Performance Estimation and Slack Matching for Pipelined Asynchronous Architectures with Choice

Gennette Gill, Vishal Gupta, and Montek Singh
Dept. of Computer Science
Univ. of North Carolina, Chapel Hill, NC 27599, USA
{gillg,vishal,montek}@cs.unc.edu

***Abstract*— This paper presents a fast analytical method for estimating the throughput of pipelined asynchronous systems, and then applies that method to develop a fast solution to the problem of pipelining "slack matching." The approach targets systems with hierarchical topologies, which typically result when high-level (block structured) language specifications are compiled into data-driven circuit implementations. A significant contribution is that our approach is the first to efficiently handle architectures with choice (*i.e.,* the presence of conditional computation constructs such if-then-else and conditional loops).**

The key idea behind the fast speed of our analysis method is to exploit information about the hierarchy of a given block-structured system, thereby yielding a runtime that is linear in the number of pipeline stages. In contrast, existing approaches typically represent an entire system as a single Petri net or marked graph, and then apply Markov chain analysis or other state enumeration methods with costly runtimes.

Building upon our analysis approach, we introduce a novel solution to the problem of slack matching, *i.e.,* determining optimal insertion of FIFO stages into a pipelined design to improve performance. We present both an optimal solution using an MILP formulation, and a fast heuristic algorithm that yielded optimal results for all of our examples.

I. Introduction

This paper presents an approach for estimating the throughput of pipelined asynchronous systems with choice, and then applies this method to develop a fast solution to the problem of slack matching. The focus is on a special class of systems that typically result when high-level block-structured language specifications are compiled into data-driven circuit implementations. In particular, we target architectures that are hierarchical compositions of basic pipeline stages using sequential, parallel, conditional, and iterative operators. By restricting our focus to this special but useful class of systems, we are able to exploit information about their hierarchy to yield fast runtimes, which allows our tools to be used for repeated analysis and optimization in a typical design flow.

Analysis and optimization of asynchronous systems involves challenges distinct from those of synchronous design. In the absence of globally clocked registers, determining the cycle time of an asynchronous system is quite challenging due to the complex, elastic and highly-concurrent interactions between many locally synchronized (*i.e.,* handshaking) components. The higher concurrency of an asynchronous system also creates new challenges: an otherwise correctly pipelined system may perform poorly because of mismatched latencies along reconvergent branches ("slack mismatch"). *Slack matching* refers to insertion of FIFO stages into a pipelined system to improve performance. It is somewhat related to the problem of retiming in synchronous designs [1] in that both approaches aim to improve throughput through latch placement. However, there is a key difference: in retiming, the number of latches along any cycle must not be changed; in slack matching, extra FIFO stages are deliberately added to certain paths and/or cycles to mitigate mismatched latencies.

Our analysis approach builds upon and significantly extends and generalizes prior work in pipeline performance analysis by Williams et al. [2], Greenstreet et al. [3], and Lines [4]. The past work had shown that the performance of certain asynchronous pipeline constructs—rings ("infinite loops"), fork/join constructs, and sequentially composed stages—can be characterized by a "canopy graph," a convex graph that describes throughput as a function of the pipeline's occupancy (*i.e.,* the number of distinct data tokens that the pipeline operates on concurrently). Our approach extends this analysis to handle conditional computation ("if-then-else"), as well as conditional iteration ("for" and "while" loops), thereby significantly extending the class of systems handled.

This fast analysis method leads to a solution for the slack matching problem. Slack matching requires determining the fewest number of buffer stages that can be added to achieve a target throughput. A straightforward formulation of the slack matching problem is as a mixed-integer linear programming (MILP) problem. Because this formulation is NP-hard, a heuristic method for adding buffer stages provides considerable speed benefits. While not being provably optimal, it still gives the same solution as the optimal MILP formulation for several examples.

Our approaches were validated through experiments on a number of examples. Results indicate that our analysis tool's throughput estimates agreed with Verilog simulations, with an average error of only 1% (max error of 3.7%). The runtime of the tool was practically negligible (less than 10 ms), even for examples with as many as 166 pipeline stages. To the best of our knowledge, no other method has been previously reported to successfully analyze systems of such complexity. In addition, we have applied our slack matching approaches (both the optimal MILP-based approach and the heuristic approach) to several examples. Although the heuristic method could theoretically return different results than the optimal MILP formulation, in all of our examples the two approaches returned the same optimal solution.

The remainder of this paper is organized as follows. Section II discusses previous work in performance analysis and slack matching, and Section III reviews background on asynchronous pipelines and canopy graph analysis. Section IV presents our new hierarchical analysis method, and Section V introduces our slack matching method. Experimental results for both methods are presented in Section VI, and Section VII gives conclusions.

II. Previous Work

Performance Analysis. Although many approaches on asynchronous timing analysis have been reported, none adequately addresses the challenge of efficiently analyzing large pipelined systems with choice. Although simulation-based methods [5]–[7] and Markov analysis methods [8]–[10] could potentially handle the full set of pipelined architectures that our analysis approach targets, these methods require running times that are quite long even more moderately-sized circuits. Previous analytic methods [2]–[4], [11], while fast, handle only a limited set of architectures (*e.g.,* rings, meshes, linear and simple fork-join pipelines). Other methods do not handle choice

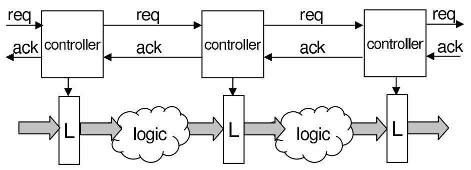

Fig. 1. A simple self-timed pipeline

[12]–[14]; this prevents these methods from being used on systems with conditionals and data-dependent loops.

Slack Matching. The slack matching methods presented by Beerel [15] and Prakash [16] give provably optimal solutions, both in terms of fewest buffer stages added and maximum throughput attained. However, since these methods are based on MILP formulations, they can be slow for large examples. Moreover, neither of these methods handles architectures with conditional computation.

