# Automatic Pipeline Register Placement Through Backannotation in Chisel

Donggyu Kim, Wenyu Tang
University of California, Berkeley
{dgkim, wenyu}@eecs.berkeley.edu

## Abstract

We improve the Chisel automatic pipelining tool by automatically optimizing the placement of the pipeline registers. We do this by back annotating Chisel graph nodes with delay data obtained through synthesis and then using this delay information to optimize the placement of the pipeline registers through the use of simulated annealing.

## 1. Introduction

In the existing automatic pipelining tool, the designer can specify the exact placement of the pipeline registers by labeling particular Chisel nodes as pipeline boundary nodes. Alternatively, the designer can label a small subset of the chisel node graph with pipeline stage numbers and have the automatic pipelining tool infer where to place the pipeline registers. In this second method of pipeline specification, the tool will pick some legal pipeline register placement without considering the consequences of the pipeline register placement on critical path delay. A pipeline register placement is legal if every combinational logic node has all of its inputs in the same pipeline stage.

In the second method of specification, we want to automatic pipelining tool to be able to infer not only a legal pipeline register placement, but also a pipeline register placement that optimizes the critical path delay. This will allow the designer to be able to create a well-balanced pipelined design without going repeatedly going through the vlsi design tools to get feedback on critical path length. Combined with ability of the automatic pipelining tool to automatically generate control logic, the designer will be able to very quickly explore the design space of different pipeline depths and hazard

## 2. Background

### 2.1 Automatic Pipelining

This project is based on the existing automatic pipelining tool built on top of Chisel[1]. The existing automatic pipelining tool allows the RTL designer to specify a one cycle implementation of a finite state machine along with a set of annotations that specify the pipeline stage number of a subset of Chisel graph nodes in the design. During the Chisel backend elaboration process, the automatic pipelining tool transforms the original one stage implementation of the finite state machine into a pipelined design with the correct hazard resolution logic according to the pipeline stage number annotations. The tool resolves pipeline hazards (which are confined to read-after-write hazards) in one of three ways: interlocking, bypassing, and speculation. The tool resolves pipeline hazards with interlocking by default and the user can select particular state elements to be resolved with bypassing and speculation with additional annotation.

This tool expands upon the work by Nurvitadhi et al [2], which provided a means to automatically pipeline finite state machines by having the designer label every combinational and sequential node in the design in a hierarchical manner using a specialized specification language. Our automatical pipelining tool does not require the user to annotate every node with a stage number and automatically finds the close-to-optimal pipeline register placements.

### 2.2 Chisel Backannotation

## 3. Project Objectives

## 4. Technical Approach

Automatically finding a close-to-optimal placement of the pipeline registers takes consists of two major parts. First, we must back annotate the Chisel node graph with delay information obtained through the VLSI tools. Once we have the delay data for each Chisel node, we then use simulated annealing to place the pipeline registers based on the delay data obtained with the back annotation tool.

### 4.1 Chisel Backannotation

#### 4.1.1 Why Backannotation?

Back-annotation is a process to provide post-processing information that reflects implementation dependent characteristics to the circuit design. For Chisel designs, it is a process to provide post Chisel compilation information that is not available in the Chisel backend.

Chisel back-annotator is developed to provide this kind of information to the the Chisel graphs in the Chisel backend, and thus, to help other tools take an advantage of the information. For example, delay information is not available until

logic synthesis, but the automatic pipelining tool uses it to put pipeline registers at the optimal places. Delay information can be obtained by timing reports from Synopsys tools, and the Chisel back-annotator delivers this information to each Chisel node to help the automatic pipelining tool work on purpose.

Another example is FPGA power modeling. To measure power efficiently, we have to generate counters for important signals that are provided by the Chisel backannotator. The power model generator produces linear power models using RTL simulations, and then the Chisel backannotator marked important signals in the Chisel graph after reading the power model. As a result, the counter generator produces counters for important signals using back-annotated Chisel graphs.

#### 4.1.2 Challenges

There are several challenges for Chisel back-annotation. The first is naming problems, which occurs when the destination nodes have different names. Signal names are given during Verilog translation, a process after the elaborate process and this is highly affected by graph topology. However, the Chisel back-annotator wants to use a low-level Chisel graph which is elaborated by the backend, while other tools like the automatic pipelining tool want to manipulate a high-level Chisel graph which is only applied by part of the elaborate process. The two graphs look different from each other, which makes it difficult to match signal names between them. Thus, we define the intermediate-level Chisel graph so that both the back-annotator and the tools are satisfied.

