Multi-Token Resource Sharing for Pipelined Asynchronous Systems

John Hansen and Montek Singh
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
{jbhansen,montek}@cs.unc.edu

***Abstract*—This paper introduces the first exact method for optimal resource sharing in a pipelined system in order to minimize area. Given as input a dependence graph and a throughput requirement, our approach searches through the space of legal resource allocations, performing both scheduling and optimal buffer insertion, in order to produce the minimum area implementation. Furthermore, we do not arbitrarily limit the number of concurrent threads or data tokens; instead, we explore the full space of legal token counts, effectively allowing the depth of pipelining to be determined by our algorithm, while concurrently minimizing area and meeting performance constraints. Our approach has been automated, and compared with an existing single-token scheduling approach. Experiments using a set of benchmarks indicate that our multi-token approach has significant advantages: (i) it can find schedules that deliver higher throughput than the single-token approach; and (ii) for the same throughput, the multi-token approach obtains solutions that consumed 33-61% less area.**

I. Introduction

This paper introduces a new approach for performing resource sharing in pipelined asynchronous systems. Since the pipelined paradigm is mainly meant for designing high-performance systems, conserving area is secondary to achieving high performance. Therefore, existing approaches to designing pipelined systems typically do not handle resource sharing (*e.g.*, [1]). On the other hand, high-level synthesis approaches that handle allocation, scheduling and binding of shared resources generally assume a control-driven architecture (*e.g.*, [2]). These latter approaches do not lend themselves easily to fast pipelined multi-token operation. This work attempts to bridge the gap between pipelined data-flow systems and control-driven shared-resource systems.

The key contribution in this paper is a novel *multi-token* scheduling approach that targets throughput rather than latency. Our proposed approach specifically targets resource sharing in a pipelined context, one where multiple instances of the problem are being computed at once. This domain is distinct from that of [3] and others, which focus on *single-token* scheduling in order to minimize the overall latency.

More specifically, this paper introduces the first exact method for minimizing area by optimally sharing resources in a pipelined multi-token manner while meeting a performance (i.e., cycle time) constraint. This problem has long been a challenge even in synchronous design, and is much harder than single-token (i.e., unpipelined) scheduling because of two reasons: (i) one cycle of the cyclic schedule can mix-and-match operations from multiple successive tokens, and (ii) there is an additional dimension to the search space because of the need for pipeline buffer insertion in order to meet correctness and performance constraints. Thus, in multi-token scheduling, the search space includes all possible resource allocations, all schedules that satisfy data dependences, all token counts, and all possible buffer insertions.

978-3-9810801-8-6/DATE12/©2012 EDAA

While there are several flavors of the resource sharing problem, this paper focuses on minimizing area subject to performance constraints. The rationale is that typically a performance bound is specified by the target application, and the designer's objective is to reduce area in order to improve yield, lower die costs, and reduce leakage power.

A key feature of our multi-token scheduling approach is that, unlike some existing approaches [4], it does not repeatedly perform "unfolding" of the data-flow graph, followed by scheduling, and finally compaction in order to determine the schedule for multi-token operation. Instead, it directly determines a compact, multi-token schedule in an optimal fashion. Central to our approach is a new graphical model—*scheduled buffered dependence graphs* (SBDGs)—which allows the entire joint search space of resource schedules and buffer insertions to be efficiently enumerated.

Our approach has been automated, and experimental results on a set of benchmarks are promising. Multiple different test cases were considered, each was synthesized using several different throughput constraints. In each case, our approach performed resource scheduling to meet the throughput constraints and reported the area of the implementation. As expected, as throughput constraints were relaxed, the area of the implementation improved. More importantly, however, the multi-token approach obtained far superior results as compared to the single-token approach of [3]. While in 10 out of 20 examples, the single-token approach could not find a schedule that met the throughput requirement, the multi-token approach found a schedule in all cases. Moreover, for examples in which single-token schedules were found, the multi-token solution consumed 33–61% less area.

The remainder of this paper is organized as follows. Section II discusses previous work. Section III gives background on dependence graphs, and then Section IV describes how we extend the model by incorporating buffering and resource schedules. Section V presents our architectural model, and then Section VI introduces our synthesis algorithm. Section VII presents results, and Section VIII gives conclusions.

