Bottleneck Analysis and Alleviation in Pipelined Systems: A Fast Hierarchical Approach

Gennette Gill and Montek Singh
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
{gillg,montek}@cs.unc.edu

Abstract—**Fast bottleneck detection and elimination is an important component of any design flow that aims at producing high-throughput systems. Bottlenecks can be difficult to find and correct, because their causes are diverse and often subtle. In this paper, we build on our recent method for performance analysis to develop a method for bottleneck identification and alleviation for pipelined asynchronous systems.**

More specifically, this paper makes two contributions. First, we introduce a method that, given a throughput goal, identifies which parts of the pipelined system constrain its throughput. Each such bottleneck is categorized based on the type of structural transformation that could potentially alleviate it: increase degree of pipelining (stage splitting, stage duplication, and loop unrolling); decrease forward latency (stage merging and parallelization); and perform slack matching. The second contribution is a method that guides the user to systematically apply these modifications to alleviate the bottlenecks and reach a target throughput goal.

We have validated the bottleneck analysis method on several examples and were able to attain the desired throughput goal in each case through iterative application of our bottleneck alleviation method. Runtimes were negligible in all cases (less than 50 ms).

I. Introduction

This paper introduces a fast hierarchical approach to address a key challenge in high-performance design: finding and correcting throughput bottlenecks in pipelined asynchronous systems. Bottleneck detection consists of pinpointing the precise portion of the architecture that is limiting the achievable throughput. Once a bottleneck has been found, our technique systematically assists the designer to apply certain structural transformations to alleviate the bottleneck.

In order to make the problem tractable, we focus on a special class of asynchronous systems: those with *hierarchically composed pipelined architectures.* Such structures are quite common when the system is designed using high-level translation methods (*e.g.*, Tangram/Haste [12], Balsa [6]) because high-level specification languages tend to be hierarchically block-structured. Moreover, even when the design approach is ad-hoc, designers often tend to implement systems with hierarchical structures. In particular, we target architectures that are hierarchical compositions of basic pipeline stages using sequential, parallel, conditional, and iterative operators. By focusing on this special but practically useful class of systems, we are able to leverage information about their hierarchy to provide fast runtimes, thereby making our approach suitable for repeated application in a design flow.

Bottleneck detection and alleviation is critical to any design flow that aims at producing high-performance systems. Bottlenecks can be difficult to find and correct because their causes are diverse and often subtle. In particular, while in synchronous systems, the problem often reduces to that of finding the longest critical path between two clocked registers, the problem is much harder for asynchronous systems because the absence of clocking means the problem is not easily decomposable. Further, for both synchronous and asynchronous systems, merely finding local cycle times is not sufficient because they do not account for complex system-level interactions (*e.g.*, systems with choice, feedback due to algorithmic loops, etc.). In the absence of a fast automated bottleneck analysis tool, a designer typically must contend with costly simulation to search for bottlenecks, a time-consuming and error-prone procedure that can make the typical design-analyze-optimize cycle in the design flow quite inefficient.

Central to the approach of this paper is the concept of *canopy graphs, i.e.*, the graph of a system's throughput as a function of the number of data items in the system (*i.e.*, its occupancy), which are often trapezoidal in shape, resembling a canopy [23], [10], [15]. The key insight used in this work is as follows. Given a system composed of, say, two subsystems, it is generally non-trivial to compute the overall throughput from only the knowledge of the throughputs of the two subsystems. However, given the complete canopy graphs of the two subsystems, it is relatively simple to compute the canopy graph, and hence throughput, for the overall system. That is, while throughputs of subsystems are not directly composable, the canopy graphs are actually composable. Our recent work [7] demonstrates and exploits this property to yield a fast hierarchical method for throughput analysis.

The contribution of this paper is twofold. First, we introduce a fast method for identifying throughput bottlenecks in a pipelined system. To this end, we build upon our recent analysis method as follows: given the canopy graph of a system computed by the method of [7], identify the limits on the achievable throughput, and descend the system's hierarchy to find the cause (*i.e.*, push the "blame" onto lower levels of hierarchy). One or multiple bottlenecks may be identified by this method. The second contribution is a method that helps alleviate the identified bottlenecks. In particular, it accepts a given throughput goal, and reports those bottlenecks that need to be corrected. For each bottleneck, the method suggests one or more structural modifications that may help in correcting that bottleneck, thereby guiding the designer to explore the space of optimizations that may lead to meeting the desired throughput goal.

The bottleneck identification and alleviation methods were implemented in an automated tool, and validated using a number of examples. For each example, the throughput analysis method correctly estimated the system's throughput, as confirmed by Verilog simulation. Further, for each example, the bottleneck identification method was used with a throughput goal set higher than the analyzed throughput. In each case, several bottlenecks were identified. Finally, the alleviation method was used to guide us to correct the bottlenecks by applying structural modifications. After the modifications, performance analysis was used again to confirm that the throughput target was met. In each case, the method successfuly identified and helped alleviate the bottlenecks and reach the throughput goal. The tool's runtime was negligible (less than 50 ms).

a) Previous Work.: While there has been much work on asynchronous performance analysis, the problem of bottleneck analysis has so far not been adequately addressed. A recent approach by Venkataramani *et al.* [22] introduces the notion of a *global critical path* in a system, and uses simulation and profiling to help the designer identify targets for optimization. Although the approach can be useful in identifying bottlenecks, its reliance on simulation can make it time-consuming.

Other prior approaches do not directly target bottleneck identification, but focus instead on finding a system's peak achievable throughput. These include (i) simulation-based approaches [3], [18], [26]; (ii) Markov analysis methods [14], [17], [25]; (iii) methods based on graph unfolding [13], [4]; and (iv) closed-form analytical solutions [23], [9], [19], [15]. The simulation, Markov analysis, and graph unfolding methods all tend to require long running times. The closed-form solutions, on the other hand, apply only to a limited set of architectures (*e.g.,* rings, meshes, linear and simple fork-join pipelines). Recently, a graph-theoretic approach was proposed that avoids graph unfolding to achieve quite fast runtimes [16]. However, all of the aforementioned approaches that are not simulation-based cannot handle systems with choice, thereby limiting their applicability to systems without conditionals or data-dependent loops. Simulation-based approaches generally are able to handle choice, but require long runtimes.