Table 1 shows the original elaborate flow and modified one for backannotation. remove type nodes and trace nodes are processes tools want to avoid because they change the graph undesirably. Thus, we apply processes necessary for backannotation prior to remove type nodes and trace nodes, and then obtain an intermediate-level Chisel graph. Signal names are assigned and information is backannotated to the intermediate-level graph. Tools like the automatic pipelining tool also manipulate this backannotated graph.

The second challenge is missing signals in logic synthesis. Missing signals are inevitable although we give signal preserving options to Synopsys tools. This occurs when Chisel nodes are mapped to generic logic blocks by Synopsys tools. These generic logic blocks are replaced by technology-dependent logic cells, and signals in the logics are replaced at the same time.

An approach to this problem is calculate the difference of the arrival times of two non-missing signals in a timing path, and assigning the average value to missing signals. For example, suppose we have a path starting from X through T1, T2, and T3 to Z where T1, T2, and T3 are missing signals, and the arrival times of X and Y are 0.1 and 0.4, respectively. The difference of the arrival times is 0.3. Each missing signal is assigned 0.1 as a delay since there are three missing signals. This is a naive way to estimate delays for missing signals, but we believe this is a beginning step for delay estimation.

Another challenge is conditional statements like if and switch. These statements are converted into logic gates in logic synthesis which are not visible in Chisel graphs.

### 4.2 Automatic Pipeline Register Placement

The automatic pipelining tool first creates a legal placement of pipeline registers based on the stage numbers of the user annotated nodes and then uses simulated annealing based on the delay data provided by the backannotation to find a close-to-optimal placement of the pipeline registers.

#### 4.2.1 Pipeline Legality

For a pipeline register placement to be legal, it must satisfy the following three conditions: **(1)** Every combinational logic node has all input signal with the same stage number.

**(2)** The stage number of every combinational logic node's output signal is greater than or equal to the maximum of its input signal stage numbers.

**(3)** There are two ways to determine the stage number of a node. One way is to trace through the node's inputs to a pipeline register and set the stage number of the node equal to the stage number of that pipeline register. Another way is to trace through the nodes consumers to a pipeline register and set the stage number of the node equal to the stage number of that pipeline register  1. For all nodes in the graph, the stage number of the node obtained through both methods must be the same.

#### 4.2.2 Wire Node Insertion

Before the automatic pipelining tool infers the initial legal placement of the pipeline registers, it inserts Chisel Bits nodes representing wires between combinational logic nodes in the original design. This ensures that a pipeline boundary never has to fall across a combinational logic node, which would be difficult to reconcile with the notion of Pipeline Legality discussed above.

#### 4.2.3 Initial Placement

The automatic pipelining tool breaks cycles in the Chisel graph by splitting architectural registers(registers present in the original one cycle design) into read and write points. The read point consists of the output of the register and the write points consist of the write data and write enables going into the register. Hence, the data sources in the Chisel node graph are read points, input nodes, and constants and the data sinks in the Chisel node are the write points and output node.

Due to reasons that will become apparent from the algorithm discussed below, in general the designer must annotate all of the data sources and data sinks in the Chisel graph in order for the automatic pipelining tool to produce a legal pipeline register placement.

The automatic pipelining tool produces the initial legal pipeline placement by propagating stage numbers out

from the user annotated Chisel nodes to their inputs and consumers in a pseudo breadth-first-search (BFS) manner. When two propagation frontiers with different stage numbers meet at the same node, propagation down that path stops and pipeline registers are inserted at that node.

We must maintain the following conditions during the pipeline stage propagation process:

**(1)** Adjust the propagation rates of each propagation frontier so that two different propagation frontiers never meet at a combinational logic node and always meets at a wire node mentioned in 4.2.2, because it does not make sense to split a combinational logic node in half with a pipeline register.

**(2)** The stage number propagated to a combinational logic node with multiple inputs must be the maximum of the stage numbers of all of its inputs. If this was not maintained, we would have some inputs of the combinational logic node have a greater stage number than the stage number of the output of the combinational node, which violates 2nd condition of Pipeline Legality. This also means that we cannot propagate to a Chisel node with multiple inputs from the input side until all of its inputs have been propagated to.

**(3)** The stage number propagated to a combinational logic node with multiple consumers must be the minimum of the stage numbers of all of its consumers. If this was not maintained, we would have some consumers of the combinational logic node have a smaller stage number than the stage number of the output of the combinational node, which cause that consumer to violate the 3rd condition of Pipeline Legality. This also means that we cannot propagate to a Chisel node with multiple consumers from the consumer side until all of its consumers have been propagated to.

