Automated Synthesis for Asynchronous FPGAs

Song Peng, David Fang, John Teifel,[*] and Rajit Manohar
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853, USA
{speng,fang,teifel,rajit}@csl.cornell.edu

ABSTRACT

We present an automatic logic synthesis method targeted for high-performance asynchronous FPGA (AFPGA) architectures. Our method transforms sequential programs as well as high-level descriptions of asynchronous circuits into fine-grain asynchronous process netlists suitable for an AFPGA. The resulting circuits are inherently pipelined, and can be physically mapped onto our AFPGA with standard partitioning and place-and-route algorithms. For a wide variety of benchmarks, our automatic synthesis method not only yields comparable logic densities and performance to those achieved by hand placement, but also attains a throughput close to the peak performance of the FPGA.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—*Program Synthesis*

General Terms

Design

Keywords

Asynchronous circuits, automated synthesis, programmable logic

1. INTRODUCTION

FPGAs are a popular technology for system prototyping due to their short turnaround time and low cost. The generality of the logic that can be implemented on an FPGA results in higher area, higher power, and lower performance compared to an application-specific design, but at greatly reduced cost. Interconnect delays in FPGAs can be large, and the wide variation in interconnect lengths possible after the mapping process complicates the design of a high-throughput FPGA. This problem is exacerbated by technology scaling, which causes the ratio of wire to gate delay to increase.

[*]John Teifel is now with the Advanced Microelectronics Department at Sandia National Laboratories.

Asynchronous FPGA design was proposed as a way to combat the problems of clock distribution in FPGAs, as well as to exploit the data-dependent nature of circuit delays by not having to time the circuit using the worst-case delay path [7]. However, mapping asynchronous logic gates into a fixed set of standard gates from the proposed FPGA architecture was a very complex task. Some challenging requirements imposed by the FPGA architecture that the CAD tools had to deal with included: (i) the isochronic fork assumption, which corresponded to ensuring a bound on the relative interconnect delay between the endpoints of the fanout of a gate; (ii) the hazard-free decomposition requirement, where a complex logic gate had to be mapped to the fixed set of gates present in the FPGA without creating any switching hazards.

Recently, we have developed a high-performance asynchronous FPGA (AFPGA) architecture that is designed using different principles [19, 20]. Instead of mapping a collection of asynchronous gates to the FPGA, we map the *functionality* of the asynchronous logic to the FPGA directly. Essentially, our AFPGA corresponds to an array of single-bit programmable asynchronous pipeline stages, where the operation of the pipeline stages can be configured to implement various functions. The AFPGA also has a pipelined interconnect, which results in high-throughput operation even in the presence of long routes.

In this paper we propose a complete CAD flow for automated synthesis of asynchronous computations onto our pipelined AFPGA architecture. The synthesized asynchronous computations are then placed and routed using `vpr` [2], and we show that the combination of our AFPGA architecture and synthesis flow results in high-performance implementations of several asynchronous benchmark circuits. Since a subset of our asynchronous design language includes sequential programs, we can also automatically translate sequential programs to pipelined, high-performance implementations. To our knowledge, this is the first complete design flow for rapid prototyping of pipelined asynchronous circuits with an asynchronous FPGA.

Asynchronous Pipelines. Asynchronous systems are designed as a collection of concurrent hardware processes that exchange messages with each other through communication ports. These messages consist of atomic data items called *tokens*. Asynchronous pipelines are constructed by connecting these ports to each other using channels, and each channel is allowed only one sender and one receiver.

Since there is no clock in asynchronous design, processes use handshake protocols to send and receive tokens in channels. Most of the channels in our asynchronous FPGA use three wires, two data wires with dual-rail encoding and one acknowledge wire, and implement a 4-phase handshake protocol to prevent data from being overrun or sampled more than once [20]. The cycle time of a pipeline stage is the time required to complete one four-phase handshake. Throughput, the inverse of the cycle time, is the rate at which tokens enter and exit the pipeline.

Logical and physical pipelining are two separate concepts in asynchronous pipelines. A new physical pipeline stage is created by adding a circuit-level pipeline stage; a logical pipeline stage requires the insertion of a data token. Sometimes inserting a physical pipeline stage can cause an asynchronous circuit to malfunction. When an asynchronous circuit operates correctly even if we change the degree of physical pipelining, it is said to be *slack elastic* [15]. Our AFPGA is implemented with slack-elastic pipelines. The performance of an asynchronous design often depends on the amount of physical pipelining in the system.

Slack-elasticity allows a designer to locally add pipelining anywhere in the system without having to "retime" the entire system to preserve functional correctness, as is required in any non-trivial synchronous design. Slack-elasticity greatly simplifies logic synthesis and physical mapping because it allows each channel to be routed through an arbitrary number of pipeline stages in the interconnect without causing the circuit to misbehave.

Asynchronous Dataflow Computation. Logic computations in asynchronous pipelines behave like fine-grained, static, data-driven dataflow systems [4], where a token traveling through an asynchronous pipeline explicitly indicates the presence or absence of data. In this model, token values required by multiple destinations must be explicitly copied. To simplify our AFPGA architecture, we restrict our computation model to deterministic dataflow computations, i.e. computations without arbitration. Such computations are automatically slack-elastic [15]. Any deterministic asynchronous dataflow graph can be built from the fundamental dataflow nodes shown in Figure 1, and their functionality is summarized as follows:

- **Copy:** duplicates tokens to n receivers.
- **Function:** computes arbitrary functions of n input variables. It waits until all input tokens have arrived and then generates a token with computed value at its output channel.
- **Merge:** performs a two-way controlled token merge and allows tokens to be conditionally read on channels. It receives a control token from channel G. If the control token has zero value, it reads a data token from channel A, otherwise it reads a data token from channel B. Finally, the data token is sent on channel Z.
- **Split:** performs a two-way controlled token split and allows tokens to be conditionally sent on different channels. It receives a control token on channel G and a data token on channel A. If the control token has zero value, it sends the data token onto channel Y, otherwise it sends that data token onto channel Z.
- **Sink:** consumes the input data token unconditionally.
- **Source:** generates new data tokens with given constant value. A new data token won't be produced until the old one is consumed by the consumer.
- **Initializer:** upon the AFPGA's power-up, it resets with an input data token with given constant value. During normal operation, it operates like a copy node.

Our AFPGA implements bit-level asynchronous dataflow nodes. By building multi-bit dataflow nodes from these bit-level FPGA nodes, we can prototype arbitrary asynchronous computations with an AFPGA.

The asynchronous FPGA design [20] uses a standard "island-style" FPGA architecture as shown in Figure 2, which is composed of logic blocks surrounded by programmable interconnect tracks. Each logic block has four inputs and four outputs, and they are equally distributed on the north, east, south and west edges. The routing tracks are dual-rail encoded and go through switch boxes.

