2012 IEEE 18th International Symposium on Asynchronous Circuits and Systems

A Fast Hierarchical Approach to Resource Sharing in 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 proposes a novel hierarchical approach for scheduling shared resources in asynchronous pipelined systems. While there have been recent approaches to asynchronous resource scheduling, the problem is especially difficult for multi-token systems, i.e., systems where computation on multiple problem instances is overlapped and pipelined, so resources are shared amongst operations across different problem instances. An approach recently proposed for multi-token scheduling [1] can optimally solve this synthesis problem for modest-sized specifications, but an exact solution for larger benchmarks can be intractable.**

To overcome this challenge, we introduce a novel method that decomposes the problem based on the hierarchy inherent in the specification. Individual blocks are isolated and scheduled, and an abstracted model of their behavior is passed to higher levels of the hierarchy. As a result, our approach obtained drastically reduced runtimes. While the resulting solution is not globally optimal, the method in practice produces high-quality solutions. Our approach has been automated and validated using a variety of benchmarks to illustrate its effectiveness in minimizing area while meeting a target throughput constraint.

I. Introduction

This paper introduces a hierarchical approach to resource sharing in pipelined asynchronous systems. The task of resource sharing involves allocation, scheduling and binding of resources, and is the key problem targeted in high-level synthesis. Traditionally, asynchronous design tools have typically either focused on high-performance pipelined design without resource sharing (e.g., [2]), or targeted a control-driven shared-resource architecture that does not lend itself easily to fast pipelined multi-token operation (e.g., [3]). An approach was recently introduced to bridge this gap by providing for multi-token resource sharing [1]. Figure 1 illustrates the concept of pipelining with and without resource sharing. This paper builds upon [1] to introduce a scheduling approach that decomposes a large system based on its hierarchy. As a result, our approach can handle significantly larger specifications in relatively short runtimes.

The approach of [1] was the *first exact method*, whether synchronous or asynchronous, for optimal resource sharing in a pipelined multi-token system. Thus the schedule is cyclic, repeating for each successive token. This problem has long been a challenge even in synchronous design, and is much harder than single-token (i.e., unpipelined) scheduling. In particular, the search space is much bigger for the multi-token problem 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 buffer insertion. Buffers need to be added for multi-token scheduling for two reasons: to avoid write-after-read hazards due to pipelined operation (correctness), and to eliminate slack mismatch (performance).

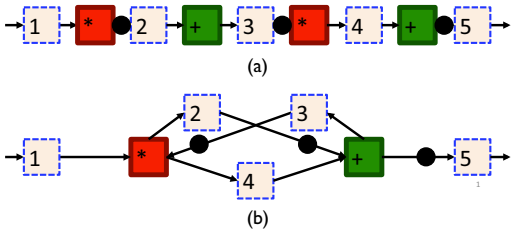


Fig. 1: a) Pipeline without resource sharing, b) Pipeline with shared resources.

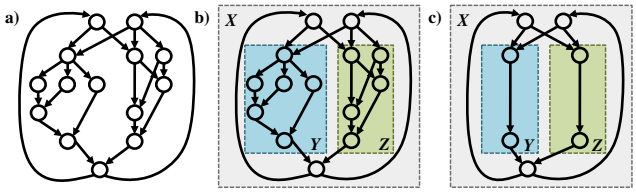


Fig. 2: a) Original graph specification, b) block-partitioned graph, and c) block-partitioned graph after blocks Y and Z are scheduled and simplified

In contrast, when there is only one token flowing through the system, the scheduling problem is typically decoupled from the determination of storage requirements. 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.

Since the exact approach of [1] can become too complex for large specifications, this paper proposes a novel hierarchical approach for handling large real-world examples. The key idea is that, instead of scheduling the full graph specification (Figure 2a) at once, the graph is decomposed into multiple sections, or *blocks* (Figure 2b) based on its hierarchy. Each smaller block is then scheduled individually with its own disjoint set of resources, thereby minimizing its own local area while meeting the global throughput constraint. The scheduled block is then replaced by a simplified model for use within the next higher-level block in the hierarchy that is yet to be scheduled (Figure 2c). This procedure repeats for each nested layer of blocks until the full graph is scheduled.

In comparison to the optimal algorithm of [1], our hierarchical algorithm may not produce a globally optimal solution, but it requires significantly less runtime. In particular, because resources are no longer shared across blocks, this method trades off strict global optimality for higher algorithmic efficiency, dependent on the partitioning granularity. While the final solution may not be the exact optimal solution in terms of area, it is still guaranteed to meet the throughput constraint.

