Automated Microarchitectural Exploration for Achieving Throughput Targets in Pipelined Asynchronous Systems

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

***Abstract*—This paper presents a systematic approach for microarchitectural exploration in pipelined asynchronous systems, with the goal of achieving a specified throughput target while minimizing a given cost function (based on energy, area, etc.). The method includes a general framework that (i) allows for a rich extensible set of microarchitectural transformations for improving throughput; and (ii) can handle a variety of cost functions, such as area, energy, $E\tau^2$ and the energy-area product.**

In general, the space of transformations that can be applied to a given circuit is potentially infinite because an arbitrarily long sequence of transformations may be applicable. To compound the challenge, the value of the given cost function can change non-monotonically as successive transformations are applied (e.g., some transformations increase area, while others decrease area), thereby making it difficult to apply a typical branch-and-bound approach to prune the search space. Our method employs simple but effective heuristic search strategies (including greedy, lookahead, and breadth-first). A key contribution is to identify commutativity of certain transformations, thereby pruning the design space significantly. The approach was automated and applied to a number of examples. Various throughput targets were assumed: from 50% to 20x throughput improvement. In each example, the approach was successful in meeting the throughput target.

I. Introduction

This paper introduces an automated approach for bottleneck removal in pipelined asynchronous systems in order to increase throughput. Automated bottleneck removal consists of assessing the outcomes of applying each available micro-architectural transformation and then choosing a sequence of transformations that yield a system with the desired throughput. Our approach is to use tree-search methods to explore the set of possible solutions and find one that is low-cost, based on a given cost metric.

As in past work [7], [8], we make this problem more tractable by focusing 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 topologies. 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.

The inclusion of bottleneck removal in an automated design flow is important for reducing designer effort in creating high-performance systems. For non-trival systems, the space of possible solutions is too large for efficient manual search. Moreover, the search space of possible optimizations is quite complex to explore because the cost function may vary non-monotonically: a transformation that seems to reduce performance may actually be the first step on a path towards a more efficient system design. Since system-level timing issues can be quite subtle and complex in asynchronous design, the absence of a fast automated bottleneck removal tool means a designer typically must contend with time-consuming and error-prone manual optimization, which hinders the practicality of asynchronous design.

The contribution of this paper is a framework for automated exploration of the design space of alternative micro-architectures starting from a given micro-architecture. We evaluate this framework with a set of five simple but powerful transformations: stage coalescing, stage splitting, duplication, buffer insertion and loop pipelining. This list of possible micro-architectural transformations is not intended to be exhaustive, but instead demonstrates that our framework is flexible enough to use a variety of transformations and apply them correctly to produce a high-throughput implementation.

We implement automated bottleneck removal as a tree search in which each node represents a micro-architecture (*i.e.*, an implementation) with the same input-output behavior as the given micro-architecture to be optimized. A node is a solution if the throughput of the implementation meets the goal throughput given for the system. Our framework employs three different search methods for discovering a low-cost solution: exhaustive breadth-first search, greedy search, and lookahead search. Results show that lookahead search rapidly generates a solution that is close to optimal for a variety of different input examples.

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 each iteration time-consuming. Additionally, bottleneck removal is not fully automated.

Methods presented by Beerel *et al.* [2], Prakash *et al.* [20], and Smirnov *et al.* [1] remove bottlenecks by strategically adding buffer stages to the circuit. These approaches, however, focus only on buffer insertion (*i.e.*, "slack matching") and do not target other micro-architectural transformations in concert with the addition of buffers.

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, only apply to a limited set of specialized architectures (*e.g.,* rings, meshes, and 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.

In contrast, this paper provides an analytical approach that is fast, that can handle systems with choice, and that provides systematic detection and elimination of system-wide bottlenecks until a throughput target is reached. The remainder of the paper is organized as follows. Section II provides background on prior work on performance analysis and bottleneck identification. Section III then introduces the new automated method for microarchitectural exploration, and Section IV presents powerful enhancements to the efficiency of the method by pruning the search space. Section V presents experimental results, and Section VI gives conclusions.

II. Background

This paper uses past performance analysis work [23], [10], [15], [7] as well as past work on bottleneck identification [8]. This section provides the necessary background on these approaches because they are used as a starting point for the work presented in this paper.

A. Performance Analysis

This section reviews relevant prior work in performance analysis for pipelined systems, which forms the basis for bottleneck identification and elimination. The analysis approach is based on what we call *canopy graphs.*

1) Asynchronous Pipeline Stages: 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.

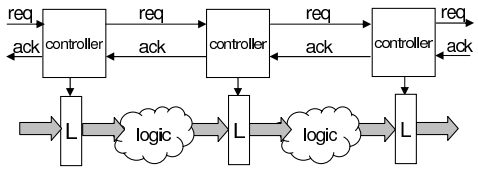

Fig. 1: A simple self-timed pipeline

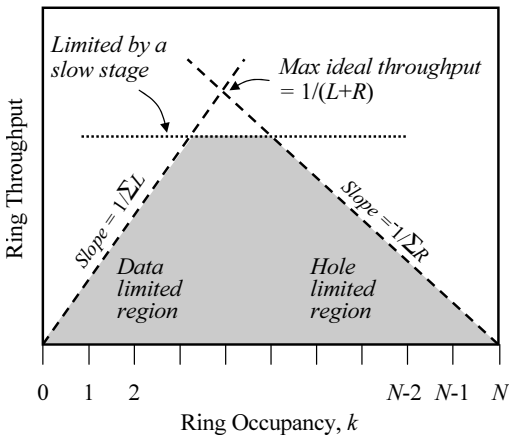

Fig. 2: The throughput of a ring as a function of the number of data items (a "canopy graph")

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 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) Analysis of Pipeline Rings: Williams and Horowitz [23] and by Greenstreet et al. [10], [9] were the first to introduce the use of canopy graphs for analyzing the throughput of pipelined asynchronous systems. 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) Canopy Graph Analysis: The above discussion presents canopy graph analysis as it was first used to apply to pipelined rings. Canopy graph analysis has since been extended for analysis of more complex circuits, and the definition of the canopy graph itself has been extended and formalized in [7], [8].