Each logic block contains a function unit, a conditional unit, two output copy units, three local source units and a local sink unit (Figure 3). In other words, a function node, a merge or split node, two copy nodes, three source nodes and a sink node can be mapped onto one physical logic block at the same time, without any resource conflict. The function block includes a 4-LUT, and the design resembles the logic block in a Xilinx Virtex™ series FPGA [9]. In addition, the function block contains special support for early-out carry generation that can lead to improved performance in an asynchronous design [20]. Carry chains can be routed using the normal interconnect (the *normal* carry chain) or using low latency carry channels that run vertically south-to-north between adjacent vertical logic blocks (the *fast* carry chain).

With this asynchronous FPGA architecture, the automated synthesis problem is to translate an asynchronous computation into bit-level dataflow blocks, and place-and-route the resulting design onto the AFPGA architecture.

The organization of this paper is as follows: Section 2 discusses the automated synthesis method for the AFPGA in detail; Section 3 presents experimental results for the synthesis flow on a number of benchmarks; Section 4 reviews the related work; Finally, we summarize the paper and present directions for future investigations in Section 5.

2. AUTOMATED SYNTHESIS

The input to our synthesis is a program written in CHP notation. CHP is a hardware description language that is widely used for asynchronous design, and is based on Hoare's CSP language [16]. Apart from a sequential programming notation, the language also includes communication primitives (send and receive) for message-passing between concurrent processes. In addition to this input, our synthesis flow also takes in a configuration file that specifies various parameters. The result of the synthesis is a bit-level dataflow implementation that is then placed and routed using `vpr` [2]. Figure 4 depicts the major parts of the synthesis flow that are described in detail below.

2.1 Canonical CHP Decomposition

A broad class of asynchronous circuits can be described abstractly as canonical CHP programs [21]. We analyze the control flow of a

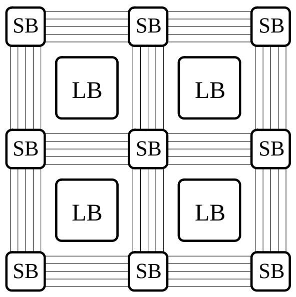


Figure 2: Asynchronous FPGA island-style architecture

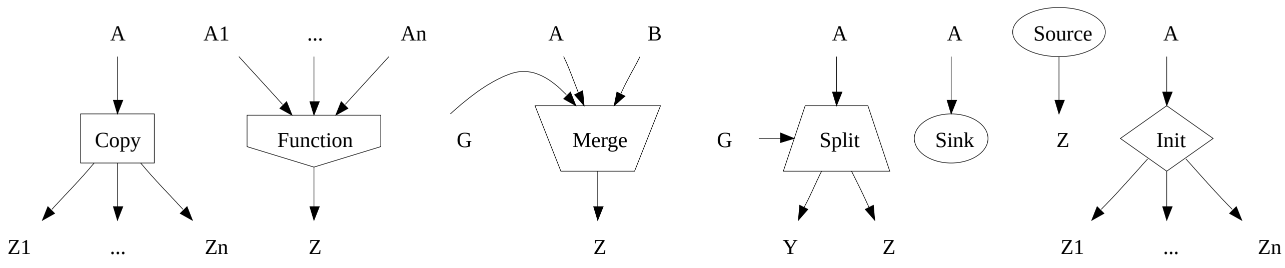


Figure 1: Dataflow graph nodes.

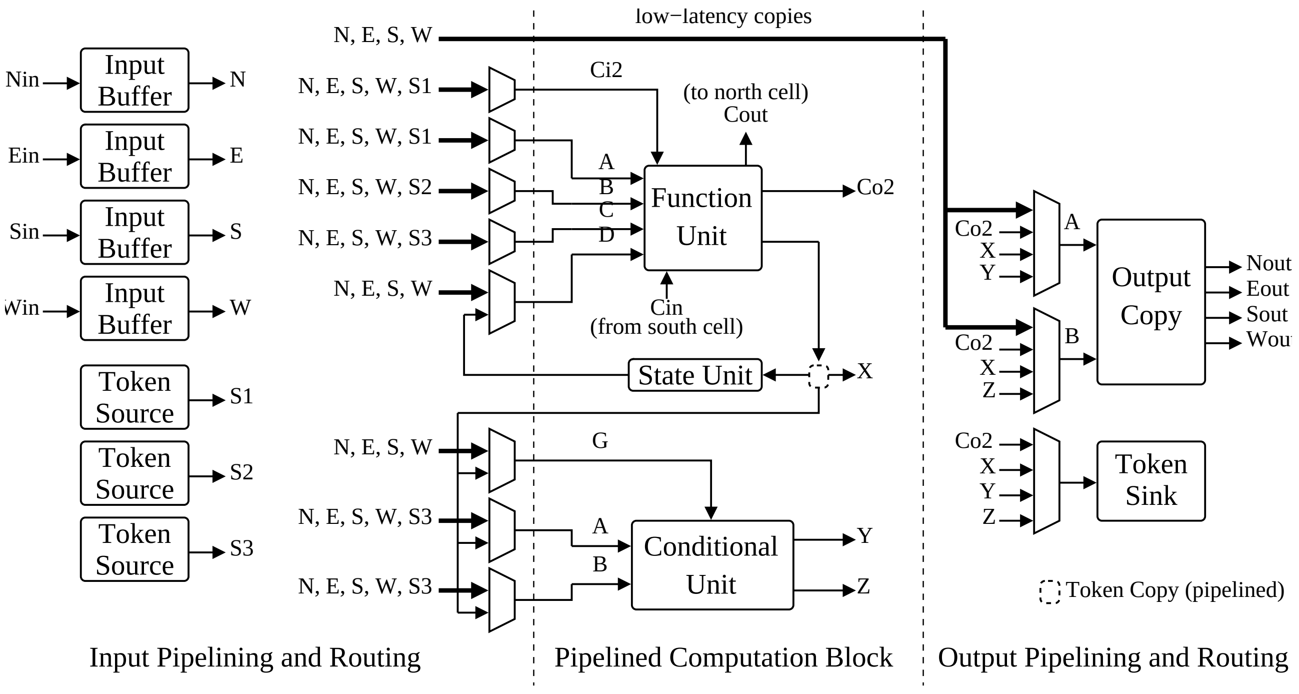


Figure 3: Pipelined asynchronous FPGA logic block.

sequential canonical CHP program using Static Token (ST) form, an intermediate representation which is an extension of the Static Single Information (SSI) form used for compiler analysis [1]. The key property of ST form is that the control-flow condition that determines when each variable is defined is *equal* to the control-flow condition under which that variable is used [21]. Once this property is established, variable definitions can be treated as token producers, and uses as token consumers. As a side-effect, the transformation also removes false dependencies between computations.