We recently introduced an analytical method that, like this paper, focuses on hierarchical pipelined systems, and is able to provide throughput estimates even for systems with over 100 stages in negligible runtime [7]. The method is able to handle systems with choice. This paper builds upon that approach and extends it to the problem of bottleneck analysis.

The remainder of this paper is organized as follows. Section II provides background on the canopy graph method of analyzing the throughput of asynchronous systems, including a review of our recent hierarchical analysis approach [7] that is the starting point for the work reported in this paper. Section III then covers some preliminaries, including definitions and notation, that will be used throughout the paper. Section IV presents our first contribution: the bottleneck identification method. Section V than presents our second contribution: the bottleneck alleviation approach. Section VI presents experimental results, and finally Section VII gives conclusions.

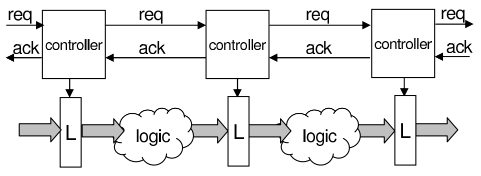

Fig. 1: A simple self-timed pipeline

II. Background: Performance Analysis using Canopy Graphs

This section reviews the methods for throughput analysis using canopy graphs. First, Section II-A focuses on relatively simple structures: linear pipelines and rings. Then, Section II-B reviews our recent approach [7] that extends canopy graph analysis to the more complex structures of conditionals and loops. Later, Sections IV and V will build upon this background to present the new contributions of this paper.

b) Assumptions.: The analysis method reviewed here and the new approaches introduced later make the following assumptions:

- *Handshaking Model:* The pipeline stages used the bundled data model, and are each capable of storing one data token concurrently with other stages (*i.e.*, often referred to as "fully decoupled," or "full buffers," or "high capacity"). The analysis can be easily adapted to "half buffers" as well.
- *Second-Order Effects:* The delay model ignores second-order effects (*e.g.*, the so-called Charlie and drafting effects [5], [24]).
- *Initialization:* The pipeline stages are assumed to be initialized empty upon power-up. The analysis can be modified to accommodate non-empty initialization.

A. Analysis of Basic Pipelined Structures

1) A Pipeline Stage: Figure 1 shows the basic structure of a bundled-data self-timed pipeline. Each pipeline stage consists of a controller, a storage element ("data latch"), and processing logic.

Three key metrics characterize the performance of a single pipeline stage: (i) the *forward latency,* L_{Stage_i}, is the time it takes one data item to flow through $Stage_i$ assuming the stage was empty and ready; (ii) the *reverse latency,* R_{Stage_i}, is the time it takes a "hole" to flow backward through $Stage_i$ assuming the stage was initially full; and (iii) the *cycle time,* T_{Stage_i}, is the minimum time that must elapse between two successive data items entering or leaving that stage. The cycle time depends on the forward and reverse latencies and on the

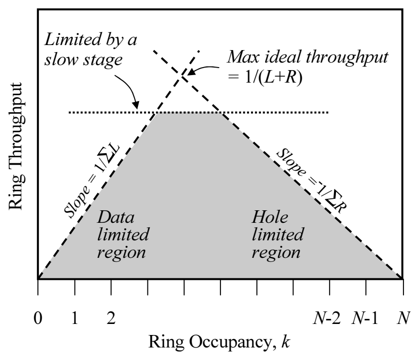

Fig. 2: The throughput of a ring as a function of the number of data items (a "canopy graph")

type of handshaking used. Typically, for full-capacity stages, a complete cycle consists of one forward and one reverse latency, so the cycle time is the sum of the two latencies: $T_{Stage_i} = L_{Stage_i} + R_{Stage_i}$.

2) Self-Timed Rings: The classic work on analyzing self-timed rings is by Williams and Horowitz [23] and by Greenstreet et al. [10], [9]. The throughput of the ring—measured as the number of data items crossing any stage boundary per second— is highly dependent on the ring's *occupancy, i.e.,* the number of data items revolving inside it. In particular, the plot of the maximum throughput versus occupancy, resembles the shape of a canopy, and will be referred to in this paper as a "canopy graph." Figure 2 shows an example.

Data Limited Operation. When the number of data items in the ring is small, the throughput is low because the stages are underutilized, and the pipeline is said to be "data limited." In particular, if there are k items in the ring, then in the time a particular data item completes one revolution around the ring (*i.e.,* $\sum_i L_{Stage_i}$), all k items would have crossed any stage boundary in the ring. Hence, the maximum ring throughput is proportional to the ring occupancy: $tpt_{Ring} \leq k/\sum_i L_{Stage_i}$.

Hole Limited Operation. If the ring is filled with data items in nearly all stages, then the ring throughput is limited because holes are needed to allow data items to flow through the pipeline; the pipeline is said to be "hole limited." If there are h holes in the ring, then in the time a particular hole completes one revolution around the ring (*i.e.,* $\sum_i R_{Stage_i}$), all h holes would have crossed any stage boundary in the ring, traveling in a direction opposite to data. Hence, h data items would have crossed any stage boundary in the forward direction. Thus, if N is the number of stages in the ring, then $h = N - k$, and the maximum ring throughput is proportional to the number of holes: $tpt_{Ring} \leq (N-k)/\sum_i R_{Stage_i}$.

Limitations Due to Local Cycle Times. The ring throughput is also limited by the cycle time of the slowest stage. In the figure, the horizontal line represents the maximum operating rate that can be sustained by the slowest stage in the ring: $tpt_{Ring} \leq 1/\max_i(T_{Stage_i})$.

3) Linear Pipelines: The behavior of a linear pipeline, *under steady-state operation,* can be modeled as that of a self-timed ring. In particular, in steady state, as one item leaves the right end of the pipeline, another item enters on the left. As shown by Lines [15] and Singh *et al.* [21], the linear pipeline's throughput is correctly modeled as a canopy graph, with the same three constraints on its operation (data limited, hole limited, and constrained by local cycle times).