1522-8681/12 $26.00 © 2012 IEEE
DOI 10.1109/ASYNC.2012.30

IEEE computer society

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

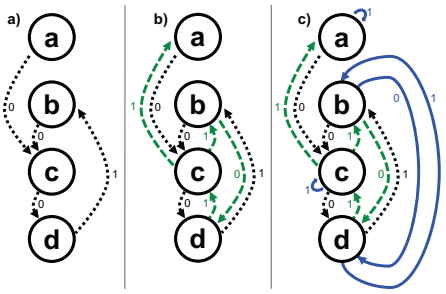


Fig. 3: Adding a) data, b) buffering, and c) resource arcs to the graph

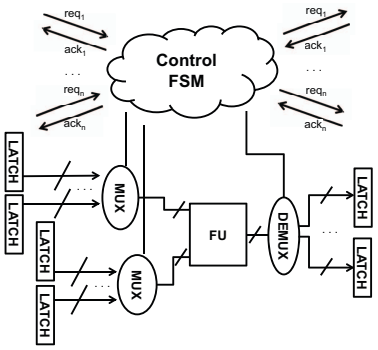


Fig. 4: Shared resource implementation

In addition to the runtime benefits, our proposed method also improves upon [1] by handling a more general class of specifications that can include conditionals (if-then-else blocks) as well as certain types of iterative blocks (fixed-count loops).

The remainder of this paper is organized as follows. Section II gives the necessary background on the approach that it builds upon [1], and reviews other previous work. Next, Section III presents they key idea of decomposition of the specification into blocks, and explains how a block is abstracted into a simpler model. Section IV illustrates how blocks are combined, using parallel and serial composition as examples, then describes our resource sharing algorithm. Finally, Section V presents results, and Section VI gives conclusions.

II. Background and Previous Work

This section provides background on the multi-token resource scheduling approach of [1], upon which the work of this paper is based. Other relevant previous work on resource sharing in asynchronous systems is also briefly reviewed.

A. Background: Multi-Token Resource Scheduling

The approach introduced in [1] is the first exact method for minimizing the area cost of a pipelined system through optimal resource sharing for a given throughput target.

1) Graphical Model: Scheduled Buffered Dependency Graphs: Central to the approach is a powerful graphical model introduced in [1], called a *scheduled buffered dependency graph* (SBDG). The model is highly expressive, allowing three types of requirements to be directly modeled: data dependencies, buffering requirements, and resource scheduling constraints. Moreover, the graph also allows the following metrics to be readily computed: cycle time, area, and number of tokens (i.e., amount of pipelining).

Figure 3 shows an example. Each operation is represented by a node (a–d). Data arcs between nodes model *read-after-write* (RAW) or data dependence (dotted arcs). Each arc has a weight, which is difference in their respective sequence or token numbers. For example, the weight 1 on the arc from d to b means the operation b_{k+1} is dependent on d_k, i.e., the result d from one computation is needed for computing b in the next computation. The other three data arcs, with weights 0, represent intra-token dependences (i.e., within the same computation). In Figure 3b, a new type of arc, called a reverse arc or buffering arc, is added for each data arc, but in the reverse direction. These reverse arcs (dashed green lines) model *write-after-read* (WAR) dependence. For example, the reverse arc from d to c with a weight of 1 indicates that operation d_k must complete before c_{K+1} can occur. Finally, in Figure 3c, resource scheduling arcs are added (solid blue lines), which encode the schedule for each resource. For example, operations b and d share a resource, executing alternately on it in the sequence $b_k, d_k, b_{k+1}, d_{k+1}, \cdots$, while operations a and c execute on their own respective dedicated resources.

2) Inferring Buffering Requirements: A key result proved in [1] (Theorem 1) is the following theorem for calculating the number of buffers required on a data channel: *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, the number of buffers required on that channel for correct operation is $m + n$.*

3) Legality Constraints: There are several restrictions that must be met by a legal graph.

Liveness Constraint: The sum of the weights of edges on any cycle must be greater than or equal to 1.

Unity Stride of Schedules: The method seeks to find only those schedules that repeat with a stride of one, i.e., the same schedule that produces a_k must repeat after one complete cycle to produce a_{k+1}. Hence, the cycle weight for a cycle consisting solely of resource scheduling arcs must be equal to 1.

Performance Constraint: The cycle time of the graph is given by its *maximum cycle mean* [4]. If an upper bound is specified on the cycle time, T, then the cycle mean of any cycle, c, in the graph must be no more than T:

$$\text{Mean}(c) = \frac{\sum_{e \in c} \text{delay}(e)}{\sum_{e \in c} \text{weight}(e)} \quad \leq \quad T \quad \text{for all cycles } c$$

where e is an edge in the cycle, and has a corresponding weight and delay. Note that the expression for delay(e) will, in general, include delay terms for forward and reverse buffer latencies, which is in turn dependent on the number of buffers required on the corresponding data channel.

4) The Scheduling Problem: The particular problem solved in [1] is formally stated as follows: Given a dependency graph (G), and a cycle time bound (T), compute a resource schedule that meets the cycle time bound and minimizes area (resources + buffers) over all possible resource allocations, over all possible buffer insertions, and over all possible token counts.

5) The Search Space: The search space is the entire space of legal scheduled buffered dependency graphs that can be formed by starting from a graph which encodes all of the specified data dependence arcs. In particular, a reverse arc is added for each data arc, but the weight on the reverse arc is

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

variable. Any integer is a candidate for a reverse arc weight as long as the legality constraints (liveness, unity stride, and cycle time) are satisfied by the graph. Thus, all possible buffer insertions on each channel are modeled. Similarly, resource arcs are added to the graph such that each resource arc belongs to a resource cycle, and each node belongs to exactly one resource cycle. The placement and weight of each resource arc are variable. Once again, all resource cycles are allowed as long as the legality constraints are satisfied. Hence, the search space allows for all possible resource allocations, all possible buffer insertions, and all possible token counts.