As an example (using a C-like syntax), the program fragment

```
...
if (g) {
  y=y+1;
}
...
```

translates to:

```
...
y0,y1=$\phi^{-1}$(g,y);
if (g) {
  y2=y1+1;
}
y3=$\phi$(g,y0,y2);
...
```

The pseudo-function ϕ^{-1} uses the condition `g` to generate two conditional copies of `y` — one for the case when the guard is false (`y0`), and the other when the guard is true. At the end of the selection statement, a conventional gated ϕ-function is used to merge the two possible paths where `y` may be defined. The combination of these two functions, with initializer and sink functions used to create initial tokens as well as consume unused tokens suffices to transform programs into dataflow graphs that can handle both if-statements as well as while- or for-loops [21].

The decomposition step maps the ST form of a CHP program onto a concurrent dataflow graph. The decomposition uses a synthesis technique for asynchronous logic known as projection [14], which allows the partitioning of a deterministic asynchronous program into concurrent entities that do not share any variables except via communication channels. Theorems 1 and 2 from [21] show that such a mapping guarantees both equivalence as well as correctness of the dataflow graph that is obtained from ST form, and contain the details of the transformation.

The concurrent dataflow graph that is obtained from ST form is composed of the seven types of fundamental nodes described in Section 1: *copy*, *function*, *split*, *merge*, *source*, *sink* and *initializer*. Token flow is represented by directed edges, which are implemented as channels in asynchronous pipelines. With our AFPGA architecture, *initializer* nodes can be fused into input buffers or state units, while the other nodes can be mapped onto different units in a logic block. The remainder of the synthesis procedure maps the resulting concurrent dataflow graph onto the AFPGA blocks.

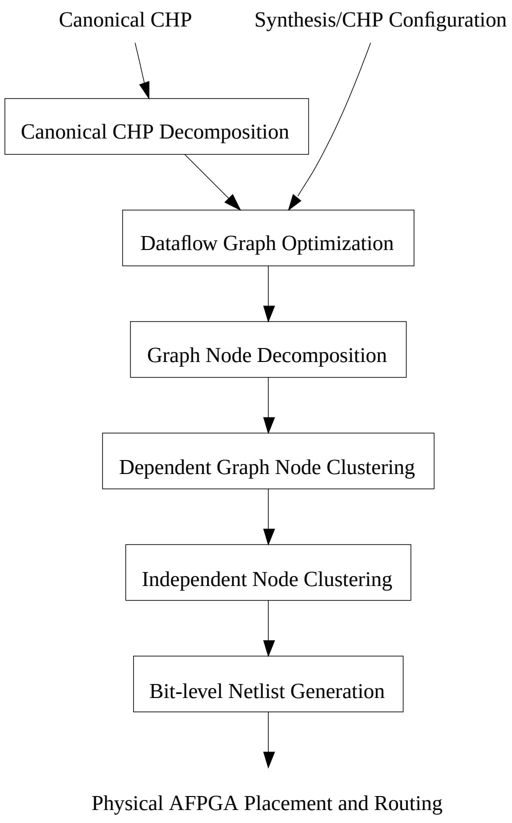

Canonical CHP
Synthesis/CHP Configuration
Canonical CHP Decomposition
Dataflow Graph Optimization
Graph Node Decomposition
Dependent Graph Node Clustering
Independent Node Clustering
Bit-level Netlist Generation
Physical AFPGA Placement and Routing

Figure 4: Automated synthesis flow.

2.2 Synthesis Configuration

In addition to the sequential canonical CHP program, the user can also customize the configuration for the synthesis procedure. Currently, we support customization of the following settings for synthesis:

- *Logic density level*: Greater logic density can be achieved by trying to merge unrelated dataflow nodes into a single AFPGA logic block. However, high density may result in difficult or infeasible routing, and possibly degraded performance. Lower density levels often increase the chance of successful routing.

- *Copy tree structure*: *logarithmic* or *linear*. Information that has to be propagated to multiple destinations must be explicitly copied, as the AFPGA communication channels are point-to-point. Logarithmic copying produces copies of tokens with balanced trees, whereas linear copying produces copies with linear trees. Logarithmic copying is preferred in bit-aligned datapaths, whereas linear copying is more appropriate for bit-skewed datapaths, which are common in feed-forward signal processing applications. Linear copying also results in easier place-and-routing. Choosing the appropriate copy structure can result in significant differences in performance.

- *Carry chain structure*: Our AFPGA architecture features both *fast* and *normal* carry chains. Since fast carry chains use low-latency dedicated channels, they greatly improve performance of arithmetic operations. One current drawback is that we have not integrated the automatic placement phase of `vpr` for fast carry chains.

- *Data width*: In general, CHP channels or variables may have arbitrary bit-width, especially for integer operations with unspecified widths (e.g. "`a+b`"). The configuration file can be used to override default width values.

2.3 Concurrent Dataflow Graph Optimization

Once we have generated an initial dataflow graph using the techniques described above, we use a sequence of optimizations to improve the quality of the synthesis. Figure 5 shows some of the common dataflow optimizations we use in our synthesis method. The optimization procedure is repeated until a fixed-point is reached. Each optimization and its rationale is described below.

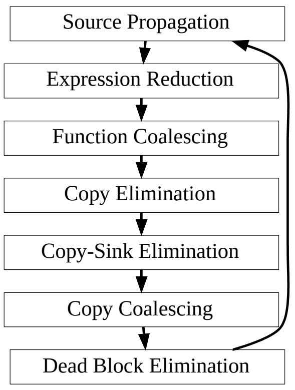


Figure 5: Dataflow graph optimization procedure.

Source Propagation. Source-propagation is similar to constant propagation in dataflow analysis. Constant sources arise due to the syntax-directed nature of the projection transformation that generates the initial dataflow graph [14]. Since source nodes repeatedly communicate the same constant value, their values may be folded into the function blocks that consume them. Reducing the number of inputs to functions may be result in fewer, simpler functions blocks, which may then accommodate other inputs.

Expression Reduction. Functions with constant inputs can often be reduced to functions of lower arity, or even constant outputs. We apply common arithmetic and logical identities to reduce expressions. (ex may be a multi-bit expression, b is a single bit expression.)

1. Arithmetic:

$$\begin{array}{cc} b+0=0+b\to b & ex+0=0+ex\to ex \\ b-0\to b & ex-0\to ex \\ \bar{1}-ex\to\sim ex & 0-ex\to -ex \end{array}$$

2. Logical (any number of bits, all commutative):

$$\begin{array}{ccc} b\,\&\,0\to 0 & b \mid 0\to b & b\oplus 0\to b \\ b\,\&\,1\to b & b \mid 1\to 1 & b\oplus 1\to\sim b \\ b\,\&\,b\to b & b \mid b\to b & b\oplus b\to 0 \end{array}$$
$$\begin{array}{cc} b\,\&\,(\sim b)\to 0 & b \mid (\sim b)\to 1 \\ b\oplus(\sim b)\to 1 & \sim(\sim b)\to b \end{array}$$

3. Logical shift:

$$ex \gg 0\to ex \qquad ex \ll 0\to ex$$
If N is greater than ex's bit-width,
$$ex \gg N\to 0 \qquad ex \ll N\to 0$$

For arithmetic right-shifts, the result depends on the sign bit.