The work by Venkataramani [17] does handle conditional execution in the form of conditional Petri nets. The method is simulation-based and requires a gate level circuit as input. The strategy is to simulate repeatedly to find bottlenecks, adding buffer stages each time. It is not proven to find the optimal solution, either in terms of fewest buffer stages or maximum throughput attained.

III. Background: Analysis Methods

A. Asynchronous Pipelines

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. Without a system-wide clock, stages achieve synchronization locally through handshaking.

Three metrics characterize the performance of a self-timed pipeline: *forward latency*, *reverse latency*, and *cycle time*. The forward latency is simply the time it takes one data item to flow through an initially empty pipeline. Thus, if the latency of Stage$_i$ is L_{Stage_i}, then the latency of the entire pipeline is $L_{Pipeline} = \sum_i L_{Stage_i}$.

Similarly, the reverse latency characterizes the speed at which empty stages or "holes" flow backward through an initially full pipeline. The reverse latency of the entire pipeline is simply the sum of the stage reverse latencies: $R_{Pipeline} = \sum_i R_{Stage_i}$.

The cycle time of a stage, denoted by T_{Stage_i}, is the minimum time elapsed between two successive data items leaving that stage. Cycle time depends on the forward and reverse latencies in the system and on the type of pipeline handshake used. Typically, a complete cycle consists of one forward and one reverse delay, so the cycle time is the sum of the two delays: $T_{Stage_i} = L_{Stage_i} + R_{Stage_i}$.

B. Pipeline Ring Analysis

The classic work on analyzing asynchronous pipelines by Williams and Horowitz [2] introduces the "canopy graph" to characterize the performance of a pipelined ring. The performance of the ring is highly dependent on its *occupancy, i.e.,* the number of data items revolving inside it. When the number of data items is small, the ring performance is low because the stages are underutilized, and the pipeline is said to be "data limited." On the other hand, when nearly every stage of the pipeline is filled with data items, the performance is limited because holes are needed to allow data items to flow through the pipeline, and the pipeline is said to be "hole limited."

Data Limited Operation. If there are k data 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 each ring stage. Let the ring throughput (tpt_{Ring}) be defined as the number of data items passing through a particular stage per second. Then the maximum ring throughput attainable is proportional to the ring occupancy: $tpt_{Ring} \leq k/\sum_i L_{Stage_i}$.

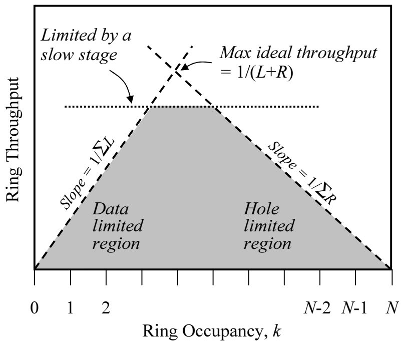

Fig. 2. The upper bounds on the maximum ring frequency: shaded area is the operating region

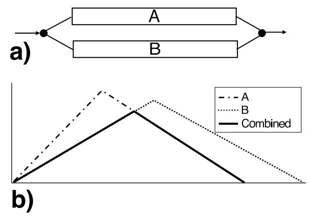

Fig. 3. Parallel composition: a) structure, b) canopy graphs [4]

Hole Limited Operation. If the ring is filled with data items in nearly all stages, then the ring throughput is limited by the number of holes in the ring. 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 the each stage, travelling in a direction opposite to data. Thus, if N is the number of stages in the ring, then $h = N - k$, and we have the following bound on the performance: $tpt_{Ring} \leq (N-k)/\sum_i R_{Stage_i}$.

Figure 2 shows a plot of the ring frequency versus its occupancy. The rising portion of the graph represents the data limited region, where performance rises linearly with the number of data items. The falling portion, similarly, represents the hole limited region, where performance drops linearly with a decrease in the number of holes.

Limitations Due to a Slow Stage. 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 [18]: $tpt_{Ring} \leq 1/\max_i(T_{Stage_i})$.

C. Parallel and Sequential Composition

As presented by Lines [4], canopy graph analysis can also be applied to linear pipelines, and their parallel and sequential compositions. Given the pipeline structure of Figure 3a), the throughput can be predicted using the canopy graphs of each branch. The combined throughput is given by the intersection of the canopy graphs of each branch because the two branches are constrained to have the same throughput and to contain the same number of data items at all times, as shown in Figure 3b).

Two pipelines composed sequentially, as shown in Figure 4a), have a combined canopy graph in which the occupancies of the two pipelines are added for any given throughput. As seen in Figure 4b), the boundary lines of the combined canopy graph fall at occupancies that are the sum of the two original canopy graphs.

IV. Analysis Method

A. Conditional Constructs

Canopy graph analysis can also estimate the throughput of pipelined conditionals that are implemented with choice. Such conditionals implement if/then/else blocks by waiting for the Boolean

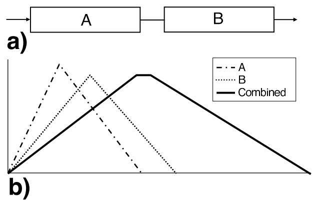

Fig. 4. Sequential composition: a) structure, b) canopy graphs [4]

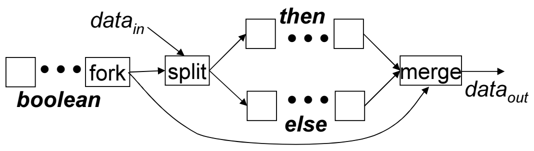

Fig. 5. A pipelined choice construct

decision value to be ready before beginning computation on one of two paths. Figure 5 shows an example of a pipelined conditional. Both the split and merge stages receive the Boolean input, to ensure that ordering is preserved as items exit.