6) The Search Algorithm: The search method is a *two-level* algorithm: a branch-and-bound technique at the top level for enumerating the space of all legal resource cycles, and an integer linear programming (ILP) technique at the second level for searching the space of arc weight assignments. In particular, in the branch-and-bound technique, as the search descends into child nodes, it incrementally adds one more resource cycle to the graph. If this new resource causes the area to increase beyond the best solution found so far, then this path is pruned. Otherwise, this partially scheduled graph undergoes an ILP search (second level) to determine arc weight assignments so that the cycle time constraint is met *and* the area of buffers inserted is minimized.

7) The Architectural Model: Figure 4 shows a fragment of the entire system, which corresponds to one shared resource (FU). There are input multiplexers that select the correct set of operands to send to the resource. Similarly, a demultiplexer routes the output of the resource to the appropriate destination latch. There is a controller for each resource that generates the appropriate select signals for the multiplexers and demultiplexers. The controller has request-acknowledge handshakes corresponding to each input and output channel. The controller logic directly encodes the schedule for the associated resource.

B. Previous Work

Several well-known techniques for high-level synthesis (HLS) exist in both the synchronous and asynchronous domains (see [5] for a survey).

A common problem in HLS is to minimize area given a performance constraint. Many existing approaches to this problem consider only latency as a constraint, targeting "single-token" or unpipelined schedules [6], [3], [7]. While some of these solutions provide optimal solutions in the asynchronous domain [6], others rely on discrete-time techniques that do not necessarily provide an optimal solution in the continuous-time domain [3], [7].

Some approaches do target throughput, therefore allowing multi-token operation as a performance constraint [8], [1], [9]. However, these approaches have drawbacks, for example, [8] requires the token count to be specified by the designer rather than allowing it to be determined by the scheduler, and therefore may miss out on optimal solutions. The work presented in [1] provides an optimal solution to our target problem, but its complexity becomes high for large examples. On the other hand, the work presented in [9] presents a heuristic synchronous approach that focuses on practicality and therefore explores only a subset of the full search space. In terms of optimizing buffer area, "slack matching" for higher performance and lower area is considered in detail in [10].

In contrast to the above, this work specifically targets the problem of multi-token scheduling in shared-resource asynchronous pipelines, building on top of an existing optimal formulation [1] to develop a hierarchical approach that trades off strict global optimality for substantial improvement in tool speed. Our method tackles a problem that is significantly more challenging than those tackled by existing formulations; here we are not only scheduling operations from a single iteration, but *multiple* iterations at once. Our solution jointly optimizes for allocation area and buffering, while performing multi-token scheduling with a variable token count. This approach is capable of handling a rich set of inputs, allowing for loops, conditionals, sequential, parallel, and nested constructs.

III. Modeling Blocks

A. Block Definition

The dependence graph is partitioned into a number of "blocks", each of which is a connected subgraph. This block partitioning is typically defined by the structure of the original source specification (*e.g.*, the body of a function, procedure, loop, conditional, etc.). In general, a block may contain not only graph nodes but one or more lower-level blocks nested in it as well.

As an illustration of a block in a graph, refer again to Figure 2b. Here we see two nested blocks, Y and Z inside block X. At the edge of each block there is an interface to external nodes and blocks. This block interface consists of the channels that connect to nodes at the edge of the block. These channels carry data between pairs of dependent nodes that straddle a block's boundary.

Figure 5a illustrates the interface of Block Y from the earlier example in Figure 2b. Only the data arcs (solid black arcs) are shown in this figure. Here we explicitly show the delay associated with each arc (in bold) in addition to the weight (in italics); recall that the delay is the latency of the operation associated with the node, while the weight is a difference in iteration counts between two operations. Two *internal interface nodes* are highlighted in blue and labeled A and C; these are the nodes that have connections outside the block. Their counterparts are *external interface nodes*, highlighted in orange, which lie outside the block and connect to the internal interface nodes.

A specification's inherent hierarchy is used as the primary guide for decomposing the specification into individual blocks. However, the granularity of partitioning can have a profound impact on the quality of results (as shown in Section V). Currently, a simple heuristic is used to split blocks that are too large according to a designer-specified node count. However, more powerful heuristics are currently being developed for block partitioning.

B. Interface-Level Abstraction

Central to our approach is the ability to simplify a block's graphical model for propagating to the next higher level in the hierarchy. In particular, each individual block is scheduled independently from other blocks in the graph using a disjoint set of resources (the full procedure is described in Section IV). Once the block has been scheduled, only an abstracted version of its model is used in its outer containing block to keep algorithmic complexity manageable.

The simplification of the model involves: (i) hiding the internal cycles in the scheduled block and considering only paths between nodes at the interface of the block, (ii) reducing the number of arcs between pairs of interface nodes via a FIFO approximation, and (iii) reducing the number of interface

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

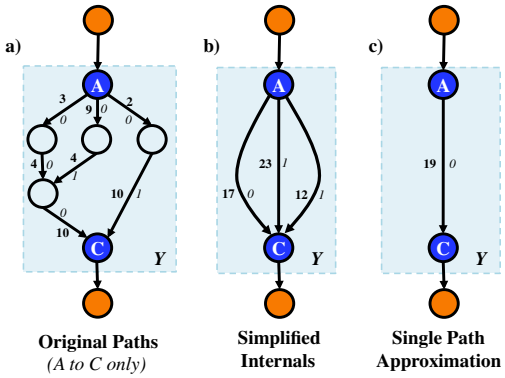


Fig. 5: Performing a single-path approximation on Block Y (*reverse paths not shown*). Delays are shown in bold, weights in italics.

nodes by synchronizing inputs and outputs to a single node each. This simplification is key in order to avoid a prohibitive amount of computational complexity.