When all inputs of a function are constant (null-ary function), one can replace the function with a source of its output value.

Function Coalescing. Composite functions may be coalesced into one function as long as the arity of the resulting function does not exceed the input capacity of a logic block (in our architecture, the limit is 4). One example is shown in Figure 6. Not only does function-coalescing result in increased logic density, but it also reduces the amount of communication over channels, thereby reducing energy significantly.

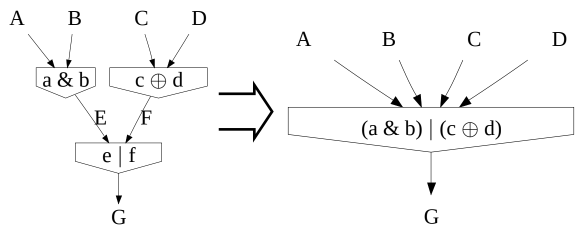


Figure 6: Logic function merge example.

Copy Elimination. A copy block can be eliminated if it copies to only one output. In this case the input channel is routed directly to the single output channel. Eliminating a copy block results in reduced pipelining on the channel that was routed through the copy block. Since the AFPGA is slack-elastic, change in pipelining does not impact the correctness of the implementation. Deadlock cannot arise because the AFPGA logic block itself contains sufficient pipelining to prevent it. This differs from a synchronous design, where such a transformation (bypassing a value around a register) would require global retiming.

Copy-Sink Elimination. If an output of a copy block is consumed by a sink, then both the output and the sink can be eliminated. Reducing the number of output copies may result in more opportunities to eliminate copies, as described above.

Copy Coalescing. Cascaded copy blocks may be coalesced if the number of final outputs does not exceed the output capacity of each logic block (4 for our architecture). One example is shown in Figure 7. Reducing the number of copy blocks may potentially improve performance by shortening the forward path of tokens. This is especially important in programs with long loop-carried dependencies.

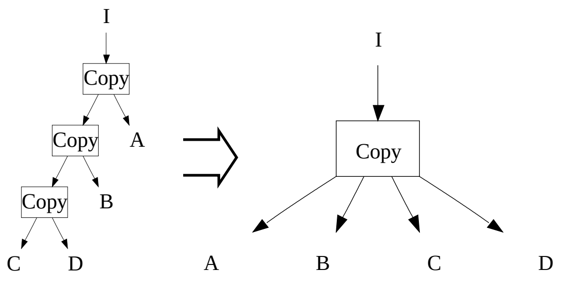


Figure 7: Copy block coalescing example.

Dead Block Elimination. Any block whose outputs are all unused or connected to sinks is a *dead block*, analogous to *dead-code elimination*. Dead blocks may be removed from the dataflow graph with all of the block's inputs redirected to sinks. Sinks can be placed on the inputs to the dead block, and the process repeated to find further opportunities for optimization.

Figure 8 illustrates an example of applying these optimizations to a concurrent dataflow graph. In this particular example, the number of nodes is reduced from 9 to 4 and the forward path length is reduced from 5 hops to 1.

2.4 Graph Node Decomposition

Since nodes in an abstract dataflow graph may have arbitrary in-degrees and out-degrees, it may be necessary to decompose over-sized nodes to fit into a particular AFPGA architecture's logic blocks. While the coalescing transformations will not introduce new dataflow blocks that have too many inputs, they may exist in the original dataflow graph. Candidate nodes for graph node decomposition are: function, split, merge, and copy.

Function Decomposition. We decompose function nodes recursively and iteratively using the following guidelines:

- Since arithmetic operations require two operands, an arithmetic subexpression must be factored out into its own function block.
- Any remaining function with greater than four input variables (for our architecture) must be decomposed until its components have at most four inputs.

This process may introduce auxiliary copy nodes for variables that appear in multiple decomposed subexpressions. For instance, the example shown in Figure 9 requires the introduction of a copy for variable b that now appears in two different function blocks.

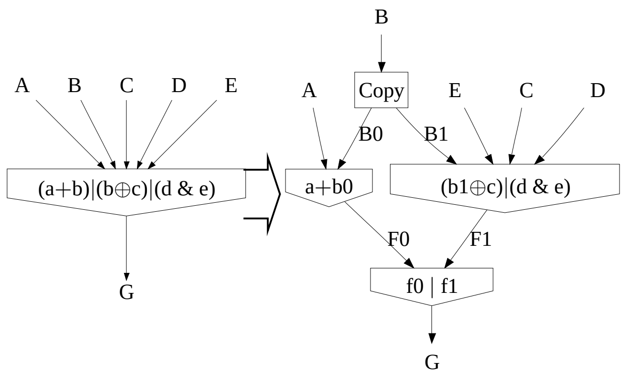
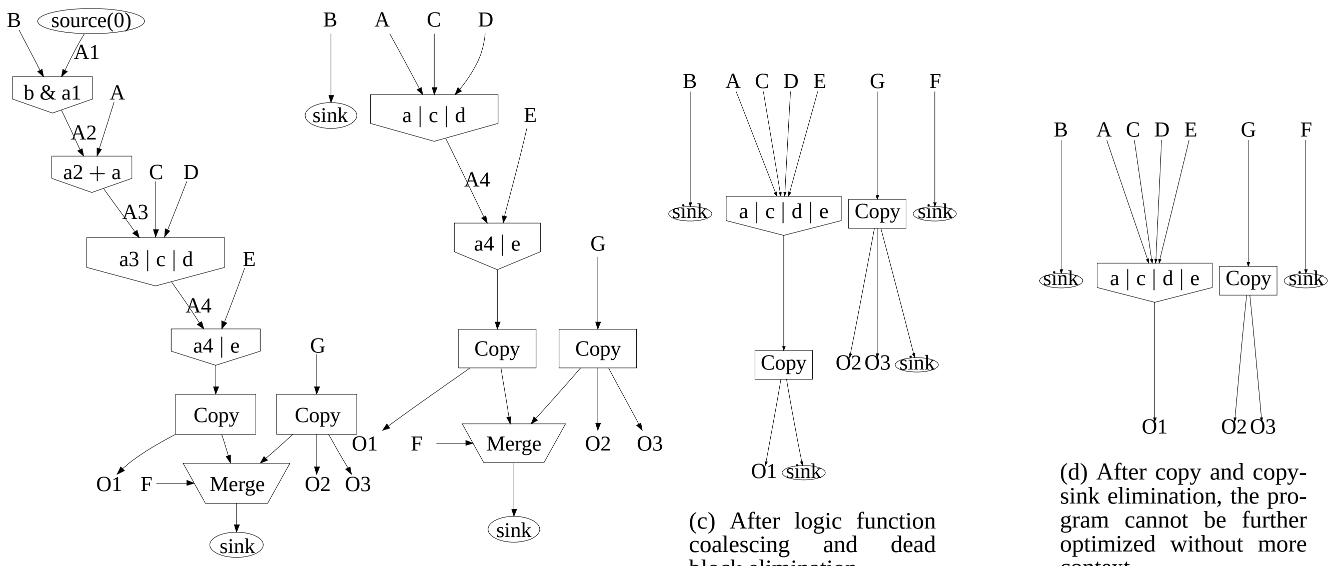


Figure 9: Function decomposition example.