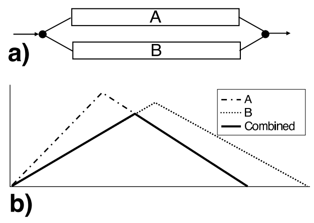

Fig. 3: Parallel composition: a) structure, b) canopy graphs

However, there are two key differences that must be noted. First, the occupancy in a linear pipeline can be fractional. That is, unlike a ring which always contains an integral number of items, a linear pipeline can have an *average* occupancy that is non-integral (*e.g.*, 4.5 items) because of phase differences between entering and exiting items.

Second, a linear pipeline can actually operate in the *entire region under the canopy.* In contrast, the operating point of a ring in steady state is typically on the boundary of the canopy. The reason for this difference is that the ring analysis assumed that the ring is isolated and operates autonomously without any interaction with the environment; therefore, it operates at the maximum throughput possible for a given occupancy. On the other hand, a linear pipeline's operation is constrained by both the left and right environments, which can force it to operate at a throughput below the maximum attainable throughput for a given occupancy.

4) Parallel and Sequential Composition: As shown by Lines [15], canopy graph analysis can also be applied to parallel and sequential compositions of linear pipelines.

For the fork-join parallel structure of Fig. 3a, the throughput of the composition is constrained by the *intersection of the canopy graphs* of the branches. The reason is that the operation of the fork-join pair is constrained so that (i) the throughput of each branch is the same, and (ii) the number of tokens in each branch is the same (assuming they were initialized empty). The result of the intersection of the constituent canopy graphs is also a canopy graph as shown in Fig. 3b.

Similarly, when two pipelines are composed sequentially, as in Fig. 4a, the throughput of the composition is constrained by the *horizontal sum of the canopy graphs* of the two constituents. The reason is that, once again, the throughput of each pipeline must be the same, but the total occupancy is now the sum of the occupancies of the two pipelines. That is, the composition can attain any throughput that both of the pipelines can sustain, and for each such throughput, the net occupancy is simply the sum of the two occupancies. The result is also a canopy graph as shown in Fig. 4b.

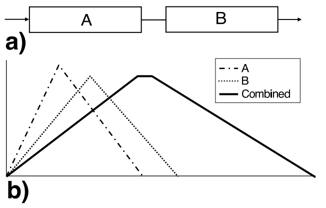

Fig. 4: Sequential composition: a) structure, b) canopy graphs

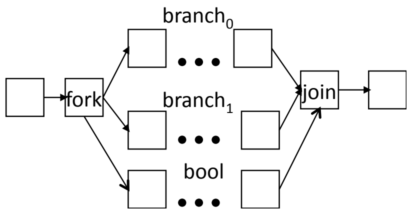

Fig. 5: A pipelined, speculative choice construct

B. Analysis of More Complex Structures

Our recent work in [7] extended the canopy graph method to handle conditional and iterative constructs. The former correspond to if-then-else blocks of computation, and the latter correspond to loops.

1) Conditional Constructs: Two flavors of conditionals are commonly used: speculative and non-speculative. In the former, both branches of computation are allowed to proceed, and the Boolean outcome is then simply used to select the result corresponding to the correct branch. At the structural level, a speculative conditional is simply composed using fork-join constructs, as shown in Fig. 5, where the join stage acts as a multiplexor that reads both results but copies only one of them to the output. Hence, the performance of speculative conditionals can be analyzed using the method just described for analyzing parallel fork-join compositions.

Non-speculative conditionals are implemented as shown in Fig. 6. The incoming stream of data items is *split* into two streams: an item is steered into either the *then* branch or the *else* branch depending on the outcome of a Boolean evaluation. Subsequently, the two streams are *merged* (*i.e.*, interleaved) into a single stream again. The merging is done so as to preserve the original ordering of the data items as they exit the conditional. Therefore, similar to the speculative conditional, the Boolean value is again communicated to both the split and merge stages.

Simplifying Assumption: For clarity of presentation, the analysis described here ignores second-order effects (*e.g.*, effect on performance if Boolean outcomes are correlated/clustered). The complete approach in [7] does cover these second-order effects, however, and the interested reader is referred to it for details.

a) Performance Analysis.: Interestingly, the analysis of a conditional construct is somewhat similar to the analysis of a parallel fork-join pair, but with a slight modification to incorporate branch probabilities. Consider a conditional that has probability p_0 of *branch*$_0$ being chosen and p_1 of *branch*$_1$ being chosen ($p_0 + p_1 = 1$). Then, to a first-order approximation, the average throughput delivered by each branch is proportional to that branch's probability: $\frac{tpt_0}{p_0} = \frac{tpt_1}{p_1}$. Moreover, since items exit the conditional in the same order they entered, the average occupancies of the two branches, k_0 and k_1, to a first-order, are also proportional to the branch probabilities: $\frac{k_0}{p_0} = \frac{k_1}{p_1}$.

In other words, the operation of the two branches is constrained such that their occupancies and throughputs, when divided by their respective branch probabilities, are equal. This result suggests that the constraints on the operation of the overall conditional can be computed by intersecting the canopy graphs of the two branches after appropriate scaling. Specifically, the canopy graph for each branch is scaled so both its axes are divided by the respective branch probability. The region underneath the intersection of the two scaled canopy graphs, which is also a canopy graph, then represents the joint operating region for the two branches. Finally, this result is composed together as a fork-join with the third branch that carries the Boolean outcome (See Fig. 6).

Fig. 7 is an example of a conditional found within the cyclic redundancy check (CRC) algorithm. Suppose it is given that, for this conditional, $p_0 = 0.7$ and $p_1 = 0.3$. Fig. 8 show the canopy graphs for *branch*$_0$ and *branch*$_1$ scaled by $\frac{1}{0.7}$ and $\frac{1}{0.3}$, respectively. The effective maximum occupancy of the scaled canopy graph for *branch*$_0$ is the original branch occupancy, 16, divided by the scaling factor p_0, which gives an effective maximum occupancy of about 23 items. That is, when *branch*$_0$ is completely full the two branches of the pipeline *together* contain a total of 23 items.