For clarity of presentation, Section IV-A.1 uses two simplifying assumptions that will be relaxed in subsequent sections: 1) the pipeline is slack matched, and 2) the values of the Boolean data are not clustered (*i.e.,* at a probability is 0.5, the Boolean values arrive in this regular pattern: $0, 1, 0, 1...$). Sections IV-A.2 and IV-A.3 relax these restrictions and handle slack mismatch and clustering, respectively.

1) Canopy Graph Method: Let us first develop a relation between the throughputs and occupancies of the two branches of a conditional. Consider a conditional that has probability p_0 of $branch_0$ being chosen and p_1 of $branch_1$ being chosen. For every item that enters (or leaves) $branch_0$, $\frac{p_1}{p_0}$ items enter (or leave) $branch_1$. For example, at a probability of $p_0 = 0.2$, for every data item that enters $branch_0$, 4 items enter $branch_1$. Further, the data items must preserve their original order upon leaving the conditional. Therefore, *under steady-state operation,* the occupancies of the two branches, k_0 and k_1, must be proportional to their respective branch probabilities: $\frac{k_0}{p_0} = \frac{k_1}{p_1}$. Similarly, the branch throughputs are proportional to branch probabilities: $\frac{tpt_0}{p_0} = \frac{tpt_1}{p_1}$.

These equations tell us that the steady-state behaviors of the two branches are constrained such that their occupancies and throughputs, when divided by their respective branch probabilities, are equal. This result suggests that the canopy graph of the combined system 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 area underneath the intersection of the two scaled canopy graphs represents the feasible throughput of the conditional construct.

Figure 6 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 $p_1 = 0.3$. Figure 7 show the canopy graphs for $branch_0$ and $branch_1$ scaled by $\frac{1}{0.7}$ and $\frac{1}{0.3}$, respectively. The intersection of the two scaled graphs shows that the conditional has an overall maximum throughput of 0.36 and a maximum occupancy of 23. 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. Figure 7 also shows data points from a Verilog simulation, which agree with the results of our analysis.

In general, the throughput of a conditional is dependent on the branch probabilities. The maximum throughput of each scaled canopy

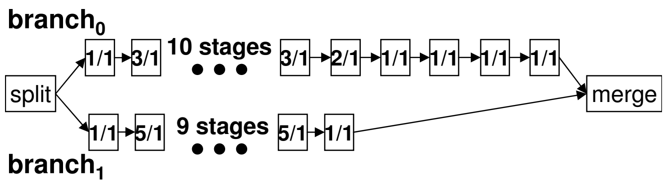

Fig. 6. A conditional from CRC

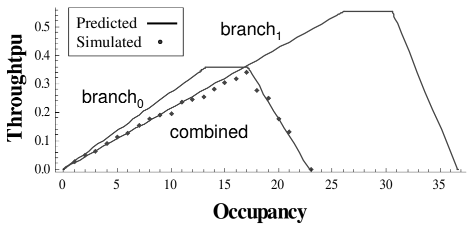

Fig. 7. Canopy graph for CRC at p1 = 0.3

graph limits the throughput of the overall system, as given by:

$$tpt_{max} = min\left(\frac{tpt_0}{p_0}, \frac{tpt_1}{p_1}\right) \tag{1}$$

Applying this equation to the example from Figure 6 yields the graph in Figure 8, which shows the maximum throughput versus the probability of choosing $branch_1$. Data points from Verilog simulation are also shown on this graph, as a validation of Equation 1.

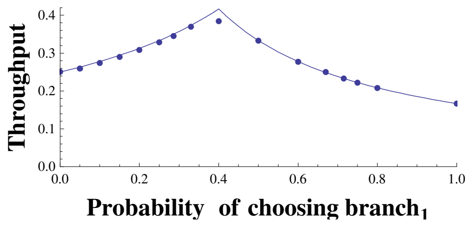

Fig. 8. Throughput depends on probability

2) The Effects of Slack Mismatch: Slack mismatch occurs in conditionals when one branch has too few buffer stages, which causes it to become full and cause stalls. The amount of decrease in throughput due to slack mismatch depends on the probability of each branch being chosen. In particular, as the probability of choosing a branch increases, the frequency of data entering that branch increases, thereby increasing the likelihood that it will become full and cause stalls.

A slack mismatch between the two branches of a conditional can be readily determined by inspecting their canopy graphs. As an illustration, let us return to the example of Figure 6, with the last four stages removed from $branch_0$, thereby decreasing its total number of stages to 12. Scaling the canopy graphs based on the probabilities $p_0 = 0.7$ and $p_1 = 0.3$ results in the graph of Figure 9. The maximum throughput at which the two graphs intersect is actually lower than the throughput of the slower branch, which indicates that slack mismatch is introducing stalling in the system. In particular, stalling occurs when $branch_0$ becomes full and prevents data from entering $branch_1$. The graph also shows data points from simulation, thereby demonstrating that the throughput decreases according to our predictions.

Figure 10 shows the overall throughput of the slack mismatched version of the pipeline of Figure 6 (*i.e.,* four stages removed from $branch_0$) as a function of the branch probability. For comparison, refer to Figure 8 again for the throughput for the slack matched