Split/Merge Decomposition. In general, split and merge nodes of dataflow graphs may conditionally communicate with an arbitrary number of other nodes. A split node with N receivers requires $\log_2 N$ bits of control to address a receiver. Since our AFPGA architecture only supports 2-way splits and merges with one bit of control, we must decompose large splits and merges into 2-way components. Currently, we only support balanced decompositions with $\log_2 N$ stages. However, applications with non-uniform conditional communication frequencies may benefit from Huffman-encoded decompositions, which result in shorter token latencies for the more frequent paths.

Copy Decomposition. Any copy node with more than 4 outputs will be decomposed into a copy tree. The resulting copy tree can be



(a) Original program

(b) After source propagation and expression reduction

(c) After logic function coalescing and dead block elimination

(d) After copy and copy-sink elimination, the program cannot be further optimized without more context.

Figure 8: Dataflow graph optimizations example.

logarithmic or linear, depending on the configuration file. For the logarithmic tree structure, we guarantee that the number of hops from the root to each leaf is equal.

2.5 Dependent Graph Node Clustering

After dataflow graph optimization and decomposition, each node in the resulting graph has a fanin/fanout that is compatible with the AFPGA architecture. However, a one-to-one mapping would result in poor utilization of the AFPGA logic block. Each AFPGA logic block is equipped with the following resources: three source units, one function unit, one 2-way conditional split/merge unit, two 4-way copy units, and one sink unit, as shown in Figure 3. Multiple dataflow graph nodes may be clustered to a single logic block if their resources (including total number of inputs and outputs) do not conflict. Clustering results in increased logic density, which may potentially improve performance by reducing latency on long interconnect paths. We describe a heuristic for clustering dependent graph nodes, nodes that are directly connected by dataflow edges.

Given our AFPGA logic block architecture (Figure 3), we give function units the highest priority because they are directly connected to all other types of nodes in the same logic block. Conditional merges and splits are also directly connected to all other types of nodes, except that they cannot communicate directly with function inputs. Copy nodes may only receive data from the function output, conditional output, and low latency copies from the logic block inputs. Source and sink nodes only communicate directly with function and conditional units. Units that cannot communicate directly with each other must be synthesized by routing them through other auxiliary units, which is less efficient. For example, a source cannot communicate directly with a copy; it must go through an auxiliary function unit first.

It is important to note that different logic block and interconnect architectures may be better suited with different clustering heuristics. For example, logic blocks with more fully connected internal components (e.g. full-crossbar connection) will naturally have fewer restrictions on direct communication between the components. Our primary objective is to minimize communication overhead through the pipelined interconnect by coalescing as many connected (dependent) graph nodes into each logic block as possible. A secondary objective is to minimize the use of auxiliary intermediate units. Given the constraints of our logic block architecture, we describe our simple greedy heuristic for clustering dependent graph nodes:

Step 1. Allocate one logic block for each function node, tentatively directing each function's output to its logic block's output. Try the following until each block's resources or I/O are exhausted:

1. For each input of the function block that comes from a source unit, place its source unit into the same cluster.
2. For each function's output that is an input to a conditional (merge or split) node, place the corresponding conditional unit into the same cluster. Redirect the function's output through the conditional unit and back out to the logic block's output.
3. For each block output that sends to a copy node, merge that copy node into the same cluster.
4. For any block input that is sunk, merge its sink node into the same cluster.

Step 2. Allocate a logic block for each remaining un-clustered conditional node. Try the following until each block's resources or I/O are exhausted:

1. For each input of a conditional node that comes from a source node, then place its source in the same cluster.
2. For each output of a conditional node that is sent to a copy node, place the copy node in the same cluster.

3. For each output of a conditional node output is that is sent to a sink node, place the sink node into the same cluster.

Step 3. Allocate a logic block for each remaining un-clustered copy node. If its input comes from a source node, merge its source into the same cluster.

Step 4. Allocate a logic block for each remaining un-clustered source node.

Step 5. Allocate a logic block for each remaining un-clustered sink node.

Other graph covering techniques [5] can also be applied to dependent node clustering. However, the above greedy heuristics is simpler while achieving high density of clustering.

2.6 Independent Graph Node Clustering

After clustering dependent graph nodes together, there may be opportunities to combine independent graph nodes—unrelated nodes where their resource demands fit within a logic block. The logic density level configuration determines how aggressively independent nodes are clustered together. Increased logic densities may result in more difficult (sometimes infeasible) placement and routing.

Each decision whether to cluster independent nodes is based on two characteristics: (i) Whether or not the two nodes are on the same path, i.e. if they are indirectly connected through other parts of the whole dataflow graph; (ii) Whether or not the two nodes communicate variables of the same bit-width. Currently, our implementation supports three logic density levels: (a) low: no independent nodes are clustered; (b) normal: attempt to merge clusters only if they lie on different paths in the graph and they communicate variables of the same bit-width; (c) high: in addition to the normal criteria, also try to merge nodes that lie on the same path regardless of their respective bit-widths.

Attempting to cluster nodes on different paths first may be more appropriate for computation flow graphs that are mostly feed-forward because it reduces the number of artificial cycles formed in the physical mapping (place-and-route). However, computation graphs with feedback loops may benefit from combining nodes that lie on the same path by imposing physical locality on closely related nodes. Comparing these slight variations in independent node clustering heuristics requires more thorough experimentation with computation graphs with different topologies.

One example of independent node clustering is shown in Figure 10. After dependent node clustering, we initially have clusters numbered (1) through (5), where cluster (3) is already a collection of dependent nodes. Nodes (1) and (4) include only a split unit, and nodes (2) and (5) include only a function unit. With the lowest density configuration, clustering would stop at this point. With the normal logic density level, nodes on different paths may be coalesced, which could result in nodes (1) and (2) being grouped into cluster (1+2). With the high logic density level, nodes on the same path may be merged, logic blocks on the same path may be coalesced, which could result in nodes (4) and (5) being grouped into cluster (4+5).