II. Previous Work

Several techniques have been proposed for performing high-level synthesis of synchronous (discrete-time) and asynchronous (continuous-time) systems; a general survey of techniques is provided by [5]. The majority of proposed techniques are heuristic, such as force-directed scheduling [6] and list scheduling [7]. In the asynchronous realm, synchronous ILP approaches have been adapted in order to approximate optimal single-token schedules, but these approaches may end up being either slow or sub-optimal depending on the discretization of time. Such asynchronous ILP-based approaches have been reported in [8], [9]. Some optimal asynchronous single-token

```
while(true){
   a=read();
   b=((3*b)+a)*0.25;
} //Loop A
```

```
while(true){
   a=read();
   b=3*d;
   c=a+b;
   d=c*0.25;
} //Loop B
```

Fig. 1: Simple code example

scheduling algorithms exist [3], [10]; these are used for comparison in the results section.

All of these approaches, however, allow only one problem instance to be computed at a time, limiting their performance substantially. Other approaches, such as [11] can allow multiple threads of execution, but the designer must specify how many tokens will exist in the implementation. [12] provides a heuristic method for synchronous multi-token scheduling but focuses on practicality (tool run-times) rather than optimality.

In contrast to these approaches, the approach we present in the following sections creates a multi-token schedule, subject to a throughput constraint, that *optimally* minimizes the total resource and buffer area of a pipeline. This approach searches the full space of multi-token schedules and concurrently performs slack-matching to meet a throughput constraint. To the best of our knowledge, this is the first optimal formulation to jointly minimize function unit area and pipeline buffer area while searching the complete multi-token scheduling search space and allowing for a variable token count.

III. Basic Graphical Model

This section reviews *folded dependence graphs* [13] as a convenient graphical model for representing repeated sets of dependent computations. The next section will introduce our extended model (scheduled buffered dependence graphs) for incorporating resource sharing and buffering.

A. Dependence Graphs

Dependence graphs are used to model data dependencies between the individual operations in a specification. An example representation is shown in Figure 2a, corresponding to the specification in Figure 1. Here the graph has been *folded* to show data and control dependence across iterations.

Here, a single node a represents the execution of the operation over all iterations (a_0, a_1, a_2, $\cdots$); the subscripts representing iteration numbers are dropped. A *weight* is associated with each arc to represent the difference in subscripts from the source node to the destination node. Thus, *intra-iteration arcs*, such as the one between operation b and c, will have a weight of 0. *Inter-iteration arcs*, such as the arc from d to b, have a non-zero weight, in this case 1. To ensure liveness of the specification, the weight on each cycle in the graph must sum up to one or greater, otherwise a deadlock is implied.

In addition to a weight on each arc, let there be a *delay* associated with each arc in the folded dependence graph (a "fixed-delay model"). The delay associated with an arc from node x to node y represents the length of time that must elapse from the instant that the x operation completes to the instant the y operation completes. No delays are associated with the nodes; instead all delays are represented on arcs. This delay is distinct from the weight associated with an arc.

B. Cycle Time Analysis

A dependence graph can be analyzed to determine its maximum throughput (or, equivalently, its minimum cycle time), using the classical approach of *maximum cycle mean* computation [14]. The cycle mean for cycle c in graph G is defined as follows:

$$\text{Mean}(c) = \frac{\sum_{e \in c} \text{delay}(e)}{\sum_{e \in c} \text{weight}(e)}$$

where e is an edge in the cycle c. The cycle time is given by the maximum of the cycle means for all cycles in the graph:

$$\text{Cycle Time}(G) = \max_{c \in G} (\text{Mean}(c))$$

Intuitively, the cycle time of an individual cycle is the total cycle delay divided by the number of tokens on that cycle. The cycle time of the full graph is the longest cycle time of any cycle in the graph.