Canopy Graph Definition. As illustrated in Figure 3, 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. 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 the origin. As a result, such systems will always have exactly one forward segment and one or more reverse segments [8].

Application to Complex Systems. Though canopy graph analysis was initially applied only to ring structures with a constant occupancy [23], it has since been applied to more complex systems. 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. Further, [15] applied canopy graph analysis to pipelined systems that have fork-join parallelism.

Recent work by Gill *et al.* [7] extended canopy graph analysis to any pipelined system that is a hierarchical composition of pipeline stages using sequential, parallel, conditional, and iterative operators. The method exploits the hierarchical nature of circuits generated from high level code in order to achieve a fast runtime for estimating system performance. Specifically, it separately computes the canopy graph for subsystems in the hierarchical system, and then composes the graphs hierarchically.

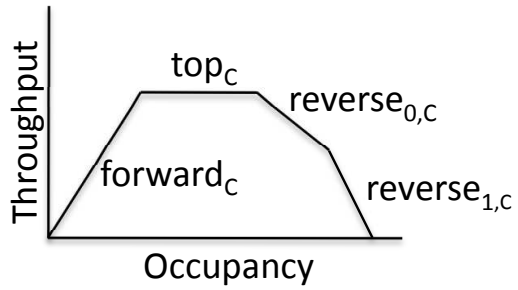

Fig. 3: Canopy graph C with four limiting segments [8]

Assumptions in Canopy Graph Analysis. The analysis methods reviewed here make the following assumptions [7]. Since our work builds on past canopy graph analysis, it also inherits these same 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.
- *Steady-State:* Canopy graph analysis applies when a system is in "steady state" operation. A system is said to be in steady state when the frequency of items entering and exiting remains constant.
- *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.

B. Bottleneck Analysis

Our recent work [8] used canopy graph analysis to identify which parts of a circuit limit the throughput of the system. The work in this paper leverages that research to create an automated method for exploring possible bottleneck alleviation solutions.

1) Bottleneck Identification: The method of [8] identifies bottlenecks by finding the segments of the canopy graph which limit throughput. The steps in that method are as follows: 1) Compute the system canopy graph 2) identify the limiting segments 3) compute an and-or formula that represents bottlenecks (i.e., groups of "culprits," with each group being a candidate for bottleneck alleviation).

Step 1 uses the canopy graph based performance analysis [7] to find the canopy graph of the entire system. Step 2 identifies which parts of the circuit have canopy graph segments that limit the canopy graph of the entire system. In particular,

TABLE I: TRICs and their applicability to each bottleneck type [8]

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

for each limiting segment of the system canopy graph, there are one or more nodes in the tree (which represents the hierarchical structure of the circuit) that contribute to that limiting segment. Step 3 generates a boolean formula that indicates which nodes or sets of nodes of the circuit tree should be modified in order to alleviate the bottleneck.

2) Bottleneck Classification: Previous work [8] classifies bottlenecks based on which segment of a canopy graph limits the throughput. We use the same terminology for classifying bottlenecks in this paper.

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 the 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.

3) Bottleneck Alleviation Methods: A Transformation for Increasing the Canopy Graph (TRIC) is a circuit transformation that can potentially alleviate a bottleneck [8]. Specifically, a TRIC is a circuit transformation that raises the throughput over some range of occupancies. TRICs can be categorized by the type of bottleneck they can alleviate. The TRICs listed in past work are summarized in Table I. Though this is not an exhaustive list of all possible performance-improving circuit transformations, the list shows that for each type of bottleneck there are several possible TRICs to choose from.

III. Automated Bottleneck Removal

This section describes the problem to be solved and our method of solution in detail. Section III-A defines the problem by specifying the inputs to and outputs from our solution framework. Section III-B describes the search space, and Section III-C presents methods for searching the space for a solution.

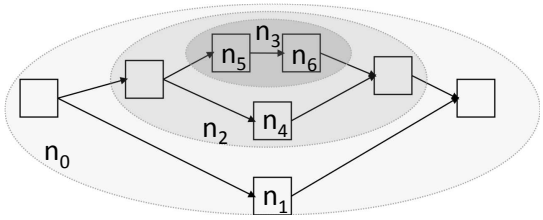

Fig. 4: Block level representation of a hierarchical system

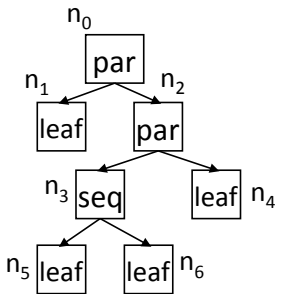

Fig. 5: A tree representing system of Figure 4

A. Problem Domain

Our method needs as inputs the following: a hierarchical representation of a circuit, a goal throughput, a set of TRICs, and a cost function. The desired output is a sequence of TRICs that, when applied to the circuit in the given order, yield a new circuit that meets or exceeds the throughput goal while minimizing the given cost function.