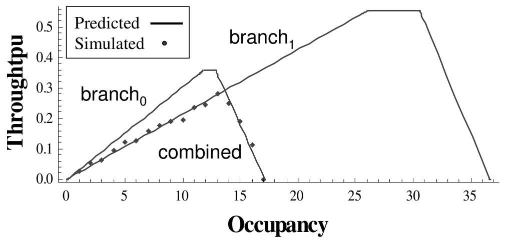

Fig. 9. Canopy graph for CRC at p1 = 0.3 with slack mismatch

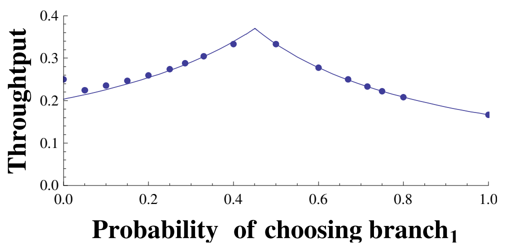

Fig. 10. Slack mismatch leads to throughput degradation

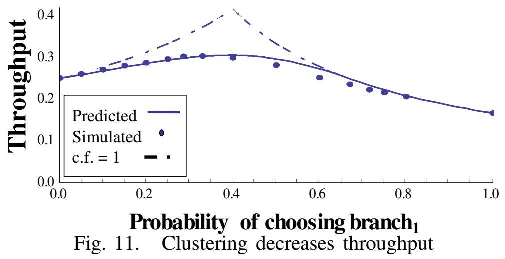

Fig. 11. Clustering decreases throughput

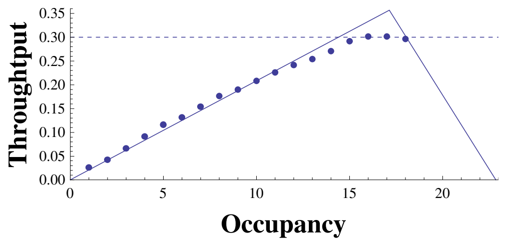

Fig. 12. Clustering decreases max throughput

version. The comparison shows that, as expected, throughput degradation occurs when $branch_0$ is likely to be chosen, and no throughput degradation occurs when $branch_1$ is likely to be chosen.[1]

3) Clustering of Boolean Values: We now introduce a more powerful modeling of choice that considers *correlation* between successive Boolean outcomes, which is captured in a metric termed the "cluster factor". Intuitively, the clustering factor quantifies the average run lengths of 1's and 0's in a sequence, normalized with respect to the base case given in Section IV-A.1 This quantity effectively models correlation between successive Boolean outcomes. A clustering factor of 1 indicates a negative correlation, which is the simple case considered in the previous sections. A high cluster factor indicates a positive correlation, in which long runs of 1's and 0's may occur. For zero correlation (*i.e.,* independent outcomes) the clustering factor is related to the probability. Specifically, the expected run length of ones for random, uncorrelated data is $E(run_1) = 1/p_0$.[2]

During circuit operation, clustering prevents throughput from reaching the maximum value predicted using the canopy graph method described in Section IV-A.1. The effective maximum occupancy also decreases, since clustered Boolean values lead to uneven filling of the two branches. However, the slopes of the canopy graph lines, which are determined by the forward and reverse latencies of the pipelines, do not change due to clustering.

The reduced maximum throughput can be written as a modified version of maximum throughput Equation 1, with the cluster factor now included. This equation as written considers the situation in which the throughput of $branch_0$ is higher. In this Equation, run is the run length based on probability and $cluster$ is an estimate for the clustering factor, based on knowledge of typical input data sets.

$$tpt_{max} = \frac{cluster \cdot (run_0 + 1)}{cluster \cdot run_0 \cdot \frac{1}{tpt_0} + cluster - 1 \cdot \frac{1}{tpt_1}} \qquad (2)$$

Applying this equation to the example of Figure 6 and assuming random, uncorrelated Boolean values, the solid line in Figure 11 shows how the maximum throughput varies with the probability of choosing $branch_1$. For comparison, it also shows the maximum throughput with a clustering factor of one (*i.e.,* completely anti-correlated Boolean values). In addition, the figure shows data from Verilog simulation, which is close to our predicted curve.

Clustering also affects the average maximum occupancy of the pipeline. Simple statistical methods determine the expected maximum occupancy of a pipeline with clustering—the expected value is found by summing the probability of each occupancy occurring multiplied by that occupancy.

To show how the canopy graph is affected by clustering, we use the example of Figure 6 once again. If $p_1 = 0.3$ and the Boolean values are random and uncorrelated, Equation 2 predicts that clustering will reduce the throughput to 0.30. Figure 12 shows how the canopy graph will be cut off at the value 0.30. In this example, the expected maximum occupancy due to clustering is 22.75, which is only slightly lower than the non-clustered maximum occupancy. Notice that the slopes of the boundary lines remain the same, although the throughput is limited by the predicted value of 0.30.

B. *Iterative Constructs*

Loops in high level code are implemented as rings in hardware. Although the classic work by Williams [2] studied the performance of rings, it finds the throughput within the ring whereas our technique finds the throughput of items leaving a ring based on some condition.

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 [19] presents an approach to implementing *for* and *while* loops that operate on multiple tokens concurrently. This loop pipelining technique handles the flow of control and data dependency challenges created by allowing multiple tokens in the ring by including a special ring interface and duplicating data values within the ring. A monitor in the interface prevents overfilling of the loop by limiting its occupancy to some optimal value, k.

We estimate the performance of both the traditional single occupancy rings and pipelined loops using a canopy graph based method. We start with the a canopy graph for the body of the loop. For traditionally implemented loops that contain only one token, we cut the canopy graph off at occupancy 1. Similarly, for a pipelined loop with a maximum occupancy of k, we cut the canopy graph off at k. Since we are interested in the loop's throughput (*i.e.,* the rate at which tokens leave the ring) the canopy graph is simply scaled down by the number of iterations each token undergoes. If the iteration

[1]Notice that in the neighborhood of $p_1 = 0$ our analysis deviates slightly from the simulation results; in this region, the steady-state assumption does not hold.