The intersection of the two scaled graphs shows that the conditional has an overall maximum throughput of 0.36. Interestingly, these throughput and occupancy values are higher than either of the branches on its own, because the two branches operate on distinct data sets in parallel. Fig. 8 also shows data points from a Verilog simulation, which agree with the results of the analysis. (Finally, this canopy must be composed with the canopy graph of the Boolean branch,

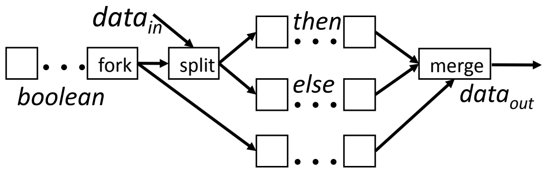

Fig. 6: A pipelined, non-speculative choice construct

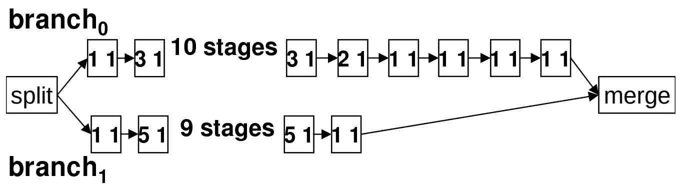

Fig. 7: A conditional from CRC

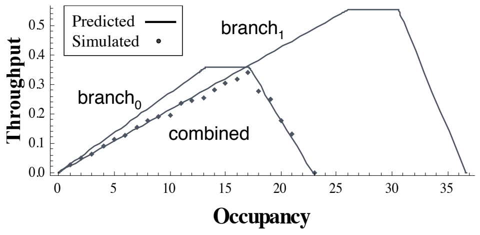

Fig. 8: Canopy graph for CRC at p1 = 0.3

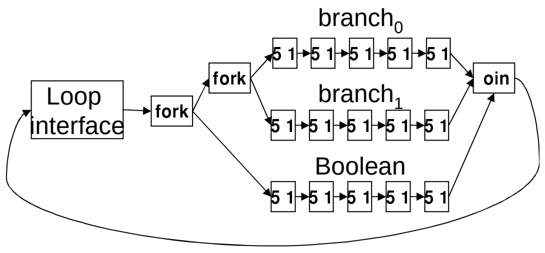

Fig. 9: Pipelined GCD loop

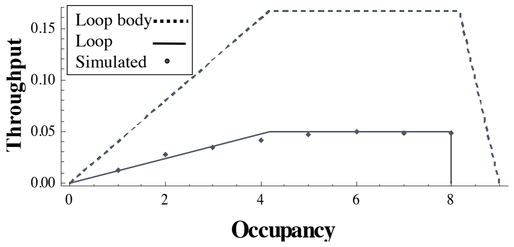

Fig. 10: Canopy graph analysis for GCD

which is ignored for simplicity in this example.)

In general, the performance of a conditional is dependent on the branch probabilities. If the branch probabilities are varied, the scaling factors applied to the respective canopy graphs vary, thereby resulting in different canopy graphs for the result.

2) Iterative Constructs: Loops in high level specifications are typically implemented as self-timed rings in hardware. Although the throughput of rings has been well studied [23], [10], the analysis has mostly focused on isolated rings running autonomously with a fixed number of data items and without communication with an environment. The recent approach of [7] extends the analysis to fully handle various types of loops, and is reviewed here.

First, let us briefly note that loops can be implemented as single-token or multi-token rings. Traditional hardware design methods typically allow only a single token inside a ring. This limits the performance, but avoids the complications of allowing multiple data tokens within the ring. Recent work by Gill *et al.* [8] introduced an approach to implementing certain loops in a manner that allows them to operate on multiple tokens concurrently. This approach to multi-token loops, called *loop pipelining,* handles the flow of control using a special ring interface, and handles structural hazards created by the presence of multiple tokens by adding extra storage where needed. A monitor in the interface prevents overfilling of the loop by limiting its occupancy to some upper bound, k, so as to avoid hole-limited scenarios. The approach assumes that either the iteration count is the same for all data items so that the items exit the loop in order, or that out-or-order completion is acceptable, or if a reorder mechanism is needed, it is present outside the loop.

The method of [7] analyzes the performance of loops (single-token as well as multi-token) using canopy graphs. The method first cuts open the loop, and focuses solely on the loop's body, which could be as simple as a linear pipeline, or a complex hierarchical pipelined block. The performance of this loop body is analyzed to yield a canopy graph. Next, a new canopy graph is computed for the loop as a whole from the canopy graph of the loop's body as follows. If the loop is a single-token ring, then the canopy graph of the loop body is clipped at unit occupancy. On the other hand, if the loop is a multi-token ring with at most k occupancy, then the loop body's canopy graph is clipped at occupancy k. Finally, the canopy graph is scaled down by the number of iterations each token undergoes. This scaling is required because the correct measure of the loop's throughput is actually the number of tokens entering/exiting per second from/to the loop's input and output environment; in contrast, the loop body's throughput is the number of tokens crossing an internal stage boundary per second, which is higher. If the iteration count is variable, the approach approximates the analysis by using the average number of iterations per token.

As an example, Fig. 9 shows a pipeline ring that implements an iterative algorithm for finding the greatest common divisor (GCD) of two numbers. The loop body consists of a conditional, implemented speculatively (both branches are computed, and the correct result selected using the Boolean outcome). Suppose loop pipelining [8] was used to allow the ring to operate with as many as 8 data tokens concurrently, and that the average iteration count is given to be 3.33. First, the canopy graph of the loop body is computed using the method for analyzing parallel fork-join structures, to produce the canopy graph of Fig. 10 (dashed lines). Next, the canopy graph is clipped at occupancy 8, and finally scaled down by the expected iteration count of 3.33 to yield the canopy graph for the loop as a whole, also shown in the same figure (solid lines). For experimental validation, the figure also shows data from Verilog simulation, which follows closely with the result of analysis.