1) Input Circuit: Our method requires a hierarchical representation of the circuit to be optimized. The hierarchical representation consists of individual pipeline stages—each of which has a forward latency, a reverse latency, and a cycle time—and operators that combine those stages. The operators supported are sequential, parallel, conditional, and iterative composition operators. Figure 4 depicts a block diagram of a circuit to be analyzed and optimized. Each block represents a single pipeline stage. The hierarchical regions are marked in order to highlight the nested, hierarchical structure of this circuit.

Within our framework, we represent each hierarchical circuit as a tree in which the leaf nodes represent individual stages and all other nodes are either parallel, sequential, conditional, or loop operators. Figure 5 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.

2) Throughput Goal: The framework requires a throughput goal for the circuit. The throughput of the system as a whole will be computed using the method described in [7], and then compared with the goal throughput.

3) Circuit Transformations (TRICs): In order to make changes to the given circuit, our framework needs access to a set of TRICs (TRansformations for Increasing the Canopy graph) [7]. For use in our framework, a TRIC must specify

both how it changes the circuit tree and the criteria for its applicability.

a) TRIC Applicability to a Bottleneck Type: Each TRIC can alleviate some set of bottleneck types. The work in [7] gives three different categories of bottlenecks that can occur; these are given the labels Type I, Type II, and Type III. Specifically, if a TRIC increases the area under the forward segment of the canopy graph, that TRIC could potentially alleviate a Type I bottleneck. Similarly, TRICs that affect the top and reverse segments of the canopy graph can be applied to Type II and Type III bottlenecks respectively. Note that our framework does not determine bottleneck type each TRIC applies to, but instead requires that this information be part of the TRIC specification.

b) TRIC Circuit Changes: Each TRIC must specify which set of circuit node types it can be applied to. For example, the coalescing TRIC [8], which combines the functionality of two stages into one stage, can be applied only to two leaf nodes that are composed sequentially.

Each TRIC specifies its changes to the circuit in terms of changes to the circuit tree, and the resulting circuit must always be representable as a hierarchical circuit tree. Examples of possible changes that a TRIC can make to the tree include removing nodes, replacing nodes with nodes of a different type, and adding nodes to a sequential construct. TRICs are not allowed to generate any nodes with more than one parent, thereby preserving the tree structure of the system specification.

4) Cost Functions: Often, there are many different ways to modify a circuit to reach a throughput goal. Therefore, our framework requires a cost function that includes some metric other than throughput. Common cost metrics include the energy area product and Eτ^2. For a cost function to be compatible with our framework, it must be able to take the tree representing the circuit as input and output the total cost of the circuit. In addition, the cost function must be meaningful when applied to any node in the circuit tree. For example, applying the cost function to node n_3 in Figure 5 should give the cost of the sub-circuit including nodes n_3, n_5, and n_6.

5) Output: The framework outputs the set of TRICS to apply; this includes information about which nodes each TRIC applies to and the order in which to apply the TRICS. It also outputs the cost of the circuit for the given cost function, and the throughput prediction for the circuit.

B. Solution Space

The solution space of this problem is the set of circuit configurations that reach the goal throughput, with a secondary goal of maintaining a low cost function. In this section, we fully define the solution space by specifying the search tree and also giving two methods for pruning the search tree.

1) Search Tree Structure: The search space can be represented by a tree structure in which each node corresponds to a circuit with the same behavior as the original input circuit. As an example, Figure 6 shows an example search tree for the circuit in Figure 4. The root vertex of the tree, C_0 represents

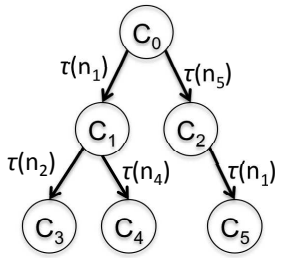

Fig. 6: A sample search tree

the original circuit specification input to the system. Each of its children represent the result of applying a TRIC to that original specification. Specifically, for every vertex C_i there is some set of transformations, $TRIC_i$, that potentially alleviate a bottleneck in the circuit.

An edge from vertex C_i to vertex C_j exists if there is some transformation in the set $TRIC_i$ that changes the circuit corresponding to C_i to the circuit corresponding to C_j. We use the notation $C_i \xrightarrow{\tau} C_j$ to denote that some TRIC, τ, when applied to the circuit represented at node C_i will yield the circuit at C_j. The weight of the edge is determined from the given cost function. Specifically, the weight of an edge from C_i to C_j is the change in the cost function between the two circuits.

A path through the tree is a series of TRICs starting at the root vertex C_0 and ending at some vertex C_i. We use the notation $C_0 \xrightarrow{p_i} C_i$ to indicate that some path p_i exists from node C_0 to node C_i. The tree consists of a set of vertices such that for every vertex there is some path from the root node to to that vertex. There is an edge from one vertex to another if there is a TRIC that could be applied to a bottleneck in the first circuit to produce the second. More formally, the set of vertices, V, and edges, E, can be defined as follows:

$$V = \{C_i \mid \exists p_i C_0 \xrightarrow{p_i} C_i\}$$
$$E = \{(C_i, C_j) \mid \exists \tau C_i \xrightarrow{\tau} C_j s.t. \tau \in TRIC_i\}$$

2) Terminating Conditions: Edges in the graph can have either positive or negative weights, depending on the specific TRIC applied and the cost function being used. As a result, the search space is non-monotonic, which makes it difficult to find heuristics for pruning the search tree. Further, the search space can be infinite because there can be arbitrarily long sequences of TRICs to consider in the absence of methods to effectively bound the search space.