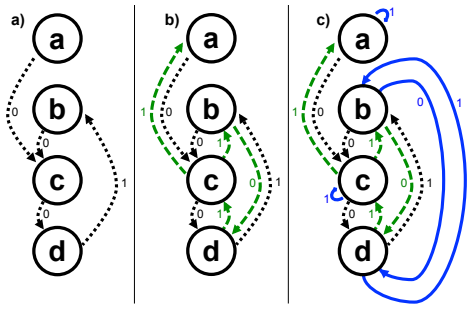

Fig. 2: a) Data, b) buffering, and c) resource arcs

IV. Extended Graphical Model

The basic model of Section III captures data-dependencies (read-after-write or RAW constraints). In this section, we introduce an extension called *scheduled buffered dependence graphs* (SBDGs), which incorporate two additional types of constraints: (i) *write-after-read (WAR) constraints,* which prevent data from being overwritten until it has been consumed; and (ii) *resource scheduling constraints,* which model resource sharing. Both of these types of constraints are modeled by adding additional arcs to the dependency graph.

A key contribution of this section is illustrating how buffering (*i.e.*, storage) requirements can be directly inferred from the graph model. In addition, it also describes how the delays of those buffers are modeled appropriately in the graph.

A. Modeling Write-After-Read (WAR) Constraints

WAR constraints are necessary to ensure that a storage location is written only after its previous value has been read. Because there is contention for storage, the extent of allowable concurrency in execution scenarios becomes restricted. In order to model this resctriction, we add WAR arcs to the dependency graph. For each data dependence arc between a pair of nodes, we add a WAR arc between the same nodes in the reverse direction, as shown in Figure 2b. Here, the dotted black arcs represent data dependence, and the dashed green arcs represent WAR constraints. In the remainder of this paper, the terms *WAR arc*, *reverse arc*, and *acknowledgment arc* are used interchangeably.

Theorem 1. *Given a data channel between two nodes a and b, with m the weight of the forward arc, and n the weight of the reverse arc (as shown in Figure 3a), the number of buffers required for correct operation is $m + n$.*

Proof: Assume an instant in time such that b_m has occurred (and therefore b_k for all $k < m$ have also already

occurred), but b_{m+1} has *not* occurred. Then, by virtue of data dependence, a_0 must have occurred. Also, because of the reverse arc from b to a, a_{m+n+1} cannot have occurred yet since b_{m+1} has not occurred. At this point in time, the events $a_1 \cdots a_{m+n}$ may occur before any further events on b, and therefore all of these results must be stored and preserved as the future events $b_{m+1} \cdots b_{2m+n}$ will need them. As a result, up to $m+n$ buffers may be required to queue up the values $a_1 \cdots a_{m+n}$. Figure 3b graphically illustrates the proof. ■

B. Modeling Buffer Delays

When buffers are present on a data channel, their delays must be correctly included during timing analysis. In particular, the forward latency through the buffers will add to the total delay from the source node to the destination node. In addition, each buffer also has a reverse latency: the time from the instant the buffer is emptied to the instant its predecessor is enabled to produce the next value.

The proposed approach to modeling the delays due to buffering is illustrated by Figure 3c. In the figure, the node a is replaced by $m+n$ new nodes, numbered $a^0 \cdots a^{m+n-1}$, each new nodes representing a distinct buffer.

For correctly modeling the timing behavior, we set the delays along the arcs are as follows:

- the weight of the forward arcs $a^0 \rightarrow a^1 \cdots a^{m+n-2} \rightarrow a^{m+n-1}$ is equal to the buffer forward latency, $buff_f$
- the weight of the forward arc $a_{m+n-1} \rightarrow b$ is equal to d, which is the latency associated with operation b
- the weight of the reverse arc $b \rightarrow a^{m+n-1}$ is equal to r, which is the reverse latency associated with b
- the weight of the reverse arcs $a^{m+n-1} \rightarrow a^{m+n-2} \cdots a^1 \rightarrow a^0$ is equal to the buffer reverse latency $buff_r$

The graph of Figure 3c is compacted into the simplified representation of Figure 3d by setting the delays appropriately:

- set the forward arc delay equal to $d + (m+n-1) * buff_f$
- set the reverse arc delay equal to $r + (m+n-1) * buff_r$
- add a self loop on a with weight $buff_f + buff_r$
- add a self loop on b with weight $d + r$

With these delay assignments, the computation of the maximum cycle mean for any graph that contains the sub-graph of Figure 3c will be correctly computed by including instead the sub-graph of Figure 3d.