[2]Expected value is found by summing this infinite geometric series $\sum_{n=0}^{\infty} p_1^n = \frac{1}{1-p_1}$

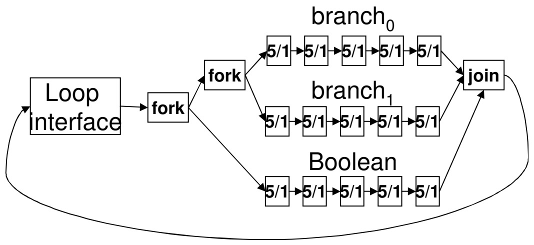

Fig. 13. Pipelined GCD loop

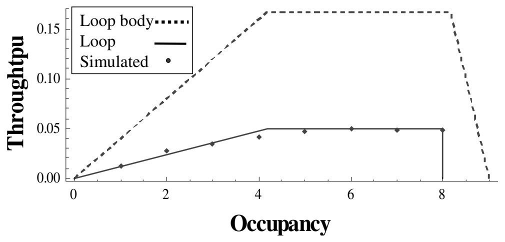

Fig. 14. Canopy graph analysis for GCD

count is variable or data-dependent, our approach approximates the throughput using the expected number of iterations. The model for expected occupancy is that a weighted coin-toss takes place at the beginning of each iteration.

As an example, Figure 13 shows a pipeline ring that implements an iterative algorithm for finding the greatest common divisor of two numbers. For this analysis, we assume that the ring interface allows 8 data items into the ring concurrently and that the average iteration count is 3.33. First, we find canopy graph of the loop body using the process given in Section III-C, to produce the canopy graph shown in Figure 14. Next, we scale the graph down by the expected iteration count, 3.33, and cut it off at the max occupancy, 8. Figure 14 shows the resulting canopy graph. The figure also shows data from Verilog simulation, which follow closely with our predicted canopy graph.

C. Analysis Algorithm

```
CanopyGraph analyze( tree t )
 if( currentNode == parallel )
    return parallel(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == if )
    return conditional(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == loop )
    return loopize(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == sequence )
    return sequence(analyze(t.leftchild), analyze(t.rightchild))
 if( currentNode == leafnode )
    return CanopyGraph( Lf, Lr )
```

Fig. 15. Our analysis algorithm

A complete analysis algorithm uses the composition functions for conditional constructs described in Section IV as well as those given in Section III. Figure 15 the shows pseudocode for the algorithm, which generates the canopy graph for the whole system by traversing the hierarchy.

The algorithm takes as input the description of a pipelined system in the form of an AST annotated with stage delays. The algorithm must also have access to the branching probability for each if/then/else construct and while loop conditional. It then proceeds much like expression evaluation: in each step, two throughput expressions (*i.e.,* canopy graphs) are combined based on their parent operator.

```
Diffeq(x, y, dx, u, a)
1 while ( x ¡ a )
2 (x1 := x+dx;
3 t1 := u*x;
4 t2 := t1 + y)
||
5 (t3 := 3*dx;
6 t5 := u*dx;
7 y1 := y+5)
8 t4 := t2*t3;
9 u1 := u - t4
<x1, u1, y1> = <x, u, y>
```

Fig. 16. Differential equation solver

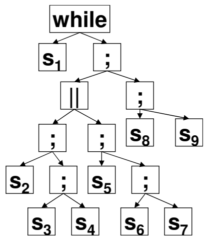

Fig. 17. Abstract syntax tree

As an example, Figure 16 shows the high-level code for an iterative differential equation solver, which has been parallelized. Figure 17 shows an abstract syntax tree representation, and Figure 18 shows one possible pipelined architecture.

The algorithm traverses the tree; when it encounters the parallel operator, it finds the two canopy graphs of the two branches {s2, s3, s4} and {s5, s6, s7}, as shown in Figure 19. As explained in Section III-C, the parallel operator creates a new canopy graph that is the intersection of these two. This graph is then composed in sequence with the remaining stages, thereby giving the graph shown in Figure 20. Finally, the method for finding the canopy graph of a loop, given in Section IV-B, produces the canopy graph shown in Figure 20. In this way, our algorithm can handle any system with arbitrary levels of nesting, by working from the bottom up.

V. Slack Matching Method

Slack matching is the problem of determining the number and positioning of additional pipeline buffers in a pipelined design in order to reach some performance goal. We offer both a mixed integer linear programming solution, which is NP-hard but optimal, and a heuristic algorithm, which is faster but not provably optimal.

A. MILP Formulation

Slack matching usng canopy graph analysis seems to lend itself quite naturally to a linear programming solution. The formulation of a linear programming problem requires a function to minimize: in this case, the total number of buffer stages added (*i.e.* s_1 + s_2 + s_3 ... + s_n). A linear programming formulation also requires a set of constraints to satisfy, in the form of inequalities. In the case of slack matching the throughput is limited to be beneath the lines of the canopy graphs.

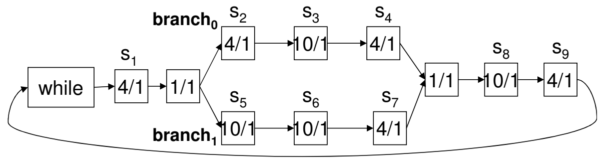

Fig. 18. Pipelined implementation of Diffeq

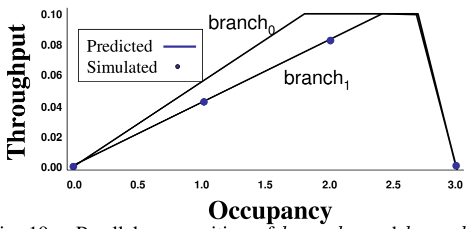

Fig. 19. Parallel composition of $branch_0$ and $branch_1$

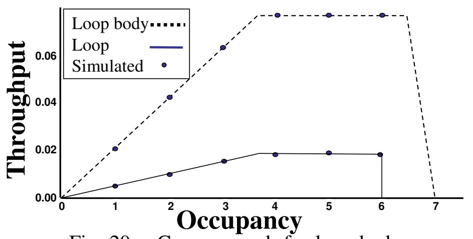

Fig. 20. Canopy graph for loop body

While Sections III and IV gave an intuitive understanding of each composition function used in our analysis algorithm, this section formalizes the work. In particular, we represent canopy graphs as sets of inequalities, and formalize the composition functions that operate the canopy graphs. These representations explicitly include the effects of slack matching buffers.