Our framework currently uses two terminating search conditions: reaching some maximum depth and finding what we call a *prime solution.* Specifically, the setting the maximum depth places the restriction that path length from the root vertex C_0 to any graph vertex be less than d, that is $||p|| < d$.

The second terminating condition is finding a *prime* solution. Intuitively, a prime solution is the first solution on a given path that reaches the given throughput goal, g. This limits the vertex set V by placing a restriction on the path p that no other solution path exists that is a prefix of p. More

formally, C_i is a vertex in the search space iff $\forall p_x | C_0 \xrightarrow{p_x} C_x \xrightarrow{p'} C_i, tpt(C_x) < g$. That is, C_i is a reachable vertex in the search space only if all the vertices that lie along the path from C_0 to C_i *do not* meet the goal. Note that since our search space is non-monotonic, stopping the search at the prime solution can potentially prevent the search from finding a lower cost solution further past a prime solution. The trade-off, which is worth this potential loss in solution quality, is that using the prime solution prunes the search space to a more manageable, non-infinite size.

C. Solution Methods

1) Exhaustive Search: An exhaustive search of the solution space is quite time consuming. In an exhaustive search, every possible path is explored until reaching the terminating condition. For exhaustive search, we use both solution depth and finding the prime solution as terminating conditions, thereby pruning the search tree at some maximum depth and at every prime solution. Figure 7 is a graphical representation of this search. A search along a given branch terminates at the prime solutions, shown as shaded circles.

Pseudocode for a recursive implementation of this algorithm is shown in Figure 8. The exhaustive search method finds all the solutions within the search tree, and then the lowest cost solution that meets the throughput goal is reported as the final solution.

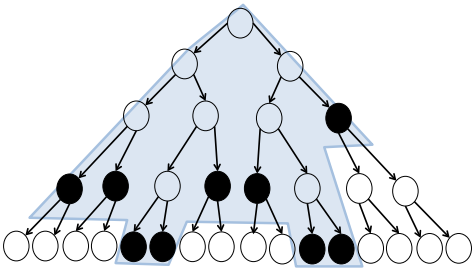

Fig. 7: Exhaustive search with prime solutions as the terminating condition

```
SolutionPath ExhaustiveSearch( TreeNode t, double goal )
    for each t.getChild
        if child.tpt >= goal
                if lowCost > child.tpt
                        lowCostNode = child
        else
                ExhaustiveSearch( child, goal )
    return lowCostNode.getPath
```

Fig. 8: Pseudocode for exhaustive search strategy

2) Greedy Search: A greedy search of the solution space prioritizes the vertices of the search tree based on which of the next vertices has the lowest heuristic function. If the node that has the lowest heuristic function is a solution, (*i.e.* the throughput is greater than the goal throughout) the search ends and the lowest cost solution found is returned. Figure 9 is a graphical representation of a greedy search. Lightly shaded nodes are analyzed but not taken while dark nodes are the nodes along the chosen path. The final node is the prime solution along that path.

Pseudocode for the greedy algorithm is shown in Figure 10. Greedy search can be faster than an exhaustive search because fewer vertices are visited. However, the solution obtained may not be the lowest cost solution. The greedy search is also bounded by prime solutions: it continues until it finds the first solution along a path.

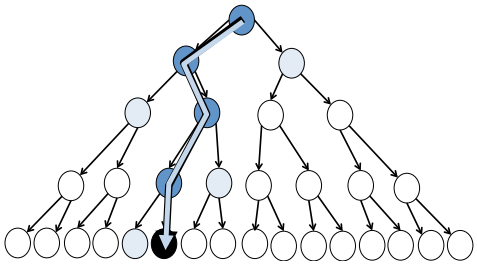

Fig. 9: Greedy search evaluates a smaller number of nodes

```
SolutionPath greedy( TreeNode t, double goal )
  for each t.getChild
        if child.heuristic < lowHeuristic
                lowHeuristicNode = child
        if child.tpt >= goal AND child.cost < lowCost
                lowCostNode = child
  if lowHeuristicNode.tpt >= goal
        return lowCostNode.SolutionPath
  else
        return greedy( lowHeuristNode, goal )
```

Fig. 10: Pseudocode for greedy search strategy

3) Lookahead Search: A lookahead search prioritizes vertices based on cost information from nodes some depth, d, down from each child of the current vertex. Thus, greedy search can be called a special case of lookahead search with a lookahead depth of 0 (*i.e.*, only immediate children are evaluated to determine the next step in the descent).

Figure 11 is a graphical representation of three steps in a lookahead-2 search. The dark shaded nodes represent nodes with the minimum value for the cost function at a given depth. The algorithm takes one step in the direction of the shaded node, and evaluates another set of nodes one level deeper in that direction. Pseudocode for a recursive implementation of this algorithm is shown in Figure 12. During the search, algorithm keeps track of the minimum cost solution that it has found at any point in the search space. When it finally terminates, it will return this minimum cost solution.

A lookahead search can produce a more optimal solution than a greedy search while still visiting a smaller number of nodes than an exhaustive search. Therefore, it can offer a speed advantage over exhaustive search while still returning a result that is close to optimal.