C. Modeling Resource Sharing

Scheduling of shared resources is modeled by adding new arcs to the dependence graph, called *resource arcs*, as shown in Figure 2c. In particular, one cycle of resource arcs is created for each available resource. The delay associated with each of these resource arcs is the latency of that resource.

The sum of the weights of the arcs in each such cycle is equal to 1 as the proposed multi-token approach only considers cyclic schedules with a unit stride. As an example, if a certain function unit executes the sequence of operations $a_i, b_j, c_k \cdots$ in one iteration, then the same function unit must execute the same sequence of operations in the next iteration, $a_{i+1}, b_{j+1}, c_{k+1} \cdots$. Therefore, the weight of each resource cycle will be equal to 1.

Property 1. *(Unit Stride Property) The cycle weight for a resource cycle with unit stride must be equal to* 1.

In practice, the delays on each arc consist of overheads beyond the operation latency. The controller delay associated with a function unit, multiplexing delay, and the forward delay of a buffer stage are also incorporated.

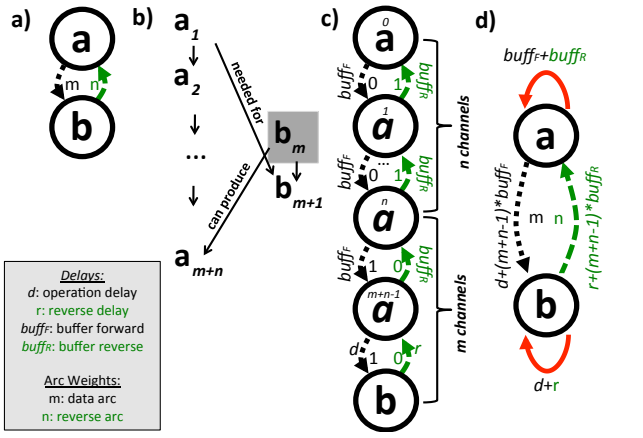

Fig. 3: Modeling buffering requirements

D. Converting the Graph to Architecture-Ready Form

Once a graph has been scheduled and slack-matched using the model above, one additional step is performed to prepare the graph for conversion into hardware. The method for mapping a graph to a hardware implementation (to be described in Section V) is straightforward, provided there are no negative weights on any arcs in the graph. Therefore, we must remove these negative arcs, and do so in such a way that the schedule, circuit performance, and buffer requirements are preserved.

Here, we prove that any graph with negative arc weights can be converted to an equivalent non-negative graph which we call the *architecture-ready* form by following a series of transformations under the constraints above.

To begin with, let us define the method of *re-weighting*. In this method, we select a node that has all positive incoming arcs. Let the weight of the smallest positive incoming arc be α. The re-weighting step reduces the weights of all incoming arcs by α, and increase the weights of all outgoing arcs by α. Since we are adding the same value to every outgoing arc that we are subtracting from the incoming arc, this method preserves the total weight on any cycle going through the node. One key aspect of re-weighting is that the weight of any non-negative arc can never become negative through re-weighting.

Theorem 2. *For a deadlock-free, strongly-connected graph, there must be at least one node in the graph that has positive weights on all its incoming arcs.*

Proof: The proof is by contradiction. If there was no node in the graph with positive weights on all its incoming arcs, it would be possible to trace a cycle in the graph with a total cycle weight of 0 or less, implying a deadlock. ■

Corollary 3. *For a deadlock-free, strongly-connected graph, we can perform an infinite number of re-weightings.*

Proof: This corollary is trivially true, as there will always exist a node that can be re-weighted (Theorem 2). ■

Lemma 4. *For a deadlock-free, strongly-connected graph, all nodes in a graph will have been re-weighted after a finite number of re-weightings.*

Proof: We begin by considering a node that has not been re-weighted, X. Because our graph is strongly-connected,

