Automatic Functional Datapath Optimization

Wenyu Tang

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-101
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-101.html

May 16, 2014

Automatic Functional Datapath Optimization

A System For Digital Circuit Frontend Specification

Wenyu Tang
University of California, Berkeley
wenyu@eecs.berkeley.edu

Abstract

Digital hardware frontend specification is defined by two extremes - RTL specification and HLS specification. RTL specification produces highly optimized designs, but requires extremely verbose low level specification on the part of the designer. On the other hand, HLS specification requires much less verbose high level specification from the designer, but does not produce well optimized designs. In this thesis, I present a middle ground that captures the pros of these two extremes - a system in which the designer specifies a basic datapath and uses a tool to automatically apply optimizations to that basic datapath.

Contents

1. Introduction

Over the past 30 years, progress in programming languages has greatly increased the productivity of software design by moving low level resource allocation and optimization tasks away from the programmer to the compiler or interpreter. In contrast, increase in productivity of digital hardware logic design has largely stalled after the transition from schematic based specification to to Register Transfer Level (RTL) specification languages such as Verilog or VHDL for front end specification.

While RTL specification succeeds in providing the digital logic designer with a specification that is higher level than transistor level or gate level schematic specification, RTL specification is still quite a low abstraction level and requires the designer to provide a very detailed specification of the design. High Level Synthesis (HLS) has been an attempt to increase the productivity of digital logic designers by going to a much higher level of specification than RTL and automatically synthesizing all of the details of the design. However, the higher abstraction level of HLS comes at

the cost of unsatisfactory performance, area, and power characteristics due to inefficiencies introduced in the automatic logic synthesis process.

In this thesis, I propose a system in which the designer specifies a basic functional datapath in a RTL like manner and selects optimizations that should be applied to the design, which will then be automatically applied through a tool. This middle ground method increases digital logic designer productivity without producing designs with unacceptable performance, power, and area characteristics like HLS. I implement two examples of such a system, AutoMArch and AutoFAME, and I describe their implementation and example applications.

2. Background

2.1 RTL And Its Limitations

In a RTL specification, digital logic designers specify a synchronous digital circuit in terms of data flow between synchronous state elements such as flip-flops and SRAM memory blocks. There are generally two types of variables in RTL languages: registers and wires. Registers correspond to synchronous state elements and wires correspond to the output of intermediate combinational logic blocks. The designer specifies the state elements needed by declaring registers and specifies the combinational logic that connects the state elements by using conditional assignment operators along with arithmetic operators on registers and wires declared within the design. This RTL specification is then synthesized into a gate level specification through a synthesis tool and then fed into the physical design portion of the IC design flow.

The first limitation of RTL specification is that mainstream RTL languages, such as Verilog and VHDL, are based on discrete event driven simulation languages and the syntax and semantics of discrete event driven simulation languages are not entirely natural for specifying digital logic. For example, state elements are not explicitly defined, but are instead inferred from variables that are specified to update when a clock signal transitions. This hurts source code readability because in order to understand a design specified by the RTL, the user needs to not only parse the semantics of the RTL, but also mentally reason about the circuit constructs implied by the RTL. The semantics of discrete event driven simulation languages also allows the designer to create un-synthesizable constructs in RTL. This leads to RTL designs that pass tests in simulation, but cannot be physically realized as a circuit.

The second limitation of RTL specification is that although RTL specification frees the designer from specifying every transistor or logic gate and their connections, RTL specification is still very tedious and verbose as the designer has to worry about exactly how every state elements updates every clock cycle. This also makes it easy for the designer to lose the big picture algorithmic view of the design as the designer is bogged down in the implementation details. The functional behavior of the design is obfuscated by the implementation details of the optimizations applied to the design.

2.2 Chisel

The first limitation of RTL mentioned in the above section can be addressed through use of RTL languages that are not based on discrete event driven simulation languages. One example of this is Chisel[1], a RTL language developed by UC Berkeley. Unlike Verilog and VHDL, it specifies digital circuits through explicit circuit component construction, not inference based on a discrete event simulator semantics. This makes it more intuitive to specify digital design through RTL because there is a very simple mapping from the RTL constructs to physical circuit constructs. Additionally, all RTL designs written in Chisel that pass tests in simulation can be physically realized as a circuit.

While Chisel does not address the second limitation of RTL specification, it will be used to specify the base functional datapath for the digital hardware frontend system proposed in this thesis.

2.3 HLS And Its Limitations

High Level Synthesis (HLS) has been an attempt to address both limitations of RTL specification mentioned in 2.1 by going to a higher level of specification than RTL. In this paradigm, the designer specifies the logic design as a data flow graph through sequential variable updates in a C like language, which frees the designer from having to specify the cycle by cycle operation of a specific datapath. Then a HLS tool maps the dataflow graph to a datapath based on performance, power, and area constraints and synthesizes a gate-level description of the design from the high level specification, which is then fed into the physical design portion of the IC design flow.

Unfortunately, designs synthesized from HLS specifications generally fail to have acceptable performance, power, and area characteristics compared to equivalent designs synthesized from RTL specifications. Much effort has been put into making the synthesis process produce more optimal designs, but rapid breakthroughs are unlikely given that several subproblems in the process have been proven to be NP-Hard[7].