2.7 Bit-level Netlist Generation

Up to this point, the data representation of variables and channels in the dataflow graph has been abstract; variables are in general multi-bit values, and the aforementioned transformations can be applied regardless of operand bit-width. To generate the final bit-level AFPGA netlist, we must decompose multi-bit components into their single-bit constituents. This is relatively straightforward,

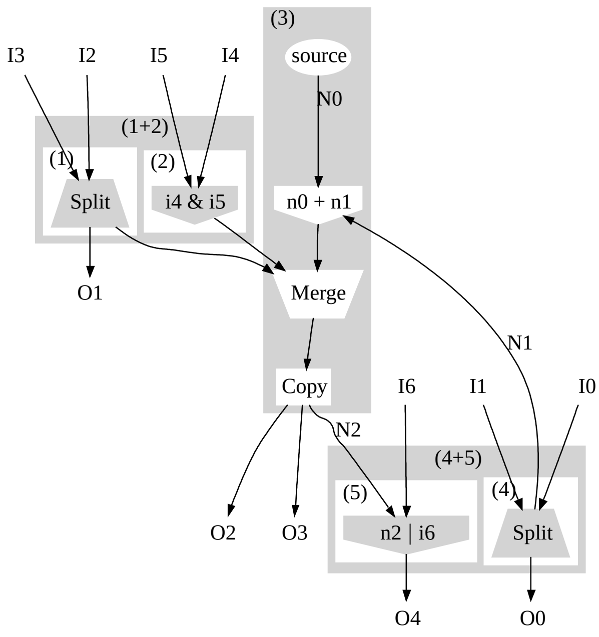


Figure 10: Node cluster and block merge example.

with the carry chain configuration being used during synthesis of arithmetic operations. The resulting AFPGA netlist can be mapped onto our AFPGA architecture using `vpr` [2] for place-and-route.

2.8 Time Complexity

Suppose V is the number of nodes and E is the number of edges in original concurrent dataflow graph. The time complexity of constructing the canonical CHP decomposition is $O(V+E)$. For our dataflow graph optimizations, the number of iterations is loosely bound by $O(V+E)$ because each iteration monotonically decreases the number of edges or nodes. The time spent on each iteration is $O(V+E)$, and thus the total cost of the optimizations is $O((V+E)^2)$. The number of dependent and independent node clustering is $O(V)$ and time cost for each such operation is loosely $O(E)$, so the clustering complexity is $O(V \cdot E)$. Netlist generation, given a constant maximum bit-width, is proportional to the number of nodes and proportional to the number of edges, and thus has $O(V \cdot E)$ complexity.

The time complexity of the whole synthesis procedure is polynomial with respect to the number of graph nodes and edges, $O((V+E)^2)$.

3. EVALUATION

We evaluate the automated synthesis method using several asynchronous circuit benchmarks. Our benchmark inputs are high-level sequential CHP descriptions, which we automatically synthesize into low-level AFPGA netlists. The resulting netlists are given to `vpr` for placement and routing. Our AFPGA was designed using conservative SCMOS design rules in TSMC $0.18\mu m$ technology. We obtained delay values from `spice` simulations of the extracted layout and used them to back-annotate an asynchronous switch-level simulator to quickly and accurately (within 5% of `spice`) evaluate the performance of benchmarks. The peak operating frequency of our AFPGA is 690 MHz for applications that use any function units, and 830 MHz for applications without function units.

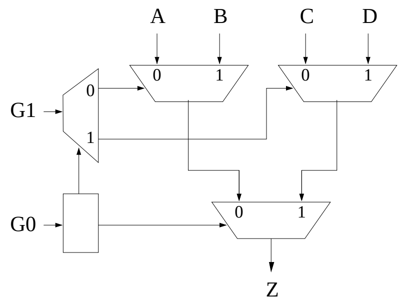
Many of our asynchronous benchmarks are taken from components of larger complex asynchronous designs, such as microprocessor units. The first two benchmarks are ALU-type circuits: an integer full-adder and a function block that evaluates boolean functions of two input integer variables based on a two-bit control input. The next two benchmarks are bit-level circuits: a bit-sifting array (called "pop-sifter") and two linear feedback shift-registers (LFSR) with different numbers and positions of the feedback taps. The register bypass benchmark is a controlled router between two inputs and two outputs. The writeback unit is a control-intensive process from a full-custom asynchronous implementation of a MIPS processor [17] that is used for precise exception handling. The PC unit is the core of the instruction fetch of an asynchronous processor designed in a class project, and it calculates the sequence of program counter (PC) values based on decoded instructions.

Due to the absence of asynchronous FPGA synthesis flows for asynchronous FPGAs, it is difficult to compare the results of our synthesis to standard design flows. Asynchronous circuits operate differently from their synchronous counterparts, and even the logical behavior of a block as simple as an adder differs in synchronous and asynchronous logic. Keeping the inherent limitations of comparing synchronous and asynchronous synthesis approaches in mind, we attempted to create "equivalent" synchronous benchmarks for comparison purposes.

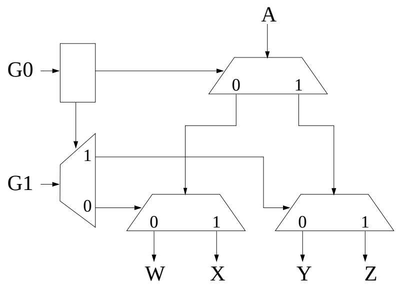
We used Verilog to implement synchronous versions of our benchmarks. (The "pop-sifter" was omitted because its synchronous implementation would incur a large amount of overhead unless it were hand-tuned to be implemented as a systolic array.) The following rules were used in the conversion: (i) The functionality in terms of data-dependencies or data-flow must be preserved; (ii) If an asynchronous benchmark had zero or one input to the entire module, the synchronous version would always assume that inputs were available at each clock edge; (iii) Otherwise, the inputs and outputs were augmented with a full/empty bit. This bit is checked before inputs are read and set when an output is produced on each cycle. These rules would allow a synchronous circuit to process one data item per cycle if its inputs were always available. The rules result in limited flow control, because a block cannot produce an output until all its inputs are ready.

The synchronous benchmarks were mapped to the Xilinx Virtex XCV150 (speed grade 6) FPGA (5-layer-metal $0.22\mu m$) using the Xilinx™ ISE Foundation tools. We chose this FPGA because our AFPGA targets implementation in terms of 5 metal layers at $0.18\mu m$, and its function unit was designed with many of the same features and resources as the blocks in the XCV150. The results from the synchronous syntheses are shown in Table 1, where *frequency* is the expected clock frequency as reported by the synthesis tools. (We also give approximately scaled frequencies for a hypothetical Virtex manufactured in a $0.18\mu m$ technology.) For each benchmark, we also list the FPGA resources required in terms of registers and 4-LUTs.