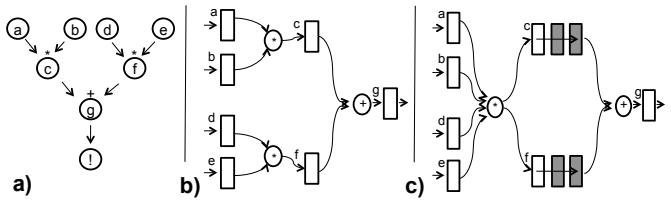

Fig. 4: a) Sample DFG, b) unshared architecture, and c) shared architecture with buffering

there is a path from X to every other node in the graph. For a path from X to any other node in the graph Y, we can sum up all the weights on this path to give a finite value. Because Y can only be re-weighted if its incoming arc is positive, this sum corresponds to the maximum number of times that Y could be re-weighted before X must be re-weighted.

Since there are a finite number of nodes in the graph, and since each can only be re-weighted a finite number of times before X is re-weighted, there are a finite number of re-weightings that can occur before X must be re-weighted. Since the number of legal re-weightings is infinite according to Theorem 2, X must eventually be re-weighted. By the same reasoning, all nodes in the graph must be re-weighted within a finite number of re-weightings. ■

Theorem 5. *Any deadlock-free, strongly-connected graph that contains arc(s) with negative edge weight(s) can be converted into an equivalent graph with no negative edge-weight arcs.*

Proof: According to Lemma 4, all nodes in the graph are guaranteed to have been re-weighted after performing a finite number of re-weightings. Therefore, we can perform re-weighting in any legal order until each node has been re-weighted at least once, and therefore each node's incoming arcs will have become non-negative. ■

Because the total weight on a cycle remains unchanged, each channel will have the same number of buffer stages after this conversion, and the cycle time of the graph will be unchanged. Additionally, since no arcs were added, removed, or redirected, each resource's schedule remains the same.

V. Architectural Model

This section introduces a data-flow, shared-resource architecture that implements the extended graphical model of Section IV. We will begin with a general overview of the datapath, then discuss the components used in the proposed architecture: buffers, forking data latches, and resources (function units).

A. Overview

A diagram illustrating the basic architecture is illustrated in Figure 4. An example DFG is shown in Figure 4a, one which performs a dot-product of a pair of two-element vectors: $< a, c > \cdot < b, d >= a \cdot b + c \cdot d$. Figure 4b shows a basic architecture for this DFG without resource sharing. Finally, Figure 4c shows our architecture with a shared multiplier and additional buffers on two data channels. This example features the three key components in the proposed architecture: (i) storage locations for variables ($a - g$) that come directly from the environment or function units, (ii) extra data buffers (in gray), and (iii) resources (shared or dedicated).

To generate an architecture from a given dependence graph, we begin by replacing each node in the graph with a data latch. This step ensures that we have at least one storage location

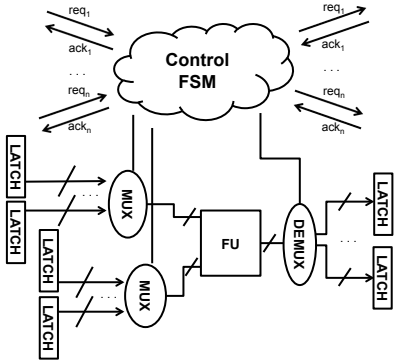

Fig. 5: Shared resource implementation

for each variable in the original specification. Then, between nodes with data-dependencies, we build a channel with zero or more additional buffers, necessary for slack matching and data synchronization. Multiple channels may be generated from the same data latch source, since a variable may be needed for different computations, but each channel from the same source variable may contain a different number of buffers. At the end of a channel, the final buffer feeds into a function unit, which will, in turn, feed into a new data latch.

B. Components

1) Buffers: The purpose of a buffer in this architecture is (i) hold older data while new data is being computed, preventing old data from being overwritten, and (ii) to improve performance via slack-matching, as described in Section III-B. A series of buffers may be placed on a channel between a data latch and the function unit it feeds into. The total count of all buffers on a channel is described in Section IV.

The buffer stage consists of a basic storage element manipulated by a simple controller. The behavior of a single buffer stage repeats as follows: (i) wait for an incoming request, (ii) latch data, acknowledge, send an outgoing request, (iii) wait for acknowledgement. While we have selected this specific pipeline style, other pipeline styles can certainly be used.