As an example, consider Block Y in Figure 5a. For simplicity, only the forward arcs are illustrated in this figure; there will be reverse arcs as well. Also for simplicity, this example is a straight-line specification (i.e., no loops, conditionals, or feedback).

To develop a simplified model for Block Y, we want to hide its internal structure, and retain only the internal paths between its interface nodes. In particular, all internal cycles are removed because, once Block Y has been individually scheduled to meet the throughput constraint, the cycle means of all internal cycles in Y must have been satisfied by the schedule. Next, we enumerate every path between each pair of interface nodes (in this example, A and C). Each such path is replaced by a single arc with a delay and weight equal to the sum of delays and sum of weights, respectively, along the path. The result of performing this abstraction of Block Y is illustrated in Figure 5b. Clearly this simplified model preserves all the information necessary for the next higher level of scheduling: any path or cycle at the next level of hierarchy that goes through Block Y, also goes through the simpler model of Figure 5b, and with the same delay and weight contribution.

C. Single-Path (FIFO) Approximation

After hiding the internals of the block, the block's abstract model consists only of the nodes on the block's interface and their connections to each other. While this model is much simpler, further simplification is necessary to reduce algorithmic complexity. In particular, because there can be a large number of unique paths from one interface node to another, maintaining every one of these paths as an arc in the block's model at the next level of abstraction may still severely impact the runtime of the algorithm. Therefore, we introduce a further model simplification step—*single-path approximation*—to reduce the number of arcs between each pair of nodes. The key idea is to model a block with a pair of interface nodes as simply a FIFO between the nodes.

Theorem. The behavior of a block can be conservatively approximated by a pair of arcs, such that the approximation is exact at the specified throughput constraint.

Proof. The remainder of this subsection offers a proof by construction. Figure 5c illustrates the key idea behind the single-path approximation. Compared to the model of Figure 5b, the three arcs are replaced with a *single arc that dominates* them. The task of computing one arc from node A to C that dominates all of the arcs from A to C in the model is not trivial. We now introduce a formal approach for computing this single-path approximation.

While our analysis approach applies to any arbitrarily large interface, for simplicity, we will continue to consider a block consisting of only two interface nodes (*e.g.*, one input and one output, as in Figure 5b), between which there are multiple arcs with different weights and delays. Remember that each arc represents an entire path through the internal nodes of the block.

To perform single-path approximation, we employ *canopy graphs* as an analysis tool [11] in order to help us visualize the bounds on attainable throughput. This model was originally proposed for pipelined rings and FIFOs, but has been used to model more general pipelined systems [12]. In the basic canopy model, the graph is delimited by three lines: a "data-limited" line with a positive slope that starts at the origin, a horizontal line that represents the slowest stage in a pipeline, and a "hole-limited" line with a negative slope that intersects the x-axis at the maximum occupancy of the ring or FIFO.

The key idea behind our approach is that the performance of a block consisting of one input node A and one output node C can be modeled using a canopy graph. In particular, for each distinct arc from A to C, there is a data-limited line in the canopy graph. Likewise, for each reverse arc from C to A, there is a hole-limited line in the canopy graph. These lines form a convex canopy shape, which defines the region of operation of the block.

The rationale is as follows. Consider one arc from A to C, with a weight of w and delay of d. Assume a steady-state scenario in which k items are occupying the path represented by this arc. Assume at a certain time instant t the node A has produced its i-th event, A_i, and node C has produced C_j. Then, after a time equal to (or greater than) the path delay d, the node C will produce the event C_{i+w} that is dependent on A_i. Therefore, the number of items produced by C is $(i-j)+w$. Hence, the throughput is $\frac{(i-j)+w}{d}$. The canopy graph is drawn so its horizontal axis represents the parameter $k=(i-j)$, and its vertical axis represents the throughput. Therefore, the given arc in the model corresponds to a data-limited line, whose horizontal intercept is $-w$, and slope is $1/d$. All arcs from A and C will have the same $(i-j)$ value, because i and j are associated with the nodes, and therefore can be plotted on a single canopy graph with the same k. Similar reasoning is used to obtain the following result: each arc in the reverse direction, from node C to A, with weight w' and delay d' corresponds to a hole-limited line in the canopy graph, with an intercept w' and a downward slope, $-1/d'$.