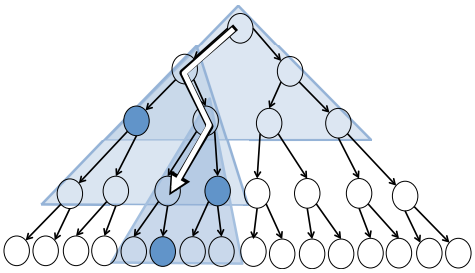

Fig. 11: Three steps in a lookahead-2 search

```
SolutionPath LookAhead( TreeNode t, double goal, double depth )
  for each t.getChild
      expand Child lookDistance nodes ahead
      record minCostSolution
      record minHeuristicNode
  if( minHeuristicNode.tpt >= goal )
        return minCostSolution
  else
        return Lookahead( minHeuristicNode, goal, depth)
```

Fig. 12: Pseudocode for lookahead search strategy

IV. Advanced Tree Pruning Approaches

This section details two further enhancements to our search techniques. The first involves recognizing which sets of TRICs are commutative (*i.e.* the order of the operations does not matter.) The second is identifying when one TRIC effectively undoes the action of a previously applied TRIC; we refer to these TRICs as reciprocals. Both enable additional pruning of the search space to eliminate redundant search paths, which leads to faster optimization runtimes.

A. Commutative Transformations

1) Definition of Commutativity for TRICs: Two TRICs are commutative if the order in which they are applied does not affect either the structure or the performance of the resulting circuit. Consider two TRICs τ_a and τ_b which transform graph vertices such that $C_0 \overset{\tau_a\tau_b}{\rightarrow} C_x$ and $C_0 \overset{\tau_b\tau_a}{\rightarrow} C_y$. If C_x and C_y are equivalent for all possible initial nodes C_0, then the two TRICs are commutative.

2) Commutativity of the Bag of TRICs: The TRICs currently implemented in our tool are coalescing, stage splitting, duplication, buffer insertion, and loop unrolling. When determining if the TRICs are commutative, we consider the set of circuit hierarchy nodes that is affected by the TRIC. TRICs that affect individual leaf nodes in the circuit hierarchy will be commutative, since their effects on the circuit are strictly local.

Loop unrolling and duplication, however, have more global effects and are potentially not commutative. Both create copies of some set of circuit nodes, thereby resulting in a change to a large part of the circuit and limiting commutativity of subsequent TRICs. For loop unrolling, we address this issue by marking the circuit nodes that have been affected by the loop unrolling. When subsequent transformations are applied, the marking enables the framework to judge that the transformation taking place after the loop unrolling is distinct from one taking place before the loop unrolling.

In duplication, we address this issue by modifying the set of TRICs that can be applied after duplication in order to restore commutativity of the operation. In particular, once a portion of the circuit has been duplicated, all TRICs that are applied to one duplicated circuit component must also be applied to its matching duplicated component. This is not likely to result in loss of solution quality because applying a TRIC to only one half of a duplicated node will not improve overall throughput.

This conclusion is reached based on knowledge of the bottleneck identification method used by the framework, from the work of [8]. Specifically, after duplication takes place, the canopy graphs (*i.e.*the throughput plotted vs. the occupancy) of the two duplicated paths will henceforth be combined using the parallel operator. The parallel operator prescribes that the canopy graphs of the two components are intersected with each other. More formally, for duplicated components the combined canopy graph is the minimum of the canopy graph of each component: $cg(k) = min(cg_0(k), cg_1(k))$. If two different TRICs, τ_a and τ_b are applied to the duplicated nodes respectively, then the new canopy graph becomes $cg(k) = min(cg_a(k), cg_b(k))$. On the other hand, if the same TRIC, τ_a, is applied to both of the duplicated components, the resulting combined canopy graph is simply $cg_a(k)$; similarly if the same TRIC, τ_b, is applied to both of the duplicated components, the resulting canopy graph is equal to $cg_b(k)$. Since it is algebraically true that $cg_a(k) \geq min(cg_a(k), cg_b(k))$ and $cg_b(k) \geq min(cg_a(k), cg_b(k))$, we conclude that the throughput obtained by applying either τ_a or τ_b to both sides will be equal to or greater than that obtained by applying τ_a to one of the duplicated components and τ_b to the other.

For this reason, we preserve the commutativity of duplication with other operations by enforcing that TRICs applied to one duplicated component must always be applied in kind to the other. Preserving commutativity allows the search space to be more effectively pruned and therefore further reduces the execution time of the algorithm. While subtle interactions between throughput and the cost function, which are not covered by this explanation of commutativity for duplication, may exist, comparing our results before and after pruning in results Section V-B2 indicates that none of the best solutions are lost after applying the as a pruning heuristic.

3) Exploiting Commutativity for Runtime Improvement: Figure 13 shows a search tree composed of the permutations of three TRICs: τ_a, τ_b, and τ_c. If the three operations are commutative, the grayed nodes are removed from the search, assuming a left-to-right traversal order for the tree.

We implement this pruning by creating a canonical representation for each path already explored. The canonical representation is a lexicographical reordering of the search path such that two paths that are equivalent under commutativity will also have the same canonical paths. Pseudocode for our algorithm using canonical paths representations to prune the search space is shown in Figure 14. After the tool determines a

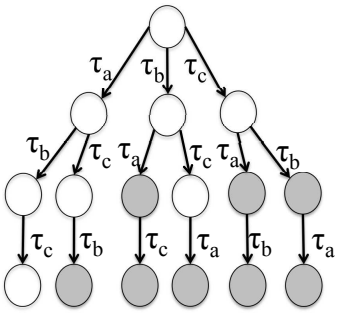

Fig. 13: Recognizing commutative operations to prune the search space

```
CanonicalTest( CompositionNode node )
    TRIC[] choices = bottleneckAnalaysis( node )
    for each choice in choices
        path += choice
          canonical_path = lexicographical_order( path )
          if( !previous_paths.contains ( canonical_path ) )
                previous_paths.add( canonical_path )
                TRICqueue.add( choice )
```

Fig. 14: Exploiting commutativity to prune the tree

set of possible next TRICs, it creates a canonical representation for each one and adds the TRIC to the queue of TRICs to be applied only if the canonical representation is not in the set of paths already explored.