Interestingly, as with linear pipelines, there are two key differences between the performance of loops and that of isolated self-timed rings that operate autonomously without an environment. Much like linear pipelines, a loop's *occupancy can be fractional* because of phase differences between arrival and departure of tokens. Moreover, a loop also can operate in the *entire region under the canopy,* instead of just the canopy's boundary, because its operation may be so constrained by its input and output environment.

III. Preliminaries

This section briefly covers the specifics of how canopy graphs are represented, and how the hierarchical system's topology is represented.

a) Canopy Graphs.: Every canopy graph consists of some number of boundary segments that represent the maximum throughput at each possible occupancy. The operating region of the system is the area below the canopy graph's boundary segments. Figure 11 shows an example canopy graph. The boundaries of this canopy graph consist of the following types of limiting segments: 1) *forward segments* bound the operating region in the data-limited region of the canopy graph and are determined by the forward latency of the system; 2) *top segments* bound the operating region in the cycle limited region of the canopy graph and are determined by the longest effective cycle time; 3) *reverse segments* bound the hole-limited region and are determined by the maximum occupancy and reverse delay of the system. In systems that are initialized as empty, the forward segment of a canopy graphs will contain point at the origin. As a result, such systems will always have exactly one forward segment and one or more reverse segments. In some cases, the top segment may be absent; we treat this as the degenerate case where the top segment is a single point.

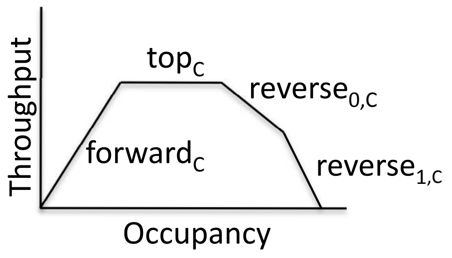

Fig. 11: Canopy graph C with four limiting segments

b) System Representation.: The systems that we consider are hierarchical, so their structure can be represented by a tree in which the leaf nodes represent individual stages and all other nodes are either parallel, sequential, conditional, or loop operators. Figure 13 depicts the tree structure of a hierarchical system. Parallel and conditional nodes have exactly two children, sequential nodes have two or more children, and loop nodes have one child.

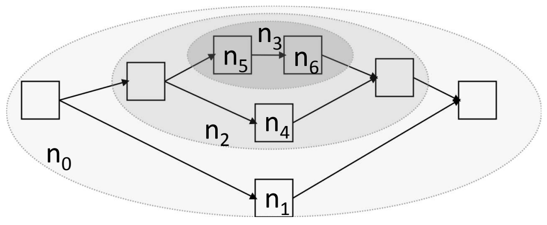

Fig. 12: Block level representation of a hierarchical system

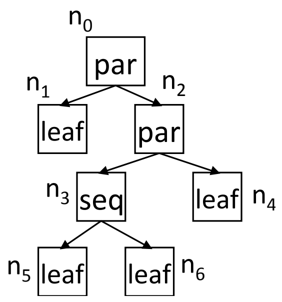

Fig. 13: A tree representing system of Figure 12

IV. Bottleneck Identification

Finding the causes for a system's throughput limits is challenging because the throughput is determined by complex interactions between its constituent stages. Using canopy graphs as the basis for analysis can help expose the full range of causes for a bottleneck. Our algorithm takes as input a system's hierarchical description, the delays associated with individual stages (*i.e.*, forward latency, reverse latency, and cycle time; see Section II), and probability estimates for any choices within the system. It gives as output an expression that represents the parts of the system that limit the throughput. More specifically, the output is an AND-OR expression that specifies how the blame is assigned to the parts identified, using AND-OR causalities.

Our bottleneck detection method consists of three basic steps. The first step is to determine the canopy graph for the entire system, hierarchically from the bottom up, by applying the analysis method of [7]. Next, for each node in the tree that represents the system's hierarchy, we determine which of its child nodes is responsible for limiting its canopy graph. Finally, the set of nodes responsible for limiting the overall system's throughput is determined.

A. Step 1: Compute Overall Canopy Graph

The first step in finding the bottlenecks is to find the canopy graph of the entire system by applying the method of [7]. The analysis begins at the leaf nodes of the tree and progresses upwards. Specifically, the canopy graph for a leaf node is determined by the forward and reverse latencies of the individual stage at that node. Subsequently, the canopy graph at each non-leaf node is found by composing the canopy graphs of its child nodes as described in [7]. During analysis, the algorithm stores in each tree node the canopy graph computed at that node; this information will be used in the next step of the algorithm.

B. Step 2: Find Limiting Segments

Once the canopy graphs have been computed for each node in the system's hierarchical tree representation, the next step is to determine which stages are responsible for the limiting lines of the canopy graphs at each node. When more than one child limits its parents canopy graph, the relationship between contributing children is one of two possibilities: 1) AND-causality: the child is part of a group of stages which jointly limit the parent's canopy and must all be structurally modified to alleviate the bottleneck; or 2) OR-causality: though several children contribute to limiting the parent's canopy, changing any one of them will alleviate the bottleneck. This method is applied to all the nodes in the system's tree representation. The result is an AND-OR tree that identifies the bottlenecks present throughout the system. This method is now described in detail.

Given a node in the hierarchy tree, the method for computing which of its children is causing the parent's canopy to be limited depends on the type of composition represented by the parent node. Each of the four cases is now discussed.

1) Parallel Nodes: The forward segment of a canopy graph at a parallel node is limited by the child canopy graph in which the forward segment has the most shallow slope. If the forward segments of the two children have the same slope, both segments are added to the AND-OR tree using an AND operator, which indicates AND-causality, *i.e.*, both children must be structurally modified in order to change the forward segment of their parent's canopy. If only one child limits the forward segment of the parent's canopy, then that child alone is added to the AND-OR tree (using a degenerate single-input AND operator).

Similarly, each reverse segment of a canopy graph is limited by one or both of its children. If it is limited by both, the appropriate reverse segments of both are added to the AND-OR tree using an AND operator. If only one child limits the reverse segment of the parent's canopy, then only that child is added to the AND-OR tree.