The resulting canopy graph is more general than the simple trapezoidal shape associated with rings and FIFOs. In particular, there are three differences. First, since there are in general multiple paths between the pair of interface nodes, there are therefore several data-limited lines (e.g., labeled A–C in Figure 6a). Similarly, there are in general multiple hole-limited lines in the canopy graph, one for each arc in the reverse direction (e.g., labeled D–F in the figure). Thus, the canopy graph is in general more multifaceted. Second, unlike the simple canopy model, in which the data-limited line starts

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

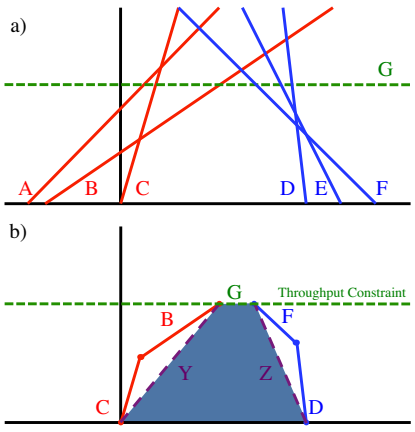


Fig. 6: Our method's approximation of throughput constraints

at the origin, the lines in Figure 6a in general have non-zero intercepts (equal to $-w$) and each may have a different intercept. Similarly, each hole-limited line may have a different intercept (equal to w). Finally, while the canopy graph of a ring or FIFO is typically bounded from above by the throughput of the slowest stage, the canopy graph here is bounded by the throughput constraint set by the designer, shown as line G. The reason for this bound is that, while the circuit may be capable of achieving higher throughput, we can safely remove the region above it from our search space to reduce the solver's run-time. The solid lines $CBGFD$ in Figure 6b represent this canopy graph (lines A and E were dominated by others).

The goal of the single-path approximation is to model this general multifaceted canopy graph by a simple trapezoidal canopy graph. That is, the behavior of the entire block of arbitrary complexity, with one input node and one output node, is approximated by the model of a FIFO. This approximation is illustrated in Figure 6b. Conceptually, the goal is to replace the convex region of the canopy graph with the maximal trapezoid that fits inside the region while meeting the throughput constraint G. The trapezoid YGZ is the canopy graph of this single-path approximation. Since the trapezoid is a proper subset of the convex region, it models only a subset of the runtime behaviors that can be exhibited by the actual block. However, the key feature is that at the throughput of interest (i.e., at line G) the behavior of the FIFO model converges to the behavior of the more complex model. Therefore, at the throughput of interest, the simplified model actually represents the behavior of the more complex model not approximately, buy *exactly*.

The simplified block model is easily constructed from the trapezoidal canopy graph YGZ. In particular, the lower end-point of the segment Y is $(-w_y, 0)$, where w_y is the minimum of all the weights of forward arcs (i.e., rightmost intercept among the data-limited lines). The upper end-point of the segment Y is (k_G, Tpt), where k_G is the maximum horizontal coordinate among the intersections of all the data-limited lines with the throughput constraint G. The intercept and slope of line Y directly provide the weight and delay attributes of the single arc from node A to C in the simplified model of Figure 5c. Similarly, there will be one arc in the reverse direction from node C to A (not shown) whose weight and delay are directly provided by line Z of the trapezoid.

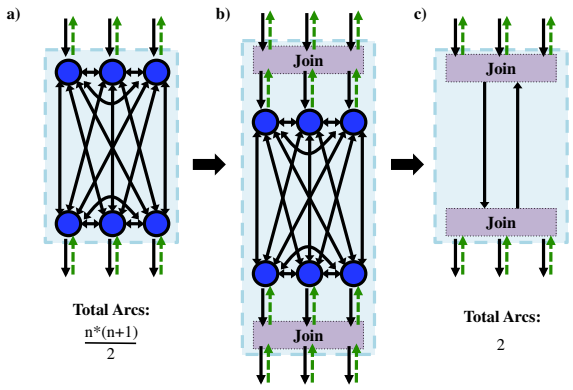


Fig. 7: Converting block to a two-port representation

In summary, this subsection introduced the single-path approximation which allows the graph model of a single-input single-output block to be modeled as a FIFO, using one arc from input to output and one arc from output to input.

D. Two-Port Transformation

Even though the single-path approximation dramatically reduces the model complexity for a single-input single-output block, further simplification may be necessary if a block's model contains multiple inputs and/or multiple outputs. In particular, assume a block has a total of n interface nodes. Thus, in order to correctly hide its internal complexity, single-path approximation must be applied to *every* pair of interface nodes. The total number of arcs in the block's model grows with the number of interface nodes on the order of $\Theta(n^2)$, as shown in Figure 7a.

Therefore, we introduce the *two-port transformation* to further help simplify a block's model. This transformation reduces the number of interface nodes to only two, one representing all the data inputs and another representing all the data outputs. This transformation can be done up-front by the designer by manually modifying the input specification to match the two-port restriction, but can also be automated as in our approach.

The two-port transformation merely requires a synchronization of inputs to the block into a single node, and a synchronization of outputs to a single node, as illustrated in Figure 7b. These synchronizations are performed by inserting join operations on the input and output sides of a block, in which a tuple of distinct values are combined at once into a single larger vector. In order to avoid problems associated with slack mismatches, the channels that carry data into the input join stage and from there to the individual input nodes are allowed to have arbitrary amount of buffering. Similarly, arbitrary amounts of buffering is allowed on the channels that feed the output join stage and that carry the data from the output join to the external environment. It is important to note that the environment of this block sends data values and receive results in a decoupled fashion on distinct channels; the joins are only internal to the block. As a result of this transformation, the block's model is now simplified using the single-path approximation technique into a single pair of nodes and a single pair of arcs, as shown in Figure 7c.