B. Reciprocal Transformations

1) Definition of Reciprocal for TRICs: Applying one TRIC can potentially be the reciprocal of another, *i.e.*, applying it will undo the change made by an earlier TRIC such that the resulting circuit has the same structure and performance characteristics as the original circuit. More formally, consider two TRICs τ_a and τ_b which transform nodes such that $C_0 \stackrel{\tau_a \tau_b}{\longrightarrow} C_x$. If C_x always equals C_0 for all possible initial vertices C_0, then τ_b is the reciprocal of τ_a.

2) TRICs as Reciprocal Transoformations: Of the TRICs used in our tool, there are two that are reciprocals of each other. Specifically, coalescing and stage splitting, when applied successively to the same pipeline stage, will net no resulting change to the circuit. Although our current set of TRICs does not include any other reciprocal transforms, introducing new TRICs (*e.g.* transformations to increase or decrease the level of parallelism) will likely create new pairs of reciprocal transformations that, when taken together, reduce to an identity transformation (i.e., cancel each other out).

3) Avoiding Identity Transforms to Improve Runtime: Figure 15 shows a search tree pruned to avoid identity transforms. In this example, τ_a and τ_b together form an identity transform. As a result, the tree vertex labeled as C_x is equivalent to the vertex labeled at C_0, and the shaded vertices will not be visited during a search. Notice that the path consisting of the series of TRICs τ_a, τ_b, τ_c is no longer explored. This will not result in the loss of a possible solution, however, because the path consisting of only τ_c is equivalent.

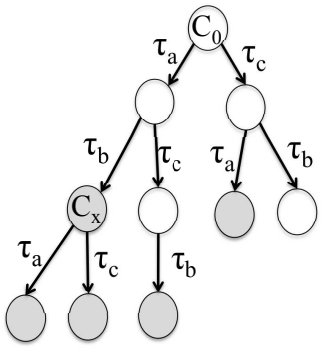

Fig. 15: Pruning search space to avoid identity transforms

We implement this optimization by flagging nodes within the hierarchical circuit that have been coalesced or split by past TRICs. These flags are then checked when determining the set of possible TRICs at a search node vertex. For example, if our tool lists coalescing among the possible optimizing transformations on a circuit node that has already been split, when that suggestion is checked against the flags it will be removed and not explored during the search.

V. Results

A. Experimental Setup

1) Circuit Examples: We test our approach for identifying and alleviating bottlenecks on five examples of varied topologies and sizes. For each example, Table II indicates which type of circuit constructs make up the specification, the initial throughput as analyzed by the method of [7], and the number of nodes in the hierarchical representation of the circuit.

2) TRICs Used: Previous work [8] gives examples of several different TRICS that can be used to alleviate bottlenecks. For the experiments given here, we used a set of five TRICs: coalescing, stage splitting, duplication with wagging, buffer

TABLE II: Information about examples used

Name	Circuit Type	Nodes	Throughput
DIFFEQ	parallel, sequential, and loop	22	0.018
MULT	parallel	10	0.038
CORDIC	parallel and sequential	39	0.083
CRC	conditional and sequential	23	0.029
JPEG	Loop, conditional, and sequential	40	0.00047

TABLE III: TRIC applicability

TRIC	Bottleneck Type	Node Type
Coalescing	Type I	Leaf node
Stage Splitting	Type II & III	Leaf node
Duplication	Type II & III	Any node
Buffer Insertion	Type III	Sequential node
Loop Unrolling	Type II & III	Loop nodes

TABLE IV: Comparison between search methods

	DIFFEQ			MULT			CORDIC			CRC			JPEG		
Search Method	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)
Exhaustive	0.036	20419	4.95	0.151	2151	882.81	0.250	1392	> 7200	0.5	196	> 7200	0.015	420445	> 7200
Greedy	0.036	21931	0.07	0.103	4485	0.07	0.167	2952	0.05	0.5	200	0.07	0.013	437606	0.20
Lookahead 1	0.036	21175	0.06	0.154	2197	0.22	0.182	2511	0.14	0.5	196	0.58	0.015	420445	15.29
Lookahead 3	0.036	21175	2.45	0.151	2151	15.65	0.250	1392	28.17	0.5	196	506.70	0.015	420445	495.35

TABLE V: Speed improvement with additional tree pruning

	DIFFEQ			MULT			CORDIC			CRC			JPEG		
Search Method	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)	Tpt	Cost	Time (s)
Exhaustive	0.036	20419	0.99	0.151	2151	21.47	0.250	1392	66.87	0.5	196	17.57	0.015	420445	2.70
Lookahead 1	0.036	21175	0.06	0.154	2197	0.20	0.182	2511	0.14	0.5	196	0.43	0.015	320445	0.39
Lookahead 3	0.036	21175	0.28	0.151	2151	4.22	0.250	1392	11.69	0.5	196	5.11	0.015	420445	1.16

insertion, and loop unrolling. Table III shows the bottleneck types and circuit node types that each of the TRICs applies to. While this is not an exhaustive set of possible circuit transformations, it is sufficient in that each type of bottleneck is handled by at least one of the TRICs.

3) Delay Models, Area and Energy: For these experiments, we use estimates for delay, area, and energy for each stage based on an estimate of its complexity. As in previous work [8], the delay model estimates the forward and reverse latency of a buffer stage at 1 ns, for a cycle time of 2 ns. Stages with logic have longer forward latencies, based on a rough estimate of the complexity of the logic (*e.g.* one full adder has twice the forward delay of the buffer stage). Reverse latencies are not affected by the presence of logic. Similarly, we use a normalized area of one for each buffer stage and an energy of two for each buffer stage.