For each of the aforementioned benchmarks, we used our CAD tools to synthesize and then place-and-route the AFPGA. For each benchmark, all three logic density levels were used: low (l), normal (n), high (h). For comparison with the automatic synthesis results, we also generated hand-synthesized and placed netlists for all benchmarks, except for the PC unit (due to its complexity). The hand-placed netlists were generated with placement macros that simply expanded patterns suitable for each benchmark. Many of our benchmark programs are relatively small so they could be feasibly hand-synthesized for comparison purposes. The results for our syntheses are shown in Table 2. For each benchmark, we report the following information: (i) "LB," the number of AFPGA logic blocks used; (ii) "Func," the number of function units used; (iii) "Cond," the number of condition units used; (iv) "Copy," the number of copy units used.



(a) 4-way merge



(b) 4-way split

Figure 11: 4-way conditional dataflow blocks, where G is a two-bit control channel (G0 is the lower bit of the control channel, and G1 is the upper bit).

Comparison With Hand-Synthesis. For all benchmarks, the automated synthesis achieved logic densities and performances comparable to those of the hand-placed versions. The majority of benchmarks attained at least 95% of the peak performance of the AFPGA. The PC unit is the only benchmark that suffers because the program contains a long feedback loop. The throughput of the AFPGA is not limited by local circuit performance but by data-dependencies. In fact, this is a real performance problem not just for the AFPGA but for the specific PC unit implementation due to the lack of a branch delay slot in the asynchronous design.

With the high density configuration, our synthesis method was able to use as little as 40% fewer logic blocks than with hand-placed versions. Although macro-placement makes for easy hand-synthesis, it generally under-utilizes the available resources. For some benchmarks, such as the PC units, synthesizing with the highest logic density resulted in infeasible routes. For some benchmarks, the synthesis results with higher logic density resulted in higher throughput, while others resulted in lower throughput. This is caused by the fact that our place-and-route is currently blind to the issues that impact the performance of asynchronous logic. As a result, there need not be any correlation between logic density and performance. We have not attempted to assist `vpr` with information about the cost of routes that constitute long latency loops. Thus, the physical mappings produced are by no means optimal.

Comparison With Synchronous Benchmarks. In spite of the

Table 1: Benchmark statistics for Xilinx Virtex XCV150 ($0.22\mu m$)

Benchmark	Registers	LUTs	Frequency (MHz)	($0.18\mu m$) (MHz)
16-bit adder	0	18	178	218
8-bit function block	0	10	267	326
register bypass	0	10	160	196
16-bit LFSR with 4-tap	1×16-bit	1	218	266
16-bit LFSR with 6-tap	1×16-bit	2	220	269
writeback unit	1×32-bit	13	104	127
8-bit PC unit	2×1-bit 3×8-bit	25	153	187
16-bit PC unit	2×1-bit 3×16-bit	49	145	177

Table 2: Benchmark statistics for automated synthesis and hand placement ($0.18\mu m$)

Benchmark	LBs	Func.	Cond.	Copy	Throughput (MHz)	% of Peak Performance
16-bit adder (synthesis(l,n,h)†)	16	16	0	0	674	97.7%
16-bit adder (hand)	16	16	0	0	674	97.7%
8-bit function block (synthesis(l,n,h))	8	8	0	0	688	99.7%
8-bit function block (hand)	8	8	0	0	688	99.7%
6-bit pop sifter (synthesis(l,n))	36	30	0	30	672	97.3%
6-bit pop sifter (synthesis(h))	30	30	0	30	655	94.9%
6-bit pop sifter (hand)	42	30	0	32	688	99.7%
16-bit LFSR with 4-tap (synthesis(l,n,h))	16	16	0	16	678	98.2%
16-bit LFSR with 4-tap (hand)	16	16	0	16	689	99.9%
16-bit LFSR with 6-tap (synthesis(l,n,h))	17	17	0	16	689	99.9%
16-bit LFSR with 6-tap (hand)	17	17	0	16	689	99.9%
register bypass (synthesis(l))	39	0	16	23	824	99.2% $*$
register bypass (synthesis(n,h))	30	0	16	23	824	99.2% $*$
register bypass (hand)	32	0	16	32	824	99.2% $*$
writeback unit (synthesis(l))	60	5	40	18	688	99.7%
writeback unit (synthesis(n))	57	5	40	18	537	77.8%
writeback unit (synthesis(h))	53	5	40	18	656	95.0%
writeback unit (hand)	54	6	38	19	658	95.3%
8-bit PC unit (synthesis (l))	138	35	85	55	293	42.5%
8-bit PC unit (synthesis (n))	123	35	85	55	218	31.6%
16-bit PC unit (synthesis (l))	254	67	165	91	270	39.1%

$*$ Peak performance is relative to 690 MHz for benchmarks that use any function units and relative to 830 MHz for benchmarks that use no function units.
† l - low density level; n - normal density level; h - high density level

limitations of comparing synchronous and asynchronous versions of our benchmarks, it is instructive to examine Tables 1 and 2 in terms of frequency of operation and resource utilization. The frequency in Table 1 corresponds to the maximum performance attainable. In all the benchmarks except the PC unit, this would be the steady-state performance of the system. However, due to stalls on the valid bit, the PC unit would operate at 50% of the frequency of the synchronous FPGA.

The differences in performance are compelling. Not only does the AFPGA outperform the synchronous FPGAs by nearly a factor of three (even accounting for technology scaling), but our synthesis flow results in physical mappings that achieve close-to-peak throughput on AFPGAs. As opposed to this, synthesis methods for synchronous FPGAs have difficulty attaining peak clock frequencies, and there have been studies of fixed-frequency FPGAs to combat this limitation [23].

In terms of logic density, the adder and function block benchmarks have essentially the same 4-LUT utilization in both the synchronous and asynchronous case, as expected. When examining benchmarks such as the LFSR, writeback, or PC unit, a resource comparison becomes complicated because the Xilinx FPGA tools identify macros whereas our current AFPGA architecture does not have any dedicated macros. However, the large number of condition and copy units in the 16-bit PC unit indicate that a future re-optimized AFPGA logic block might include additional resources for handling control-intensive benchmarks. The large number of condition units is also caused by the limitations of a two-way split/merge as the condition unit. Decomposing a four-way split into two-way splits not only requires a tree of splits, but the control inputs to the second-level splits also require additional splits. This situation is illustrated in Figure 11, which shows a 4-way split and a 4-way merge implemented with 2-way splits and merges. Modifying the condition unit to support 4-way split/merges could drastically reduce the number of condition units required by the AFPGA.