The top segment of the canopy graph at a parallel node is limited in one of two possible ways. First, if any of its children have a top segment with the same limiting throughput value, then that node is added (or those nodes are added) to the AND-OR tree using an AND operator.

Second, the throughput may instead be limited by a slack mismatch. Specifically, the forward segment of one child intersects one of the reverse segments of the other child, thereby limiting throughput of the composition, even though each child individually could have supported higher throughput. In this case, the offending reverse segment of one child and the offending forward segment of the other child are both added to the AND-OR tree using an OR operator. This case is one of OR-causality because, as shown later in Section V, the bottleneck can be removed by modifying either of the two children.

2) Conditional Nodes: The method for identifying bottlenecks at a conditional node is similar to the method for identifying bottlenecks at a parallel node, for all segment types. Specifically, first the canopy of each child is scaled by dividing it by the probability of that branch, as explained earlier in Section II-B. Then, the conditional is treated as a parallel node, and the method described above applied to all segments of the conditional's canopy graph.

3) Sequential Nodes: The forward segment of a canopy graph at a sequential node is affected by the forward segment of every child, so the forward segment of every child of the sequential node is added to the AND-OR tree with an OR operator.

Similarly, each reverse segment of the canopy graph is affected by one reverse segment of every child. Specifically, the reverse segment of the node spans some range of throughputs; any child reverse segment that also falls within this range affects that segment of the parent's canopy. Each of these reverse segments is added to the AND-OR tree with an OR operator.

The top segment of sequential node is limited by the top segment of that child that has the lowest throughput among all children. If more than one of the child nodes have top segments at the same throughput, each of them is added to the AND-OR tree with an AND operator.

4) Loop Nodes: As described in Section II-B, the canopy graph of a loop node is simply the canopy graph of the loop's body scaled down vertically by the expected iteration count. Thus, the forward, top and reverse segments of the canopy of the loop node are limited, respectively, by the forward, top and reverse segments of its child's canopy graph. Hence, the child's (*i.e.*, the loop body's) segments are added to AND-OR tree (using a degenerate AND operator).

C. Step 3: Compute And-Or Bottleneck Formula

Once the AND-OR tree for the limiting segments has been built, we can find an expression for the segments in the entire system that affect the overall throughput. Figure 15 depicts an AND-OR tree for the simple example of Figure 12. The nodes are annotated with the type of limiting segment and the operator of the and-or tree. At each node in the tree, the responsibility can be placed upon the current node OR it can be passed on to the child nodes of the AND-OR tree. The final expression for this systems bottlenecks can be found by recursively combining the expressions for each node.

Figure 14 shows pseudocode for the algorithm that determines this final expression. At each node, the expression for its child nodes are found and combined with the operator at that node. The expression for the node itself is then added. Using the example of Figure 15, at node n_0 first the expressions for n_1 and n_2 are found and then the top segment of n_0 is appended to the expression with the OR operator. The final expression for the bottlenecks in the system is $top_{n_0} + (top_{n_1} \cdot (top_{n_2} + (top_{n_3} \cdot top_{n_4})))$.

```
Expression limitingSegments( node, segment )
    for each child of node
        expression += limitingSegments(child, childsegment)
        expression += operator
    expression = segment "OR" expression
    return expression
```

Fig. 14: Pseudocode for generating limiting segment expression

V. Bottleneck Alleviation

In the previous section, we described an algorithm to find the sets of stages that together limit the throughput of a system.

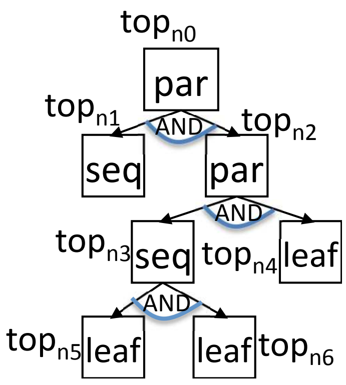

Fig. 15: AND-OR tree for the system of Figure 13

In this section, we classify the different types of bottlenecks and offer some solutions for each kind. Based on the type of bottlenecks found in a given system, we offer these solution choices as part of a user-guided tool for removing bottlenecks to reach some target throughput.

A. *Bottleneck Classification*

To aid in alleviating bottlenecks, we categorize them based on the type underlying problem within the system that needs to be solved.

Type I: Latency Dependent Bottlenecks. Latency dependent bottlenecks are caused by a part of the system having a forward latency that acts as a bottleneck. If a forward segment has been indicated as a limiting segment, this implies that a Type 1 bottleneck exists at that system node.

Type II: Cycle Time Dependent Bottlenecks. Cycle time dependent bottlenecks occur when the cycle time of one part of the system limits throughput. In terms of canopy graphs, if a top segment is indicated by our bottleneck identification method, this implies that a Type 2 bottleneck exists at that system node.

Type III: Occupancy Dependent Bottlenecks. Occupancy dependent bottlenecks are caused by part of the system having insufficient buffering or a high reverse latency. If a reverse segment is indicated as a limiting segment, this implies that a Type 3 bottleneck exists at that system node.

B. *Bag of TRICs*

Each of these TRICs (TRansformations for Increasing the Canopy) raises the throughput for some range of occupancies. Although each of these TRICs has been used before, they have not previously been explicitly analyzed based on how they affect the canopy graph. This list is not intended to be exhaustive but rather illustrative of the use of different throughput optimization techniques within the framework of our approach.

1) Coalescing stages: Two adjacent, sequential pipeline stages can be grouped together into one stage, thereby reducing the total forward latency by removing some latches from the forward path. As indicated in Figure 16, the reduced forward latency leads to expansion of the canopy graph to alleviate Type I bottlenecks. However, this transformation may also lead to and increased maximum cycle time and also reduce the total effective occupancy. Additionally, this TRIC applies only to adjacent leaf nodes composed in sequence.