The output of a function unit goes into a special buffer: a forking data latch. This latch may forward data down multiple paths, unlike a standard buffer, that sends data down only one path. Based on the architecture-ready graph produced by our algorithm, a buffer stage will either be initialized as full (a 1 on a forward data arc) or empty (a 1 on the reverse data arc).

2) Function Unit and Control: Function units may be dedicated or shared. If dedicated, no complex control is necessary. If a function unit is shared, there will be multiple inputs to be multiplexed and outputs that need to be routed. The diagram for a function unit is shown in Figure 5.

All of the handshake channels feed into a state machine that controls the schedule of operations on the function unit. This state machine is not global, but is instead a local controller, one per function unit. The state machines repeats the following steps indefinitely: (i) consult schedule to set input and output multiplexers, (ii) forward incoming request to data latch with appropriate matched delay, (iii) forward acknowledgement from data latch to inputs.

VI. Problem Formulation

In this section, we describe an optimal approach for synthesis. We first give a top-level overview of the approach,

then describe a branch and bound strategy for scheduling, allocation, and binding of resources. Next, we describe an ILP-based approach for verifying the throughput constraint by performing slack-matching.

A. Overview of Approach

The multi-token scheduling problem can be broken down into two specific sub-problems: (i) scheduling and allocating function units, and (ii) verifying that the schedule meets the throughput constraint after optimal buffering. Therefore, the proposed solution has been broken down into two phases: a branch-and-bound scheduling phase, and an ILP-based technique for optimal buffer insertion and ensuring satisfaction of the throughput constraint.

At the top level, the proposed approach steps through the scheduling process for each function unit, allocating additional units as necessary. This branch-and-bound algorithm fully schedules each resource one by one. As each function unit is scheduled, an ILP instance is run to ensure it meets the throughput constraint given the opportunity for buffer insertion. Once a legal schedule is found with all operations scheduled that meets the cycle constraint, this schedule compared with the best solution so far, and searching continues until no better schedules can be found.

B. Scheduling, Binding, and Allocation: Branch and Bound

The branch and bound portion of the proposed approach begins with the original graph with all the data-dependencies in place. Next, a reverse arc is added between each data-dependent node in order to produce a complete channel.

The basic recursive scheduling algorithm is as follows:

1) Generate a list of unscheduled items, sorted lexicographically. Select the first unscheduled item from the list.
2) Create a list of resources on which this item could execute, subject to an area constraint.
3) Explore scheduling the operation on each one of these resources in a depth-first fashion.
4) After an operation has been scheduled, create a list of unscheduled nodes remaining that could execute on the same resource. Include the first node on this resource's schedule in order to complete the resource's cycle.
5) Explore scheduling each child operation on this resource in a depth-first fashion, adding a resource arc from the previously scheduled node to the current node.
6) If the resource cycle has been closed, compute the buffering needed to achieve the throughput constraint by running ILP described in VI-C. If buffering cannot meet the throughput constraint, or if the total area exceeds the best area, we stop exploring this partial schedule.
7) If there are unscheduled nodes remaining, return to Step 1. Otherwise, a new best area solution has been found. This value is recorded and scheduling continues.

Beyond the basic bounding performed in Step 6, we can improve run-time by adding a few additional optimizations. The first optimization is to estimate the minimum area for unscheduled operations by using utilization analysis, and use this value to help prune. This estimation includes preserving the minimum amount of buffers needed for a partially scheduled implementation, as we know that this amount cannot decrease as we continue to schedule more items. Additional optimizations include ordering the search space to consider more promising results first, and employing backtracking when generating partial schedules to determine which scheduling steps introduced additional buffers.

C. Buffering and Cycle Time Constraints: ILP

After the step of scheduling each specific resource, the result must be confirmed to meet the performance constraint specified by the designer. In order to meet the throughput constraint, additional buffers may be inserted automatically by the algorithm. Because the designer's goal is area minimization, we aim to minimize the count of these additional buffers.

The steps of buffer insertion and confirming that a schedule meets the throughput constraint are performed in tandem using an ILP approach. In this formulation, we will insert the performance constraints as linear constraints in the ILP, and allow the solver to vary the number of buffers. The sum of buffers in the implementation will be the minimization target.