4. RELATED WORK

Early asynchronous FPGA architectures [6, 13, 18] were largely based on clocked FPGA circuits, which made them hazard-prone and inefficient at prototyping asynchronous logic. While later designs [12, 8, 22] concentrated on implementing programmable circuits more optimized towards asynchronous logic, their performance failed to match clocked FPGAs. More recent work [19, 20, 24] has demonstrated that asynchronous dataflow FPGAs, whose logic blocks provide both computation and high-speed pipelining, can compete with synchronous FPGA architectures. In an orthogonal direction of research, asynchronous circuits have been investigated for use in course-grain reconfigurable architectures [10].

Previous tools developed for asynchronous FPGAs have concentrated on removing timing hazards that are introduced in asynchronous logic when they are mapped onto circuits that are either purely synchronous or based on synchronous designs. JBits is a Xilinx programming tool that has been used to adjust delays in small asynchronous circuits that have been mapped onto clocked FPGA devices [11]. The place-and-route tool for the Montage AFPGA [7] forced asynchronous signals with fanout to have equal delay along their branches to prevent timing hazards. Since the Montage FPGA used circuits derived from a clocked FPGA, the logic mapper tool also had to guarantee that only one LUT input changed at a time to prevent glitches from appearing on its output. Some of synthesis optimizations in this paper were also used in PipeBench compiler [3] for a synchronous pipelined FPGA. In contrast to previous tools, the synthesis tool described in this paper is targeted towards a high-performance dataflow AFPGA [20].

Our synthesis tool specifically addresses the problem of efficiently synthesizing asynchronous logic for the AFPGA architecture, by translating high-level descriptions of circuits into optimized, bit-level dataflow graphs suitable for physical mapping to the AFPGA. Furthermore, our AFPGA can use clocked place-and-route tools to route data channels rather than individual signals, and the circuits guarantee correct operation of the AFPGA regardless of the delays on the channel signals [20].

5. CONCLUSION

We described an automated synthesis method for an asynchronous FPGA (AFPGA) architecture. With dataflow graph representation and a variety of optimizations, the high-level asynchronous circuit description can be transformed into low-level AFPGA netlist correctly and efficiently. The experimental evaluations show that our synthesis method can achieve both good logic density and high throughput compared to hand-placed counterparts. To the best of our knowledge, this is the first automated synthesis method that can map asynchronous designs to pipelined asynchronous FPGAs. The results we have obtained can serve as the basis for future investigations into asynchronous FPGA architectures and synthesis flows. Comparisons with larger and more complex benchmarks would greatly improve the evaluation of our synthesis.

In the future, we will incorporate arbitration support to our synthesis method so that non-deterministic systems can also be synthesized automatically. This requires more than an extension to the synthesis algorithm; non-determinism can destroy the property of slack elasticity, and new theory is required to safely handle transformations with non-determinism. User-interface improvements that could be implemented include non-canonical form support in sequential program decomposition. The heuristics for optimization could be improved by incorporating dataflow graph topology analysis and routing resource budget scheduling to improve independent node clustering. Finally, the place-and-route tools could be improved by incorporating asynchronous pipeline dynamics into a timing-driven optimization phase.

6. REFERENCES

[1] C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, 1999.

[2] V. Betz and J. Rose. VPR: A new packing, placement, and routing tool for FPGA research. In *Proceedings of International Workshop on Field Programmable Logic and Applications*, September 1997.

[3] M. Budiu and S. C. Goldstein. Fast compilation for pipelined reconfigurable fabrics. In *Proceedings of International Symposium on Field Programmable Gate Arrays*, February 1999.

[4] J. B. Dennis. *The Evolution of 'Static' Data-Flow Architecture*. Prentice Hall, 1991.

[5] Y. Guo, G. J. M. Smit, H. Broersma, and P. M. Heysters. A graph covering algorithm for a coarse grain reconfigurable system. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, June 2003.

[6] S. Hauck, G. Borriello, S. Burns, and C. Ebeling. Montage: An FPGA for synchronous and asynchronous circuits. In *Proceedings of International Conference on Field Programmable Logic and Applications*, Vienna, August 1992.

[7] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An FPGA for implementing asynchronous circuits. *IEEE Design and Test of Computers*, 11(3):60–69, 1994.

[8] D. L. How. A self clocked FPGA for general purpose logic emulation. In *Proceedings of of the IEEE Custom Integrated Circuits Conference*, May 1996.

[9] Xilinx Inc. Virtex™ 2.5V field programmable gate arrays. Xilinx Data Sheet, 2002.

[10] H. Kagotani and H. Schmit. Asynchronous piperench: Architecture and performance estimations. In *Proceedings of Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[11] E. Keller. Building asynchronous circuits with JBits. In *Proceedings of International Conference on Field Programmable Logic and Applications*, August 2001.

[12] R. Konishi, H. Ito, H. Nakada, A. Nagoya, K. Oguri, N . Imlig, T. Shiozawa, M. Inamori, and K. Nagami. PCA-1: A fully asynchronous self-reconfigurable LSI. In *Proceedings of International Symposium on Asynchronous Circuits and Systems*, March 2001.

[13] K. Maheswaran. Implementing self-timed circuits in field programmable gate arrays. Master's thesis, U. C. Davis, 1995.

[14] R. Manohar, T.-K. Lee, and A. J. Martin. Projection: A synthesis technique for concurrent systems. In *Proceedings of International Symposium on Asynchronous Circuits and Systems*, April 1999.

[15] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, June 1998.

[16] A. J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report CS-TR-93-28, Caltech, 1993.

[17] A.J. Martin, A. Lines, R. Manohar, M.Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the Conference on Advanced Research in VLSI*, 1997.

[18] R. Payne. Asynchronous FPGA architectures. *IEEE Computers and Digital Techniques*, 143(5):282–286, 1996.

[19] J. Teifel and R. Manohar. Programmable asynchronous pipeline arrays. In *Proceedings of International Conference on Field Programmable Logic and Applications*, September 2003.

[20] J. Teifel and R. Manohar. Highly pipelined asynchronous FPGAs. In *Proceedings of International Symposium on Field Programmable Gate Arrays*, February 2004.

[21] J. Teifel and R. Manohar. Static tokens: using dataflow to automate concurrent pipeline synthesis. In *Proceedings of International Symposium on Asynchronous Circuits and Systems*, April 2004.

[22] C. Traver, R. B. Reese, and M. A. Thornton. Cell designs for self-timed FPGAs. In *Proceedings of of ASIC/SOC Conference*, 2001.

[23] N. Weaver, J. Hauser, and J. Wawrzynek. The sfra: A corner-turn FPGA architecture. In *Proceedings of International Symposium on Field Programmable Gate Arrays*, February 2004.

[24] C. Wong, A. J. Martin, and P. Thomas. An architecture for asynchronous FPGAs. In *Proceedings of International Conference on Field Programmable Technology*, December 2003.