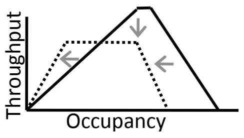

Fig. 16: Coalescing

2) Parallelization: As described in [11], sequential computations can sometimes be changed to occur in parallel, based on the dependency graph of the computations. As indicated in Figure 17, this reduces the forward latency by splitting one sequential path into two parallel paths. However, it also reduces the occupancy of the system. Additionally, this TRIC applies only to nodes that are currently composed in series and has the further restriction that the data dependencies between the nodes must allow for parallel execution.

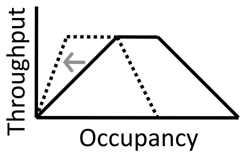

Fig. 17: Parallelization

3) Stage Splitting: Stage splitting increases the level of pipelining by taking a high-latency stage and splitting it into two, lower latency stages. As indicated in Figure 18 this leads to an increase in throughput due to the decreased cycle time of the lower latency stages. It also increases the occupancy of the system by adding a new stage. However, it also increase the forward latency because the stage overhead (*i.e.* latches and stage controllers) increases due to the fine-grained pipelining. Additionally, this TRIC applies only to leaf nodes.

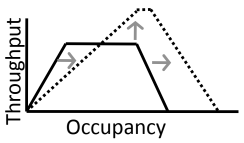

Fig. 18: Stage Splitting

4) Loop Pipelining: Loop pipelining [8] is a method for breaking up a high-latency loop, which essentially acts as a single slow stage within the system. Although the details are more complex than stage splitting, the result is similar and the illustration of Figure 18 that was used for stage splitting applies for loop pipelining as well. In particular, loop pipelining will worsen the forward latency of the loop improving the throughput and increasing the maximum occupancy. This TRIC, of course, applies only to loop nodes.

5) Duplication with Wagging: A wagging buffer [2] improves throughput by sending data alternatingly along one path and then the other. A node and all its children must be fully duplicated in order to apply the wagging buffer method. As illustrated in Figure 19, applying this TRIC increases the throughput as well as the occupancy. It has a second order effect of increasing the forward latency due to the overheads of the wagging buffer, but this effect is constant regardless of the amount of logic duplicated. This TRIC can be applied at any type of node.

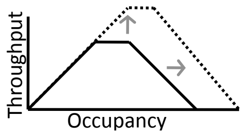

Fig. 19: Duplication with Wagging

6) Loop Unrolling: Loop unrolling [8] improves throughput by increasing number of algorithm iterations that a data item completes during one trip through the ring, thereby decreasing the total number of times that each item must cycle through ring. Loop unrolling is a type of duplication, and has the same effects on the canopy graph as illustrated in Figure 19. Specifically, the throughput and occupancy increase and a small overhead for forward latency occurs. Additionally, this TRIC can be applied only to loop nodes.

7) Buffer Stage Insertion: Buffer stage insertion is a common technique for improving throughput, because it is used to accomplish slack matching [1] [20]. As illustrated in Figure 20 adding buffers increases the occupancy but also worsens the forward latency. This TRIC can be applied at any type of node.

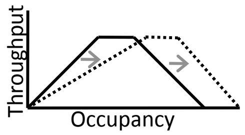

Fig. 20: Buffer Insertion

C. Strategic Application of TRICs

Determining which transform will alleviate a bottleneck in the system is non-trivial because each TRIC applies to only some types of bottlenecks and some types of nodes. Based on the AND-OR tree of bottlenecks, as described in Section IV, our system reports a sum-of-products form of possible TRICs. It then relies on the user's knowledge of the design domain to choose the TRIC that is most compatible with their needs (*e.g.* a designer might avoid excessive duplication if area is a concern.)

a) Determining TRIC Applicability.: Table I summarizes the rules that our tool uses when determining which TRICs to suggest to the user. In the table, a check mark indicates that the TRIC is a good candidate for removing that type of bottleneck, an *X* indicates that it will exacerbate a given type of bottleneck, and a dash indicates that the TRIC will make little to no change in that type of bottleneck.

	Type 1	Type 2	Type 3
Coalescing	√	X	X
Parallelization	√	-	X
Stage Splitting	X	√	√
Loop Pipelining	X	√	√
Duplication	-	√	√
Loop Unrolling	-	√	√
Buffer Insertion	X	-	√

TABLE I: TRICs applicability to the bottleneck types

b) Iterative Algorithm for Bottleneck Alleviation.: The use of our tool for iterative bottleneck elimination is described by the very high-level algorithm of Figure 21. First, the user picks some goal throughput for the system. Next, the algorithm given in Section IV is used to identify the bottlenecks. Then choices of TRICs and the nodes on which to apply them are listed for the user. After applying one or more TRIC, the target throughput might not yet have been met. Often, removing one bottleneck reveals another one, so the method must be repeated until the goal throughput is met or surpassed.

```
while( throughput < goal )
    Find bottlenecks
    List possible fixes
    Apply user design choice
```

Fig. 21: Iterative algorithm for bottleneck alleviation

VI. Results

We tested our bottleneck identification and alleviation methods on a number of examples with varied topologies: **1)** CORDIC: parallel and sequential **2)** CRC: conditional and sequential **4)** DIFFEQ: parallel, sequential, and loop **5)** MULT: parallel We report two different versions of the CORDIC example, one of which contains a conditional and another which does not.

c) Bottleneck Identification.: Table II shows the results of bottleneck identification. The table reports the size of the example in terms of number of nodes in the hierarchical tree representation of the system. It also shows the predicted throughput of the system based on the analysis method of [7], and for comparison shows Verilog simulation results for the same example; this indicates that our analysis is quite accurate for these examples. Stage latencies were chosen such that a FIFO stage has a forward and reverse latency of 1 ns each (*i.e.*, a cycle time of 2 ns). More complex stages had correspondingly longer latencies. The table further reports the number of limiting segments found for each example, and the runtime on a 2.1 GHz Intel Core 2 Duo machine. These results demonstrate that our method is both fast enough to be part of an iterative optimization loop and accurate enough to provide useful bottleneck information.