Due to conditions (2) and (3), if not all of the data sources and data sinks are user annotated, the stage propagation process may never complete.

#### 4.2.4 Optimizing Register Placement

Once the tool finds an initial legal pipeline register placement, it uses simulated annealing to optimize the placement of the pipeline registers based on the delay data obtained from the Chisel backannotation.

**Simulated Annealing Algorithm**.

We want a fast method of finding a close-to-optimal pipeline register placement without getting stuck at local minimums. Simulated annealing is a gradient decent like algorithm that iteratively improves upon the current solution by randomly generating neighbor solutions from the current solution and then picking the neighboring solution that decreases the cost function the most as the next solution. Unlike gradient decent, simulated annealing occasionally picks neighbor solutions that are worse than the current solution. This allows the algorithm to avoid getting stuck at local minimums. The algorithm uses the following formula to decide if it should choose a neighbor solution as the next solution:

R(0, 1) is a random number in the interval [0, 1] At higher values of T (temperature), the algorithm is more likely to accept bad solutions. As the algorithm progresses, the value of T is lowered so that the algorithm eventually settles on a close-to-optimal solution. On each iteration, we lower T by the following formula:

This cooling schedule forces the algorithm to try many bad solutions in the very early iterations and then focus on trying good solutions for the majority of the remaining iterations. There are many possible cooling schedules, but this one has worked well for us.

We terminate the algorithm when the temperature falls below 0.01 or if the iteration count exceeds 3000. Weve found the critical path length has generally converged by those two limits in the designs we evaluated. The cost function used for our problem is the maximum combinational path length in the design and we obtain neighbor solutions from the current solution by moving Chisel nodes across pipeline boundaries in a way that mains a legal pipeline register placement

**Generating Neighbor Solutions**.

We must maintain a legal pipeline register placement as we generate new pipeline register placements from the current pipeline register placement. To accomplish this, we look at the current pipeline register placement and choose eligible combinational logic nodes to move across pipeline stage boundaries. A combinational logic nodes is eligible if it meets the following conditions:

**(1)** The node is not an architectural register read point, architectural register write point, input node, or output node. We choose to not allow the automatic pipelining tool to move these nodes because doing so could have large unforeseen performance implications. For example, if we are pipelining a RISC processor, moving the write point of the PC register to later stages will cause branches to be resolved later in the pipeline and thus cause a larger branch mispredict penalty.

**(2)** The node outputs directly to a pipeline register. In this case, it is safe to move the node forward across the pipeline boundary by removing the pipeline register from the output of the node and adding pipeline registers to all the inputs of the node and still maintain a legal pipeline register placement.

**(3)** The node has all of its inputs driven directly by a pipeline register. In this case, it is safe to move the node backward across the pipeline boundary by removing the pipeline registers from the inputs of the node and adding a pipeline register to the output of the node and still maintain a legal pipeline register placement.

See figure x for an illustration of why violating (2) and (3) would result in an illegal pipeline register placement.

**Evaluating Cost Function**.

We calculate the longest combinational path in the design by finding the propagation delay to each combinational logic node from an architectural register or pipeline register recursively. The propagation delay to a node is the maximum of (sum of propagation delay to input and delay across input)

over all of its inputs. The propagation delay to the output wire of architectural registers and pipeline registers is considered zero (we do not have the means to properly backannotate clk-to-q delays currently). Thus, the longest combinational path is the maximum of the above calculated propagation delay over all nodes in the Chisel graph.

When we generate a new pipeline register placement from the current pipeline register placement, we do not need to recalculate the propagation delay to every node. Instead we store the propagation delays of all the Chisel graph nodes for the current pipeline register placement in a map of Chisel node to propagation delay and incrementally update the map as we modify the pipeline register placement. When we move a combinational logic node forward across a pipeline boundary, we only have to recalculate the propagation delay for the nodes that are on the consumer tree of the node that was moved. When we move a combinational logic node backward across a pipeline boundary, we only have to recalculate the propagation delay for the nodes that were previously on the consumer tree of the node that was moved. This allows us to evaluate the cost of new pipeline register placements quickly, even for large designs.

### 4.3 Tool Flow

## 5. Results

### 5.1 Chisel Backannotation

### 5.2 Automatic Pipeline Register Placement

### 5.3 Combined Tool Results

## 6. Conclusion and Future Work

## References


[1] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12.

[2] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. L. Lu. Automatic pipelining from transactional datapath specifications. In *Design Automation and Test in Europe*, DATE '10.