The following notation is used below:

- $\mathcal{F}$: the set of *forward* arcs (data dependencies)
- $\mathcal{R}$: the set of *reverse* arcs (WAR constraints)
- $\mathcal{S}$: the set of resource *scheduling* arcs
- $\mathcal{C}$: the set of *cycles* in the dependence graph
- $\mathcal{CS}$: the set of cycles consisting solely of scheduling arcs
- T: the *target cycle time* specified by the designer
- $ch_{\#}$: the number of channels in the graph
- $n_{\#}$: the number of nodes in the graph

The set of variables in the ILP consists of the weight(e) for each $e \in \mathcal{R} \cup \mathcal{S}$ (reverse and scheduling arcs).

1) Cost Function: The cost function to minimize is simply the total number of buffers required. As described in IV-A, the total number of buffers required on a channel is given by the sum of the weights on the forward and reverse arcs that constitute that channel. However, if a node has more than one output channels (*i.e.*, it represents a fork), then the first latch is common to all channels; any additional buffers added are disjoint. Therefore the cost function is:

$$\sum_{e \in (\mathcal{F} \cup \mathcal{R})} \text{weight}(e) - ch_{\#} + n_{\#}$$

2) Constraints: For each cycle in the graph, we enumerate three sets of constraints to ensure that (i) the liveness property is met; (ii) only schedules with stride of 1 are allowed; and (iii) the performance target is met.

Liveness constraint: The sum of the weights on a cycle must be greater than or equal to 1:

$$\sum_{e \in c} \text{weight}(e) \quad \geq \quad 1 \qquad \text{for all } c \in \mathcal{C}$$

Unity stride of schedules: According to Property 1, the cycle weight for a cycle consisting solely of resource scheduling arcs must be equal to 1:

$$\sum_{e \in c} \text{weight}(e) \quad = \quad 1 \qquad \text{for all } c \in \mathcal{CS}$$

Performance constraint: As discussed in Section III-B, the cycle mean for each cycle in the graph must be less than or equal to the target cycle time specified, T, which can be rewritten as the linear constraint:

$$\sum_{e \in c} \text{delay}(e) \quad \leq \quad T \cdot \sum_{e \in c} \text{weight}(e) \qquad \text{for all } c \in \mathcal{C}$$

Note that the the expression for delay(e) will, in general, include delay terms for forward and reverse buffer latencies,

TABLE I: Function unit parameters

Function Unit	Area (unit)	Latency (unit)
Add	8	8
Subtract	8	8
Multiply	48	9
Shift/Logical	8	8
Buffer	2	1 / 1

which is in turn dependent on the number of buffers required on the corresponding data channel. As discussed in Section IV-B, the number of buffers is determined by the sum of the weights of the forward arc (known constant) and the weight of the reverse arc (a variable in ILP). Therefore, the cycle mean constraints are linear in the variables.

VII. Results

Setup. In our experiments we used six different benchmark DFGs for analysis, described in [3]. For each test case, we set constraints for cycle time and optimized for minimum area, and compared out results to the single-token approach of [3]. We provided the same library of functional units to all benchmarks; their parameters are shown in Table I.

Our approach was implemented in Java on a Macbook Pro with a 2.8 GHz Intel Core 2 Duo processor and 4GB of RAM on JVM 1.6. For solving the ILP instances, we used the ILOG CPLEX tool. Runtimes are shown in seconds.

Discussion. Table II shows the experimental results for the optimal approach. The first two columns list the benchmark and throughput constraint respectively. For the single-token solver in [3], this throughput constraint was equal to the latency constraint, as only single-token schedules were produced. The next column shows the logic area a single token schedule could produce using the single-token approach (this method ignores buffer area). The next three columns show the results of the multi-token approach, including logic, buffer, and total area. The final column shows the run-time in seconds.

The results clearly show that a multi-token approach is superior to a single-token approach in terms of function unit area. In all test cases, the logic area was less than or equal to that of the single-token solver. In fact, in most cases, the *total* area (including buffering) was lower than the function unit area of the single-token solver. Further, there are several instances where the single-token approach *cannot* meet the throughput constraint, even with infinite resources.