		Throughput (MHz)		# Limiting Segments Found			
Example	Nodes	Simulated	Analysis	Forward	Top	Reverse	Runtime (ms)
CRC	26	286	292	16	1	12	42
Cordic Cond	30	90.9	90.9	12	2	9	21
Cordic	43	83.3	83.3	0	3	0	40
Diffeq	13	18.2	18.3	0	8	0	27
Mult	29	38.5	38.7	10	1	3	20

TABLE II: Bottleneck identification: finding limiting segments

Example	Throughput			# iterations	Type			TRICS
	orig	goal	final		I	II	III	
CRC	286	342	345	4	1	0	3	coalesce; add bufffers
Cordic cond	90.9	109	111	2	0	0	2	add buffers
Cordic	83.3	100	101	2	0	1	2	split stages
Diffeq	182	218	267	1	3	0	0	split stages; duplicate
Mult	38.4	46.2	62.5	6	5	0	1	coalesce; add buffers

TABLE III: Iterative bottleneck alleviation

d) Bottleneck Alleviation.: Table III shows the bottleneck alleviation results attained through iterative application of our method to reach a goal throughput. Each of the examples presents a different challenge to removing bottlenecks. In order to highlight how examples of similar sizes can require a different number of iterations and different TRICS to reach the same throughput, we set the goal throughput for the iterative algorithm shown in Figure 21 to be the same for each example and iterate till the goal is reached.

In each example, we set the goal to be 20% higher than the original throughput and performed some number of TRICs to reach that goal throughput. For each example, there are many different choices for eliminating bottlenecks; the results here represent one possible set of choices for each example.

In every case, we were able to reach the desired throughput through iterative application of our method. The table reports the number of iterations that we used to reach this goal, the type of bottlenecks that were targeted, and the TRICs used for each example.

VII. Conclusions and Future Work

In this work, we presented a method for identifying the bottlenecks in a hierarchical system. Our results indicate that this method is both fast enough and accurate enough to be used in the optimization loop as part of a design flow. We further presented a user-guided iterative method for alleviating the bottlenecks within a system, based on some performance goal. This method was successful in reaching the target throughput in all examples that we tried.

Though we presented several TRICs for improving the throughput, this list is certainly not exhaustive. It is our hope that others will have suggestions for additional TRICs—whether they are currently known or newly discovered—that will fit into the framework of our tool. The end goal is to have a rich variety of TRICS, each of which is analyzed based on its effect on the canopy graph and usefulness in treating each of the three types of bottlnecks.

This work has focused solely on improving throughput without considering the costs in terms of other useful metrics such as energy, latency, and area. Our tool relies on a designer's knowledge of their domain to choose among the throughput improvement options. In the future, we hope to integrate cost functions into this tool, which will indicate not only throughput improvement but also the tradeoffs present. The end goal is to have a fully automated system that allows designer input for design space exploration.

References

[1] P. A. Beerel, N.-H. Kim, A. Lines, and M. Davies. Slack matching asynchronous designs. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2006.

[2] C. H. K. v. Berkel, C. Niessen, M. Rem, and R. W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[3] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[4] S. Chakraborty, K. Yun, and D. Dill. Timing analysis of asynchronous systems using time separation of events. *IEEE Trans. on Computer-Aided Design*, 18(8):1061–1076, Aug. 1999.

[5] J. C. Ebergen, S. Fairbanks, and I. E. Sutherland. Predicting performance of micropipelines using Charlie diagrams. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 238–246, 1998.

[6] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, 2002.

[7] G. Gill, V. Gupta, and M. Singh. Performance estimation and slack matching for pipelined asynchronous architectures with choice. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 449–456, Nov. 2008.

[8] G. Gill, J. Hansen, and M. Singh. Loop pipelining for high-throughput stream computation using self-timed rings. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2006.

[9] M. R. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI Signal Processing*, 2(3):139–148, Nov. 1990.

[10] M. R. Greenstreet, T. E. Williams, and J. Staunstrup. Self-timed iteration. In C. H. Séquin, editor, *VLSI '87. VLSI Design of Digital Systems*, pages 309–322. North-Holland, Aug. 1987.

[11] J. Hansen and M. Singh. Concurrency-enhancing transformations for asynchronous behavioral specifications: A data-driven approach. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 2008.

[12] The Haste/TiDE Design Flow. http://www.handshakesolutions.com.

[13] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Trans. on Computers*, 44(11):1306–1317, Nov. 1995.

[14] P. Kudva, G. Gopalakrishnan, and E. Brunvand. Performance analysis and optimization for asynchronous circuits. In *Proc. Int. Conf. Computer Design (ICCD)*. IEEE Computer Society Press, Oct. 1994.

[15] A. M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1998.

[16] P. B. McGee and S. M. Nowick. An efficient algorithm for time separation of events in concurrent systems. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 2007.

[17] P. B. McGee, S. M. Nowick, and E. G. Coffman. Efficient performance analysis of asynchronous systems based on periodicity. In *Proc. of the 3rd IEEE/ACM/IFIP Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 225–230, 2005.

[18] E. G. Mercer and C. J. Myers. Stochastic cycle period analysis in timed circuits. In *Proc. Int. Symp. on Circuits and Systems*, pages 172–175, 2000.

[19] P. Pang and M. Greenstreet. Self-timed meshes are faster than synchronous. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 30–39, Apr. 1997.

[20] P. Prakash and A. J. Martin. Slack matching quasi delay-insensitive circuits. In *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2006.

[21] M. Singh, J. A. Tierno, A. Rylyakov, S. Rylov, and S. M. Nowick. An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 GigaHertz. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 84–95, Apr. 2002.

[22] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein. Global critical path: A tool for system-level timing analysis. In *Proc. ACM/IEEE Design Automation Conf.*, pages 783–786, June 2007.

[23] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang. A self-timed chip for division. In P. Losleben, editor, *Advanced Research in VLSI*, pages 75–95. MIT Press, 1987.

[24] A. J. Winstanley, A. Garivier, and M. R. Greenstreet. An event spacing experiment. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 47–56, Apr. 2002.

[25] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75, Apr. 1997.

[26] A. Xie and P. A. Beerel. Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets. In A. Yakovlev, L. Gomes, and L. Lavagno, editors, *Hardware Design and Petri Nets*, pages 239–268. Kluwer Academic Publishers, Mar. 2000.