B. Experimental Results

1) Comparing solution methods.: The running time and solution quality for any example varies based on the type of search method used. Table IV compares the the runtime of the basic method described in Section III-A on a 2.1 GHz Intel Core 2 Duo machine. For each example, the throughput goal is low enough to obtain meaningful results from the exhaustive method while being high enough to highlight the differences between the methods. CORDIC, CRC, and DIFFEQ have a 50% throughput improvement goal while MULT has a 2x throughput goal and JPEG has a 20x throughput goal. This experiment uses the cost function of energy over the square of throughput, which is equivalent to the common $E\tau^2$ metric. It also uses $E\tau^2$ as its search heuristic during the greedy and lookahead searches.

Table IV shows results for each solution method. The lookahead method appears twice with two different lookahead depths, 1 and 3. For every example, the exhaustive method is the most time consuming while the greedy method is the least time consuming. In addition, the exhaustive method gives a lower cost solution than the greedy solution for every example. Note that for some examples, the exhaustive method did not fully finish after running for over 2 hours. The result shown is the best out of the partial set of results.

Based on these results, the lookahead 1 search seems to be the best tradeoff between runtime and solution quality. It found the best solution possible for CRC and JPEG and is within 4% of the best solution for DIFFEQ and MULT. On CORDIC, however, it yields a cost that is 80% higher than the best solution. In addition, as the throughput goals get higher, the error in the lookahead 1 search will likely increase as well.

2) Runtime Improvement with Tree Pruning: Table V shows the runtimes with the addition of the pruning methods described in Section IV. Greedy is not included because it will not benefit much from additional tree pruning. All the other search methods find the same quality of solutions as those without pruning, but with at a much faster runtime. The quality of the results, in terms of throughput and cost, indicates that the pruning we employ does not eliminate good solutions from the search space.

Lookahead 1 search has the least amount of runtime benefit from the use of additional pruning, since it was already searching a much smaller number of vertices than the other methods. Both lookahead 3 and the exhaustive search method show large improvement in the execution time with additional tree pruning. JPEG shows the most improvement, with the exhaustive and lookahead 3 searches improving by 2662x and 428x respectively. Although DIFFEQ shows the least amount of improvement, the speed improvement is still quite high: exhaustive and lookahead 3 search runtimes improve by 5x and 8.7x respectively.

Based on these results, the search method with the best tradeoff between execution time and solution quality is lookahead 3. In all but one of the examples, it reached a solution with the same cost as the exhaustive solution. In DIFFEQ, it was just 3.7% higher. For these reasons, lookahead 3 is the search we use in the following experiments to gather all further results.

3) Varying Cost Metrics and Goals: Based on the execution time and solution quality of the examples in the previous sections, lookahead 3 is a efficient method for finding a high-

TABLE VI: Results using different cost functions and goal throughputs

	Initial & Target Throughput			Various Cost Functions									
				$E\tau^2$		$E\tau \cdot a$		$Energy$		$area$		$Energy \cdot area$	
	Initial tpt	Increase (X)	Goal tpt	Final tpt	Cost	Final tpt	Cost	Final tpt	Cost	Final tpt	Cost	Final tpt	Cost
DIFFEQ	0.018	1.500	0.027	0.036	21175	0.031	17160	0.030	24	0.030	28	0.031	528
	0.018	2.000	0.036	0.036	21175	0.038	26640	0.036	26	0.038	36	0.038	1008
MULT	0.038	2.000	0.077	0.151	2151	0.100	20700	0.077	44	0.077	44	0.077	1936
	0.038	5.000	0.192	0.414	327	0.400	21725	0.200	50	0.200	100	0.200	4704
CORDIC	0.083	1.500	0.125	0.250	1392	0.128	57215	0.143	81	0.128	88	0.128	7304
	0.083	2.000	0.167	0.333	801	0.274	61136	0.186	81	0.167	152	0.186	12636
	0.083	3.000	0.250	0.500	364	0.404	55259	0.333	86	0.255	204	0.255	17748
CRC	0.286	1.500	0.429	0.500	196	0.500	7056	0.500	49	0.500	144	0.476	3360
	0.286	2.000	0.571	0.571	266	0.571	19488	0.571	82	0.571	220	0.571	9676
JPEG	4.73E-04	20.000	9.47E-03	1.53E-02	420445	1.28E-02	1044108	1.28E-02	97	1.28E-02	136	1.28E-02	13386
	4.73E-04	40.000	1.89E-02	2.22E-02	212625	2.22E-02	1096200	2.22E-02	105	2.22E-02	232	2.22E-02	24360

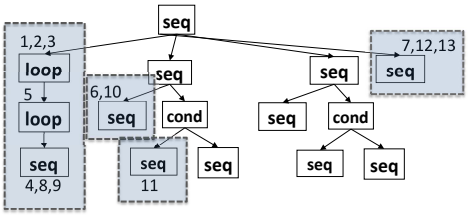

Fig. 16: JPEG hierarchy with detected bottlenecks

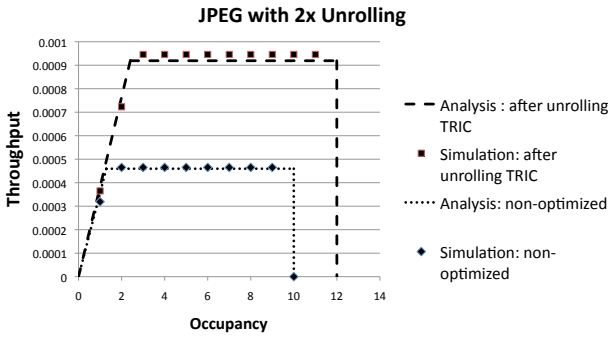

Fig. 17: The performance of the JPEG benchmark after loop unrolling

quality solution, if it is not necessary that the solution be completely optimal. Table VI therefore uses the lookahead 3 search method to find the solution for two different cost functions. Each example is tested for two different throughputs: DIFFEQ, CORDIC, and CRC have goals of 50% and 2x throughput improvement while MULT has goals of 2x and 5x throughput improvement and JPEG has goals of 20x and 40x.