Although equations for calculating the sequential and parallel dynamic slack are given in the previous work by [4], we reformulate them here to do all of the following: (i) represent non-trapezoidal canopy graphs, (ii) explicitly include the effects of slack matching stages, and (iii) cast them in a form suitable for an MILP solver.

1) Canopy Graph Equations: Inequalities of the following form can represent any canopy graph, cg_i. In particular, each canopy graph representation consists of one inequality in the form of 3 that forms the horizontal cutoff line, one in the form of 4 that forms the positive sloped forward line, and one or more in the form of 5 that represent the negative sloped reverse lines.

$$tpt_i \leq \frac{1}{T_i} \tag{3}$$

$$tpt_i \leq \frac{k}{F_i + s_i \cdot f_s} \tag{4}$$

$$tpt_i \leq \frac{-k}{R + s_i \cdot r_s} + \frac{N_i + s_i}{R_i + s_i \cdot r_s} \tag{5}$$

where T_i is the highest local cycle time of the stages in cd_i, F_i is the sum of the forward latencies of stages in cg_i, R_i is the sum of the reverse latencies of stages in cg_i, N_i is the number of stages in the pipeline, s_i is the number of slack stages to add to cg_i, and f_s and f_r are the forward and reverse delays of a slack stage.

2) Composition Functions: Canopy graph composition functions operate on the inequalities that represent the canopy graphs. The conditional and loop transforms consist of scaling and line intersection. Parallel composition consists of line intersection while sequential composition is equivalent to adding the inverse of two line functions. The following gives composition functions in terms of forward and reverse delays, number of stages, and number of slack stages added.

Conditional Composition. Conditional composition consists of scaling two canopy graphs based on their probabilities, as discussed in Section IV-A. In order to represent canopy graph cg_3 that is formed by composing two canopy graphs, cg_1 and cg_2, in a conditional construct scale the canopy graph inequalities for cg_1 and cg_2 by dividing the right side of the inqualities of Eq. 3–5 by the probability of each branch being chosen.

In addition, if clustering is present, as discussed in Section IV-A.3 further modifications are needed: a new maximum throughput is added and the effective occupancy changes based on the expected occupancy. In particular, the new maximum throughput is determined using the probabilities of each branch being chosen along with the current maximum throughput of the two branches, as given in Eq. 2. Also, the effect that new slack stages have on the increase in occupancy will be changed based on the expected occupancy. That is, inequalities of the form 5 should be re-written as follows to take into account the expected occupancy equation.

$$tpt_i \leq \frac{-k}{R + s_i \cdot r_s} + \frac{expected(p_i, N_i + s_i))}{R_i + r_i \cdot r_s} \tag{6}$$

Note that while the effective occupancy increase caused by adding slack stages decreases based on the expected occupancy, the effect on the reverse latency remains the same. The modified inequalities can be used in further hierarchical composition steps in the same way as the original equations.

Loop Composition. Loop composition, discussed in Section IV-B, is accomplished by scaling the canopy graph, cg_1, of the loop body and applying a cutoff token capacity. To scale the canopy graph inequalities, simply divide the right side of the inequalities that represent canopy graph cg_1, as shown in the inequalities of Eq. 3–5, by the expected number of loop iterations. In addition, a cutoff line indicating the limited occupancy, based on the designer specified loop capacity, must be added.

$$N_i \leq K_i \tag{7}$$

where K_i is the maximum occupancy of the loop represented by canopy graph i.

Parallel Composition. As shown in Section III-C two pipelines structures in parallel have a combined canopy graph that in the area under both canopy graphs. In order to represent canopy graph cg_3 that is formed by composing two canopy graphs, cg_1 and cg_2, in parallel, it is sufficient to simply retain all of the inequalities of Eq. 3–5 and 7 of both cg_1 and cg_2.

Sequential Composition. As discussed in Section III-C, putting two pipelined structures in series causes the total number of tokens at each throughput to be added. The following inequalities—Eq. 8, 9, and 10—represent part of the canopy graph, cg_3, that is formed by composing two canopy graphs, cg_1 and cg_2, in sequence. The are formed using the canopy graph inequalities of Eq. 3–5 respectively.

$$tpt_3 \leq \frac{1}{max(T_1, T_2)} \tag{8}$$

$$tpt_3 \leq \frac{k}{F_1 + s_1 \cdot f_s + F_2 + s_2 \cdot f_s} \tag{9}$$

$$tpt_3 \leq \frac{k}{R_1 + s_1 \cdot r_s + R_2 + s_2 \cdot rs} + \frac{N_1}{R_1} + \frac{N_2}{R_2} \tag{10}$$

Note that any occupancy cutoff line of form 7 introduced through loop composition can actually be handled using inequality 10. In particular, the cutoff line has no reverse latency R and is not affected by the number of slack stages s, so both of those terms are zero within the inequality. To find all of the inequalities that represent canopy graph cg_3, it is necessary to perform these operations for each set of inequalities that represent cg_1 and cg_2.

B. Heuristic Algorithm

Our heuristic algorithm performs slack matching by starting at the lowest level of the hierarchy and then moving on to successive levels. At each level, it determines which of two branches needs additional stages and then adds stages somewhere along that path, possibly descending down the hierarchy. Because the algorithm uses

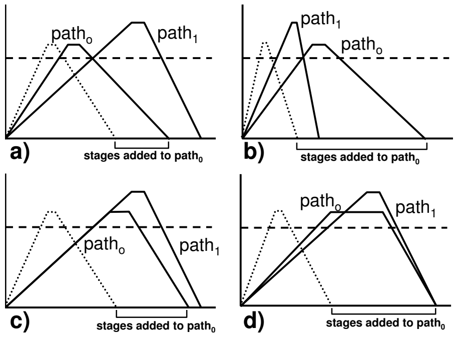

Fig. 21. Key canopy graph intersections

no backtracking or exhaustive search, it is faster than our MILP solution and other past slack matching algorithms [15], [16].

