem 5.2 we know we can lengthen the graph so that all non-prime anchors of *v* are made redundant. Furthermore, since *G* is elementary and ordered we know that *jP A(v)j* = 1. Let $_0$*;* $_1$*; . . . ;* $_k$ denote the anchor clusters that have been ordered. Since *G* is elementary, each cluster contains a single anchor. Consider two consecutive clusters $_i$ = *fag* and $_{i+1}$ = *fbg* such that *a 2 A(v)*. The length of a longest path *lp(b; v)* (if one exists) from *b* to *v* is denoted by l_{vb} = *jlp(b; v)j*.

In order to assign *v* to the link L_i, a maximal timing constraint e_{vb} is added. If a path of positive length from *b* to *v* exists, then w_{vb} = *jlp(b; v)j*, otherwise w_{vb} = 0. The value of the weight ensures that no positive cycle is formed. Note that l_{vb} is bounded, because if l_{bv} is unbounded, then *a* cannot be a prime anchor of *v*, which contradicts our assumption. Since no forward edges are added, the anchor sets for all vertices remain unchanged. Since *G* is well-posed, no anchor sets have been modified, and no positive cycles have been introduced, $\acute{G}$ is well-posed.

We know by definition that *jlp(a; v)j; jlp(a; b)j* $^{max}_a$. Adding e_{vb} creates a new path from *a* to *b* with length *jlp(a; v)j* + w_{vb} *jlp(a; v)j* $^{max}_a$; thus $^{max}_a$ remains unchanged. We now show that the maximal offsets of the other anchors do not increase as a result of the added edge. Consider an anchor *d* where *a 2 A(d)*, i.e. *d* follows *a* in the ordering. For the sake of contradiction, assume that$^{max}_d$ has increased. This implies a path *(d; v)* exists from *d* to *v* containing the unbounded edge weight *(d)*. Since *a 2 A(d)*, it follows that *a* is not a prime anchor of *v*. This contradicts the fact the *v* has been assigned to cluster $_i$ = *fag*. By contradiction, $^{max}_d$ cannot not increase.

Likewise, consider an anchor *c* where *c 2 A(a)*, i.e. *c* precedes *a* in the ordering. For the sake of contradiction, assume that $^{max}_{c}$ has increased. This implies that a vertex w exists such that there is a path *(c; w)* from c to w with exactly one unbounded weight *(c)*. Furthermore, *(c; w)* must contain the newly added edge e_{vb}. Since paths from c to a and from a to v exist, it follows that a path *(c; a; v; w)* exists. This path has at least two unbounded weights *(c)* and *(a)*. This implies that c is not a prime anchor of w, and by contradiction, $^{max}_{c}$cannot increase.

From the above arguments all maximal offsets remain unchanged, i.e. $COST^{G'}_{offset} = COST^{G}_{offset}$. Since *jP A(v)j* = 1 for all v except for the source vertex, making the graph taut reduces the synchronization

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,” *IEEE Transactions on CAD/ICAS*, vol. 10, no. 1, pp. 74–84, Jan. 1991.

[3] G. Saucier, C. Duff, and F. Poirot, “State assignment of controllers for optimal area implementation,” in *Proceedings of the European Design Automation Conference*, (Glasgow, Scotland), pp. 547–551, Mar. 1990.

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

[5] 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.

[6] R. Camposano and W. Rosenstiel, “Synthesizing circuits from behavioral descriptions,” *IEEE Transactions on CAD/ICAS*, vol. 8, no. 2, pp. 171–180, Feb. 1989.

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

[8] D. C. Ku and G. DeMicheli, *High Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[9] D. C. Ku and G. DeMicheli, “Constrained resource sharing and conflict resolution in Hebe,” *INTEGRATION, the VLSI Journal*, vol. 12, no. 2, pp. 131–165, Dec. 1991.

[10] D. C. Ku and G. DeMicheli, “Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits,” *IEEE Transactions on CAD/ICAS*, May 1992.