While the two-port specification style can significantly reduce the complexity of the graphical model, the downside is that the simplified model has somewhat less concurrency

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

because of the additional synchronizations of inputs and outputs. Therefore, for the same throughput target, the use of this simplified model may cause the scheduling approach to require more resources than the true optimum, resulting in some area overheads. Additional buffering will generally also become necessary due to slack matching at the input and output synchronization joins. However, the benefit of this approach is significant: the model simplification results in much shorter runtimes, allowing larger real-world examples to be scheduled in reasonable tool runtimes.

Our current approach to the two-port transformation is as follows. If the module already has a small number of ports (*e.g.*, less than a specified parameter), then it need not be simplified. However, if the number of ports exceeds this specified bound, we perform the transformation.

E. Handling Conditionals and Loops

In order to deal with real-world examples, a key requirement is handling conditionals and loops in the specification. While these constructs in general introduce choice in the graph, and therefore typically need more expressive models such as Petri nets, our approach handles them within our graphical model by imposing certain restrictions.

For conditionals, our approach assumes a commonly used implementation style that corresponds to *conditional assignment*: execute both branches of the conditional in parallel and then subsequently selects the appropriate value at the output of the conditional [2], [12]. At the microarchitecture level, this implementation corresponds to a fork-join construct; there is actually no true choice exercised in this structure, and the appropriate result is simply selected using an output multiplexer. Thus, conditional assignments can already be handled by our graphical model without any extensions.

For loops, our approach imposes certain restrictions in order to be able to model them without choice and to avoid out-of-order completion: (i) the loop must have a fixed iteration count, (ii) the loop body must not perform any channel communication actions, and (iii) dependences across problem instances (tokens) are disallowed in loops; however dependences between iterations of a loop for a single problem instance are allowed. With these assumptions, a loop's body is treated just like a non-loop block, and scheduled with its own set of resources. The only modification required is that the throughput constraint T for the overall loop construct must be multiplied by the iteration count N because the loop's body must now process $N \cdot T$ computations per second.

A complete description of our approach to handle conditionals and loops is omitted due to space limitations. However, the automated approach fully supports these constructs, including arbitrary levels of nesting.

F. Absence of Choice

While the architectural model of Figure 4 seems to suggest that the FSM controller must choose where to forward data, in terms of the graphical model, there actually is no choice involved. In particular, the FSM directly implements the schedule encoded in the dependence graph, which itself is devoid of any choice; therefore, the output destination of the data is directly determined by the state of the controller.

IV. Hierarchical Composition and Area-Minimization

Once a block is scheduled and has been replaced with a simplified model, we can then use this simplified model at the next higher level in the hierarchy. In this subsection, we illustrate how blocks are combined when they are constructed sequentially, in parallel, or when one is nested inside another. In this section, we will use examples that are two-port specifications (i.e., one input and one output) for simplicity of presentation, though our approach can handle multiple input and output ports.

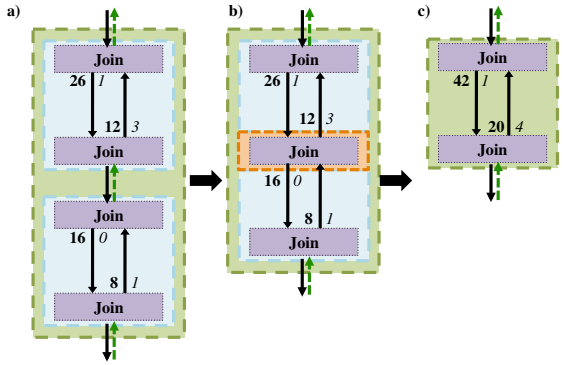


Fig. 8: Composing two sequential blocks

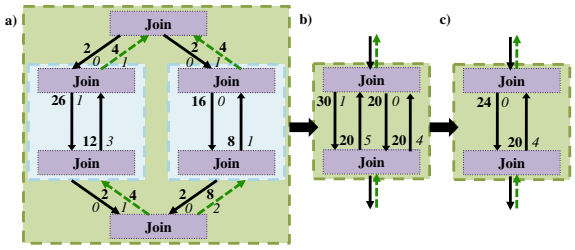


Fig. 9: Composing parallel blocks into a single block

A. Example: Sequential Blocks

Consider the example of Figure 8a, which shows two blocks that have already been scheduled individually and simplified. Each block now contains only a pair of arcs between two join buffers.

Composition of the two sequential blocks is quite straightforward. First, since the data produced at the output of the top block and consumed at the input of the bottom block are the same, we merge the two join buffers into a single buffer, as shown in Figure 8b. Next, we create a single block that represents the sequential behavior of the two blocks. We sum up the weights and delays on the forward path from input to output to produce a representative forward arc in the merged block. Likewise, we sum up the weights and delays on the reverse path to produce a representative reverse arc, as shown in Figure 8(c). This merged block is the simplified model that is propagated to the next higher level in the hierarchy.

B. Example: Parallel Blocks

Combination of parallel blocks is illustrated in Figure 9, using the same pair of blocks from Figure 8. In this example, we have already scheduled the two parallel blocks, and have introduced a pair of synchronizing joins to model a two-port specification of their encapsulating block.