Now let us consider the effect of the throughput constraint on buffer area: when the throughput constraints become very tight we begin to see an increase in buffer area because more pipelining is needed. For example, consider the first two test cases of *COS,* where the buffer area is 52 under a tight throughput constraint of 16, but when the throughput constraint is relaxed to 32, the buffer area reduces to 48.

Finally, the runtime of this approach is illustrated in the last column. For the single-token approach, the run-time was under 5 seconds in each test case. In the multi-token test cases, the runtime was under 10 seconds in all but four test cases. Three of those successfully completed in under an hour, while one test case did not complete within 8 hours and was manually terminated. This one case demonstrates the high complexity of the search space, and motivates the need for faster heuristic approaches, which is part of our ongoing work.

TABLE II: Synthesis results and tool runtimes

Benchmark	Cycle Time Constraint	Area (unit) 1-● Logic	Multi-● Logic	Multi-● Buffers	Multi-● Total	Tool Runtime (s)
ODE	9	-	272	30	302	0.2
ODE	34	160	112	18	130	0.2
ODE	50	112	64	18	82	0.2
DP8	9	-	440	48	488	0.3
DP8	27	-	168	32	200	0.4
DP8	35	416	160	32	192	0.4
DP8	50	208	112	32	144	0.7
DP8	90	104	56	32	88	0.7
COS	16	-	800	52	852	3.8
COS	32	-	304	48	352	355
COS	75	208	104	48	152	1908
7TH	9	-	832	88	920	0.7
7TH	16	-	776	58	834	1.1
7TH	45	-	168	58	226	51
7TH	90	168	112	58	170	493
ELP	9	-	592	202	794	2.5
ELP	115	168	-	-	-	>8hr
TEA	32	-	40	36	76	7.8
TEA	40	48	32	36	68	4.1
TEA	43	32	32	34	66	5.9

VIII. Conclusion

In this paper we described an optimal method for generating multi-token schedules for performing resource sharing in a pipelined system. We illustrated how these schedules could be modeled graphically, described an architecture to implement these schedules, and developed an exact algorithm to minimize overall logic and buffer area while meeting a throughput constraint. In future work, we aim to pursue a heuristic approach that can handle larger examples.

References

[1] M. Budiu, "Spatial computation," Ph.D. dissertation, Carnegie Mellon University, Computer Science Department, December 2003.

[2] S. F. Nielsen, J. Sparsø, and J. Madsen, "Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation." *Digital Systems Design, Euromicro Symposium on*, pp. 298–305, 2004.

[3] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems," in *Proc. Int. Symp. on Asynchronous Circuits and Systems (ASYNC)*, 2010.

[4] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ser. DAC '90. ACM, 1990, pp. 444–449.

[5] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[6] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *DAC '87: Proceedings of the 24th ACM/IEEE Design Automation Conference*. ACM, 1987, pp. 195–202.

[7] A. M. Sllame and V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis," in *DSD '02: Proceedings of the Euromicro Symposium on Digital Systems Design*, 2002, p. 316.

[8] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the haste/tide design flow," in *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2009, pp. 185–194.

[9] H. Saito, N. Hamada, N. Jindapetch, T. Yoneda, C. Myers, and T. Nanya, "Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E90-A, no. 12, 2007.

[10] J. Hansen and M. Singh, "An energy and power-aware approach to high-level synthesis of of asynchronous systems," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 2010.

[11] S. Tugsinavisut, R. Su, and P. A. Beerel, "High-level synthesis for highly concurrent hardware systems," *Application of Concurrency to System Design, International Conference on*, pp. 79–90, 2006.

[12] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe, "Realistic performance-constrained pipelining in high-level synthesis," in *Proc. Design, Automation and Test in Europe (DATE)*, 2011, pp. 1382–1387.

[13] T. E. Williams, "Self-timed rings and their application to division," Ph.D. dissertation, Stanford, CA, USA, 1991, uMI Order No. GAX92-05744.

[14] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system performance analysis," *IEEE Transactions on CAD*, vol. 17, pp. 889–899, 1997.