Heuristic. In order to understand our heuristic for adding stages, note that the area under a canopy graph represents a set of conditions under which the circuit can operate. Any buffer stage addition that expands the area under the canopy graph—such that the resulting canopy graph is a superset of the previous canopy graph—will never be detrimental to the performance of the system. Our heuristic for the placement of a buffer stage, therefore, is to compare choices for stage placement based on which will expand the canopy graph most.

Four types of intersection calculations are important in determining the number of stages to add and finding which stage placement will expand the canopy graph most. To describe these intersection scenarios throughout this section, the terms "forward" and "reverse" generically refer to the boundary lines of the canopy graphs in the data limited and hole limited regions respectively.

The first interesting canopy graph intersection point is at the minimum number of buffer stages needed to achieve a goal throughput for two paths, $path_0$ and $path_1$. Figure 21(a) shows how adding slack stages to $path_0$ creates an intersection between the two canopy graphs at the desired throughput level. To find this number of stages, use the reverse line of $path_0$, the forward line of $path_1$ and solve for the number of stages, s_0, to add to $path_0$.

At some point, however, placing additional stages on $path_0$ will cause the throughput to degrade. Figure 21(b) shows the point at which placing additional buffer stages will decrease throughput. Finding the number of stages is identical similar to the previous intersection point, except the forward line of $path_0$ and the reverse line of $path_1$ are used.

At some number of added slack stages, which we call the *forward degradation* point, the slopes of the two forward lines will be identical, as illustrated in Figure 21(c). To find this number of stages, set the two forward lines equal to each other and solve for the number of stages to add to $path_0$. Note that adding further slack stages will cause the combined canopy graph to have a forward line with a shallower slope, thereby reducing the maximum throughput possible *at some occupancies.* This condition will lead us to avoid placing stages along this path, if possible, because it will cause the canopy graph to get smaller, which goes against the stage placement heuristic.

Finally, Figure 21(d) shows that adding stages can cause the effective occupancies of the two canopy graphs to be identical. At this point, which we call the *reverse degradation* point, adding further stages does not increase the maximum occupancy of the resulting canopy graph. This condition will lead us to avoid placing stages along this path, if possible, because it will not expand the canopy graph.

Algorithm. Our slack matching algorithm uses the canopy graph expansion heuristic to decide on slack stage placement. Pseudocode for this algorithm is shown in Figure 22. At each level the algorithm calculates and stores the minimum and maximum number of stages for one of the branches.

When placing stages at each level, it first descends down to the lowest level along that path. It will add stages at that level in order to slack match *unless* adding a stage will cause it to reach the maximum number of stages, the forward degradation point, or the reverse degradation point. It repeats this at each level, up to the current level, until the minimum number of stages has been added.

```
For each level of the hierarchy
    Calculate min and max for current level
    Descend to lowest level of hierarchy
    While min is not yet reached
        Add stages till reaching max or degradation points
        Move up to next level
    Store max stages and degrad point for current level
```

Fig. 22. Our slack matching algorithm

VI. Results

Analysis Method. The analysis algorithm of Section IV-C was implemented in Java and run on a Pentium 4 2.4GHz CPU machine. The tool was used to analyze the performance of six examples, which cover all of the different hierarchical compositions: **1)** CORDIC: parallel and sequential **2)** CRC: conditional and sequential **3)** GCD: nested parallel and loop **4)** DIFFEQ: parallel, sequential, and loop **5)** MULT: parallel, and **6)** Raytracing: conditional with nested loop. For validation, each example was also specified in behavioral Verilog and simulated for 1000 randomly generated data items. Stage latencies were chosen such that a FIFO stage has a forward and reverse latency of 1 ns (*i.e.,* a cycle time of 2 ns); more complex stages had correspondingly longer latencies.

Table I compares the throughput found through Verilog simulation to the throughput computed by our analysis method and reports the percent error. For two examples, CRC and CORDIC, experiments were performed both on the inner conditional construct ("cond") which formed the core of the algorithm, as well as on the full implementation ("full"). Where applicable, results for both the slack matched ("match") and non-matched ("mis") versions of some examples are reported separately. In addition, the results for the inner conditional of the CRC algorithm ("CRC cond") are provided both for unclustered inputs and for inputs with clustering ("cluster").

Results show that the performance estimates produced by our analysis method are within 3.7% of the results obtained by simulation (average 1% error). Most notably, our method accurately predicts the maximum throughput of the large, complex raytracing kernel example ("raytracing match," with 166 pipeline stages) with negligible error. The results also confirm the speed of the algorithm: for all examples, the runtime was approximately 10 ms or less. Therefore, our analysis method is fast and accurate enough to be used for repeated analysis and optimization as part of a design flow.

Table II compares our analysis tool's runtime on linear FIFOs of varying lengths with that of one previous approach [9]. The results show that while the runtime of our tool is negligible, the previous approach becomes quite slow as the FIFO length increases to 11 stages and beyond. A more detailed comparison with other prior approaches has not yet been possible without access to those tools and examples. However, prior approaches are typically either based on Markov analysis or other state enumeration methods [8]–[10], or based on stochastic simulation [5]–[7], and therefore all exhibit a steep increase in runtime with design size. Further, many