The buffering requirements between the joining buffers and each block have already been determined and are shown in Figure 9a (the sum of the weights in italics on the black and green arcs for each channel). Notice the four channels

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

```
procedure hierarchicallySchedule(block, parent){
   for each child_block in block{
      hierarchicallySchedule(child_block, block)
   }
   scheduleOptimally(block)
   determine interface nodes for block
   perform two-port transformation //optional
   for each pair (A,B) of interface nodes{
      compute all paths from A to B
      add single path approximation arc from A to B
      compute all paths from B to A
      add single path approximation arc from B to A
   }
   remove block internals
   replace block in parent with simplified model
}
```

Fig. 10: Hierarchical area-minimization algorithm

connecting the two parallel blocks to the synchronizing joins at the parent level. While three out of these four channels have a single buffer on them, the fourth one (bottom right) requires two buffers to meet the throughput constraint because of slack matching requirements. These buffering requirements are automatically computing by the basic algorithm of [1] that is used to schedule each individual block (see Section II-A).

In Figure 9b, the internal nodes have been hidden by enumerating all the paths between the interface nodes, leaving two arcs in either direction. Finally, single-path approximation is used to produce the final simplified model, shown in Figure 9c.

C. Nested Blocks

The example of parallel composition illustrated in Figure 9 is actually one of nested composition: in this case there are two separate blocks nested inside an encapsulating block with its own buffering. A more general encapsulating block is capable of having its own computation nodes in addition to any nested blocks.

Arbitrary levels of nesting are handled by our approach, the only caveat being that blocks nested in loops must follow the rules specified in Subsection III-E. In particular, the throughput of the internals of a loop must meet the throughput constraint *multiplied by the iteration count of the loop*. Thus, loops nested within loops will see this factor become the *product* of their iteration counts.

D. Hierarchical Area-Minimization

Figure 10 gives a basic overview of the proposed hierarchical area-minimization algorithm. The algorithm begins with the top-most block, and descends into its children depth-first until a block with no children is encountered (a "leaf" block). The block is scheduled optimally using the approach outlined in [1] in order to produce the minimum area implementation for the block.

Next, the interface nodes are determined for the block. If the two-port transformation is used (specified by designer), input and output synchronization nodes are inserted to transform the block into a single-input single-output block. Otherwise, this step is skipped (typically if the number of interface nodes in the block is less than a set threshold).

In the next step, for each pair of nodes on the interface, the algorithm determines the delay and weight of each path that connects the nodes. Then, a single-path approximation is performed to replace all the paths from one node to the other with a single arc. A similar single-path approximation is computed for the paths that travel in the opposite direction as well.

TABLE I: Function unit parameters

Function Unit	Area (unit)	Latency (unit)
Add	8	32
Subtract	8	32
Multiply	256	64
Shift/Logical	8	32
Buffer	2	1
Merge	3	2
Join	3	2
Guard	1	2

Finally, all other internal nodes and arcs are removed from the block. This new simplified model is then inserted in place of the original block for the purpose of scheduling the operations in the parent block.

If the parent block contains other children that have yet to be scheduled, these blocks are then scheduled and simplified using the same method. Eventually the parent block itself will be ready for scheduling. This recursive procedure continues up the chain until the top level block is scheduled.

V. Results

This section illustrates the proposed method's capability to find an area-minimized solution under a variety of constraints. Utilizing benchmarks from [1], the solver was run for a total of 29 test cases and compared to the results of [1]. These results were verified for correctness using a discrete event simulator that modeled the behavior of each channel, resource, and buffer.

Setup. In experimentation, seven different benchmark DFGs were used for analysis. For each test case, we set constraints for cycle time and optimized for minimum area. We provided the same library of functional units to all benchmarks, shown in Table I. Beyond the initial six benchmarks from [1], one additional benchmark was added, *FIB*, which computes the Fibonacci number of a given input. The *TEA* and *FIB* benchmarks both contained loop structures that were unrolled at varying degrees for a set of experiments.

Discussion of Results. The first set of results is shown in Table II. We compare our approach to that of [1] to illustrate the area overhead and run-times of a hierarchical approach compared to a standard approach. The first two sets of examples, *ODE* and *DP8*, used a maximum block size of 12 nodes, while the other test cases used a maximum block size of 24 nodes. Our algorithm partitioned the given specification based on its hierarchical structure; however, if a single block in the hierarchy contained more than 12/24 nodes, it was partitioned into multiple blocks.

Let us first consider the run-time of each solver. For the non-hierarchical approach of [1], three test cases could not converge within an hour time limit: the *ELP* test cases and one *7TH* test case (in *italics*). Our approach solved each of these in under a minute (under a second for *7TH*).

The run-time of our solver was significantly faster than that of [1] in all cases. For test cases that took longer than a second for the non-hierarchical solver to compute, our approach is several orders of magnitude faster. For example, *7TH-130* had its run-time reduced from 1100 seconds to under a second with less than 1% penalty in area.