For each example, we ran the search method with five different cost metrics. The search heuristic used for each cost was simply the cost over throughput. In every case, our system was able to find a solution that meets the throughput goal. This indicates the ability of our system to work with a variety of different cost functions.

4) JPEG Case Study: A more detailed look at one example, the JPEG encoder [11], shows the steps in the optimization process. This benchmark uses 8-bit arithmetic and is set to convert an image of 8 pixels by 8 pixels. Figure 16 shows a diagram of the hierarchy of the JPEG encoder. It includes nested loops, conditionals, and serial composition. The JPEG benchmark was also modeled at the gate level in Veriog; the tool for performance analysis can also output a Verilog model of the hierarchical system.

Because the main bottleneck is the loop, one of the first steps suggested by the bottleneck identification method is loop unrolling. If the inner loop is unrolled one time (*i.e.* the body of the inner loop is repeated twice) the performance is expected to double. Figure 17 shows the analyzed performance of the system after applying the loop unrolling TRIC one time. For comparison, the non-optimized performance is displayed again in the same chart. Verilog simulation confirms that the throughput indeed reaches to twice the non-optimized throughput. The throughput predicted by the analysis method is 9.19E-4 and the throughput found through simulation is 9.46E-4. This is a percent error of about 2.9%. This may be attributed to errors in modeling the circuit or to a possible over-estimate of the overheads associated with a pipeline loop.

To provide a challenge for the optimization framework, we next give it a goal throughput of 40x the initial throughput, which would require it to attain the throughput of 1.84E-2. The optimization method also needs a cost function to minimize during its search; in this case we use $E\tau^2$ as the cost function. The optimization method performs a total of 13 steps in reaching this goal, and in alleviates bottlenecks throughout the entire system. Figure 16 indicates the areas of the circuit in which the optimization method applies TRICs. The areas are numbered to show the order in which the TRICs were applied. In particular, the loop receives much attention, with the first three TRICs going towards loop unrolling. Within the loop, splitting of a slow stage also takes place. After the fifth TRIC, the loop ceases to be the bottleneck and the attention is turned to the first conditional in the system, which needs to be slack matched. Finally, a very slow, large multiplication at the end of the sequence of stages becomes the bottleneck, so it is split and pipelined into a series of faster stages in using a sequence of TRICs.

The final throughput of the system after all this optimization

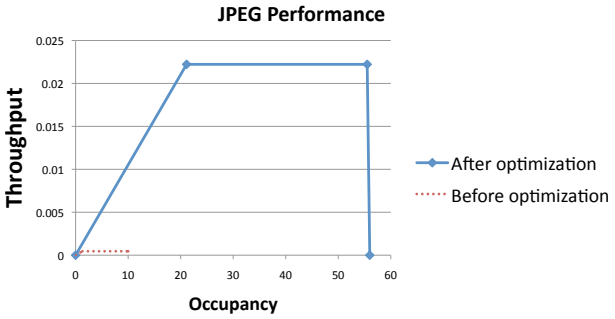

Fig. 18: JPEG performance after 40x throughput improvement

takes place is shown in Figure 18. The final performance achieved is 2.2E-2. Though this slightly overshoots the performance goal, the optimization system found that this circuit is the best one for meeting the goal throughput of 1.84E-2 while maintaining a low energy cost. For comparison, the results before optimization are also shown on the same chart; the reader must look closely as the initial throughput was much lower.

VI. Conclusion

This paper presented a framework for iterative bottleneck removal. Our results show our framework can identify a series of circuit transformations that reach the desired throughput goal while maintaining a low value for the given cost function. The framework was able to achieve between 50% and 20x throughput improvement when using the $E\tau^2$ cost metric on five example circuits without advanced pruning techniques. With the addition of advanced pruning and searching techniques, it was able to achieve between 2x and 40x throughput improvement on all example circuits. When comparing the different search methods, the results also indicate that lookahead 3 search offers a good trade-off between algorithm run time and solution quality compared to greedy search and exhaustive search. The framework can handle a variety of different cost metrics, and our results show successful optimization for five different cost functions: $E\tau^2$, energy·area·, energy alone, area alone, and the energy-area product. All execution times using advanced pruning techniques were less than two minutes for all examples over all cost functions and throughput goals. The results as a whole indicate that our framework is capable of quickly providing high-quality solutions for minimizing various cost functions while meeting a throughput goal.

References

[1] A. T. Alexander Smirnov. Heuristic based throughput analysis and optimization of asynchronous pipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, May 2009.

[2] 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.

[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 and M. Singh. Bottleneck analysis and alleviation in pipelined systems: A fast hierarchical approach. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, May 2009.

[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. Concurrency-enhancing transformations for asynchronous behavioral specifications. Master's thesis, University of North Carolina at Chapel Hill, 2007.

[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.