TABLE I

RESULTS OF THE PERFORMANCE ANALYSIS METHOD: VALIDATION AGAINST SIMULATION

	Size	Throughput (MHz)		Error	Runtime
Example	(stages)	Simulated	Predicted	(%)	(ms)
CORDIC cond	31	167	167	∼0	∼10
CORDIC cond mis	23	90.9	90.9	∼0	∼10
CORDIC full	44	83.3	83.3	∼0	∼10
CRC cond	27	352	357	1.4	∼10
CRC cond cluster	27	305	300	1.6	∼10
CRC cond mis	23	292	286	2.0	∼10
CRC full	67	333	333	∼0	∼10
DIFFEQ	10	18.3	18.2	0.5	∼0
GCD	21	49.0	50.0	2.0	∼10
raytracing mis	21	161	167	3.7	∼10
raytracing match	166	222	222	∼0	∼10
MULT mis	13	38.7	38.4	1.9	∼0
MULT match	21	167	167	∼0	∼0

TABLE II

COMPARISON OF RUN TIME WITH A PREVIOUS APPROACH [9]

Size	Runtime	
(stages)	McGee [9]	Ours
3	0.003 s	∼ 0 ms
5	0.031 s	∼ 0 ms
7	0.986 s	∼ 0 ms
9	29.835 s	∼ 0 ms
11	4686.126 s	∼ 0 ms

previous methods do not handle systems with choice [2]–[4], [11]–[14]. In contrast, our approach does not require state enumeration or simulation; it leverages the hierarchy information to provide a dramatically fast runtime, and can handle architectures with choice.

Slack Matching. Table III shows experimental results for our slack matching approach. Each example was slack matched using both the MILP-based approach and the heuristic approach. The number of FIFO stages added by each of the two approaches is reported in the table. For all examples, the heuristic algorithm placed stages optimally (identical to the MILP approach), even though the examples were chosen to represent a wide range of architectures, including nestings of parallel and sequential constructs, and conditionals and loops. This result indicates that the heuristic approach will often lead to an optimal solution. The table also shows the throughput determined via Verilog simulation for each example, both before and after slack matching was performed, and the speedup obtained (up to 332%).

Note that three out of the seven examples did not require the insertion of any slack stages. These examples are actually quite interesting, because they highlight the fact that the heuristic algorithm will successfully avoid adding unnecessary stages. For example, the inner conditional of the CORDIC algorithm, when analyzed individually, requires 8 slack stages to operate at its maximum speed. However, when this conditional is nested within the rest of the CORDIC system, our approach chooses not to insert any stages. It successfully determines that parts of the system outside the conditional act as a bottleneck, and that inserting stages anywhere will not improve performance.

VII. CONCLUSIONS

This paper introduced methods for performance analysis and slack matching for asynchronous pipelined systems. These methods are fast because they are able to take advantage of the hierarchical structure of the systems they target. The analysis method correctly estimated the throughput of several complex examples, with negligible runtimes.

TABLE III

RESULTS FOR THE SLACK MATCHING METHOD: BUFFER STAGES ADDED AND SPEEDUP OBTAINED

	Stages Added		Throughput (MHz)		Speedup
Example	MILP	Heuristic	before	after	(%)
CORDIC cond	8	8	90.9	167	84%
CORDIC	0	0	83.0	n/a	n/a
CRC	3	3	292	352	21%
DIFFEQ	0	0	18.3	n/a	n/a
GCD	0	0	49.0	n/a	n/a
raytracing	145	145	161	222	38%
MULT	8	8	38.7	167	332%

For slack matching, both an optimal MILP-based method, and a faster heuristic method are introduced. The heuristic method gives the same solution as the optimal MILP formulation for all of our examples.

Future work in this area includes extending the delay model to include min/max delays or delay distributions, and a study of the sensitivity of the slack matching algorithm to changes in the probability of conditionals.

REFERENCES

[1] N. Shenoy, "Retiming: theory and practice," *Integr. VLSI J.*, vol. 22, no. 1-2, pp. 1–21, 1997.
[2] T. E. Williams, M. Horowitz, R. L. Alverson, and T. S. Yang, "A self-timed chip for division," in *Advanced Research in VLSI*, P. Losleben, Ed. MIT Press, 1987, pp. 75–95.
[3] M. R. Greenstreet and K. Steiglitz, "Bubbles can make self-timed pipelines fast," *Journal of VLSI Signal Processing*, vol. 2, no. 3, pp. 139–148, Nov. 1990.
[4] A. M. Lines, "Pipelined asynchronous circuits," Master's thesis, California Institute of Technology, 1998.
[5] S. M. Burns, "Performance analysis and optimization of asynchronous circuits," Ph.D. dissertation, California Institute of Technology, 1991.
[6] E. G. Mercer and C. J. Myers, "Stochastic cycle period analysis in timed circuits," in *Proc. Int. Symp. on Circuits and Systems*, 2000, pp. 172–175.
[7] A. Xie and P. A. Beerel, "Performance analysis of asynchronous circuits and systems using stochastic timed Petri nets," in *Hardware Design and Petri Nets*, A. Yakovlev, L. Gomes, and L. Lavagno, Eds. Kluwer Academic Publishers, Mar. 2000, pp. 239–268.
[8] 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.
[9] 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)*, 2005, pp. 225–230.
[10] 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*, Apr. 1997, pp. 64–75.
[11] P. Pang and M. Greenstreet, "Self-timed meshes are faster than synchronous," in *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997, pp. 30–39.
[12] 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*, vol. 44, no. 11, pp. 1306–1317, Nov. 1995.
[13] S. Chakraborty, K. Yun, and D. Dill, "Timing analysis of asynchronous systems using time separation of events," *IEEE Trans. on Computer-Aided Design*, vol. 18, no. 8, pp. 1061–1076, Aug. 1999.
[14] 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.
[15] 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.
[16] P. Prakash and A. J. Martin, "Slack matching quasi delay-insensitive circuits," in *Proc. Int. Symp. on Asynchronous Circuits and Systems*, 2006.
[17] G. Venkataramani and S. C. Goldstein, "Leveraging protocol knowledge in slack matching," in *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 2006, pp. 724–729.
[18] T. E. Williams, "Self-timed rings and their application to division," Ph.D. dissertation, Stanford University, June 1991.
[19] 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.