Let us now consider the area overheads of the hierarchical approach. The worst case, DP8-258, took over twice as much area than that of a non-hierarchical approach. The high area overhead in this case is due to a low block size and a

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Effect of unroll count and block size on implementation area and tool performance

Benchmark	Unroll Count	Node Count	Block size: 8 nodes Area	8 nodes Time (s)	12 nodes Area	12 nodes Time (s)	16 nodes Area	16 nodes Time (s)
TEA	4	94	2797	1.8	2291	2.3	2389	5.9
TEA	8	174	5927	6.4	5075	23.4	3745	142
TEA	16	334	11163	21.3	9173	95	7131	654

TABLE II: Optimal vs. Hierarchical Approach

Benchmark	Cycle Time Constraint	Optimal [1] Area (unit)	Optimal [1] Runtime (s)	Hierarchical Area (unit)	Hierarchical Runtime (s)
ODE	130	814	0.99	826	0.38
ODE	258	556	0.98	556	0.35
DP8	66	2110	0.62	2314	0.23
DP8	258	806	0.40	1814	0.31
COS	66	4182	12.2	4196	0.68
COS	130	2124	1850	2136	31.1
7TH	98	3954	1100	3970	0.43
7TH	130	2154	*3600	2360	0.57
ELP	66	-	-	2352	16.7
ELP	130	-	-	1238	58.7

* indicates execution incomplete after an hour; best result found is shown. "-" indicates tool did not produce any result within an hour.

TABLE III: Effect of cycle-time constraint and iteration count on implementation area

Benchmark	Iteration Count	Cycle-Time Constraint	Block size: 32 nodes Area	32 nodes Time (ms)
FIB	8	500	128	1100
FIB	8	200	168	580
FIB	8	100	208	880
FIB	8	50	344	700
FIB	16	500	152	1160
FIB	16	200	208	560
FIB	16	100	344	670
FIB	32	500	208	930
FIB	32	200	344	530
FIB	64	500	320	780

sub-optimal automated partitioning. Two other test cases had approximately 10% overhead. While these three test cases with 10% or higher overhead indicate room for improvement in our method, all of the remaining test cases had an area overhead of 1% or less, which shows the approach is quite promising.

In practice, the loss of optimality in total resource area can be attributed solely to the use of disjoint, partitioned sets of resources for each block, as this partitioning prevents sharing across blocks. Optimality loss in buffer area can also be attributed to this partitioning, but may also occur when a two-port model is used for a block.

Now let us consider the set of results shown in Table III. The results in this table run exclusively on the *FIB* benchmark, a specification with a loop construct with its body unrolled eight times. In this table, we see two trends: (i) the total area of a circuit increases as the cycle-time constraint becomes tighter, and (ii) increasing the iteration count of a loop causes a circuit to consume additional area so it can provision an adequate number of resources to provide the desired throughput.

Now, consider the effect of unroll count and block size on the area of a circuit produced by our tool, as shown in Table IV. By unrolling a specification, the total area consumed increases, because additional storage is needed in the pipeline. The total area roughly doubles when increasing unrolling by a factor of 2x. By increasing the size of partitioned blocks, we see that the area cost is generally lower as the solution trends closer to the global optimum. One anomaly exists between *TEA* unrolled 4x with 12-node versus 16-node blocks, again, due to a sub-optimal partitioning.

Finally, consider the run-time of the hierarchical method as a function of benchmark size and block size (Table IV). When approximately doubling the number of nodes from 94 to 174, the execution time of the solver increased by 4-25x, dependent on block size. However, a subsequent doubling increased the execution time of the solver by only 3.5-4.5x. The key observation to take away from this example is that, while the complexity of the hierarchical scheduling approach is certainly not linear, it also does not exhibit exponential increases in run-time based on node size. However, increasing the block size does have a significant impact, because scheduling each individual block is done by the high-complexity non-hierarchical approach of [1].

VI. Conclusion

This paper described a fast, hierarchical approach to attacking the problem of resource sharing in pipelines. A key aspect of this approach is a method for abstracting an optimally scheduled block into a much simpler representation: a single pair of nodes and arcs. Experimental results show a significant improvement in runtime (orders of magnitude for several test cases) with minimal overheads in most test cases (10% or less in all but one test case).

References

[1] J. Hansen and M. Singh, "Multi-token resource sharing for pipelined asynchronous systems," in *Proc. Design, Automation and Test in Europe (DATE)*, 2012.

[2] M. Budiu, "Spatial computation," Ph.D. dissertation, Carnegie Mellon University, Computer Science Department, December 2003, technical report CMU-CS-03-217.

[3] 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.

[4] A. Dasdan and R. K. Gupta, "Faster maximum and minimum mean cycle algorithms for system performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 889–899, 1997.

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

[6] 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.

[7] 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, pp. 2790–2799, 2007.

[8] 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.

[9] 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.

[10] P. Beerel, A. Lines, M. Davies, and N.-H. Kim, "Slack matching asynchronous designs," in *Asynchronous Circuits and Systems, 2006. 12th IEEE International Symposium on*, march 2006, pp. 11 pp. –194.

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

[12] G. D. Gill, "Analysis and optimization for pipelined asynchronous systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 2010, aAI3402352.

Authorized licensed use limited to: University of North Carolina at Chapel Hill. Downloaded on November 20,2026 at 04:40:19 UTC from IEEE Xplore. Restrictions apply.