3. Proposed Solution

HLS produces designs with unacceptable performance, power, and area characteristics because the synthesis tool has to solve the computationally difficult problem of formulating a datapath that executes the dataflow graph and fits within the given performance, power, and area constraints. HLS tools do synthesize well optimized designs for specific patterns that occur in the high-level specification, so hardware designers working with HLS find themselves tuning high-level code to make specific synthesis tools produce exactly the datapath they want.

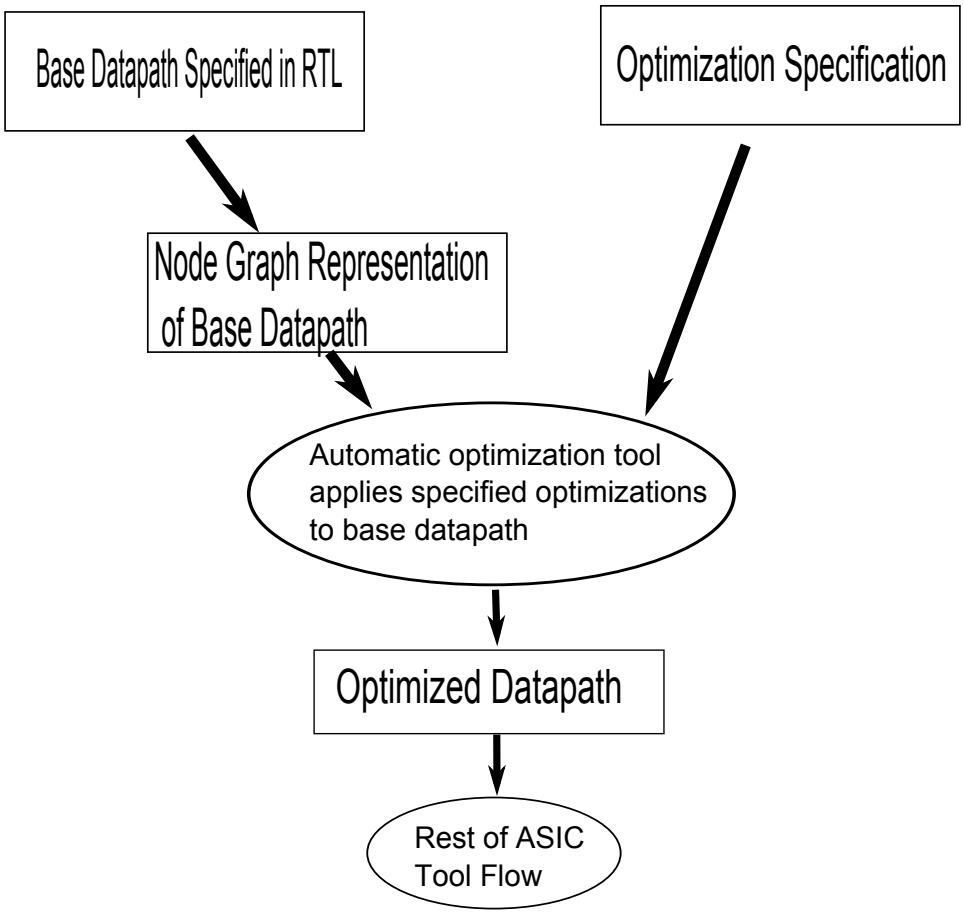

Figure 1: General Tool Flow

This is clearly a case of automation trying to do too much. The HLS tools have a hard time formulating optimized datapaths, so the designers have to essentially tell the HLS tools what datapath they should use in a roundabout way by tuning the high-level specification. Clearly, designers would be more productive if they can specify the datapath directly.

It seems like this conclusion tells us that designers should just do logic design using RTL in the traditional manner. However, much of traditional RTL specification deals with issues outside of simple datapath design. Logic designers spend much of their time specifying additional logic required to make the datapath fit performance, power, and area constraints. Some common optimization techniques include time-multiplexing functional units, pipelining, multithreading, out-of-order execution, etc. These commonly used datapath optimization techniques can be captured as algorithms and applied automatically.

This thesis proposes a system in which the designer creates a RTL specification of a functionally correct base datapath with no optimizations implemented and separately specifies a series of optimizations to be performed on the datapath. Then automatic tools that know how to generically apply common optimization techniques can apply the specified optimizations to the base functional datapath and produce an optimized gate level specification to be fed into the next step in the IC design flow which may be physical design, FPGA synthesis, or simulation. This work flow is shown in Figure 1. This system of specification will hence be referred to as Automatic Functional Datapath Optimization (AFDO). AFDO allows the designer to specify a digital circuit with higher productivity than RTL specification without incurring the performance, power, and area penalties of HLS specification.

The automatic optimization tools discussed above work in the following manner. First, some initial processing transforms the RTL specification of the base datapath into a node graph data structure, where each node represents a digital circuit component (wire, combinational logic block, or state element) and each node contains input and consumer pointers to other nodes representing the topology of the circuit. Second, the tools implement the specified optimizations by modifying this node graph - creating new nodes, changing input/consumer pointers, copying existing nodes, etc. Third,

the tool outputs the modified circuit in some specified representation.

Although AFDO can be implemented if the base datapath is specified in discrete event simulation based RTL languages such as Verilog or VHDL, it is better for the base datapath to be specified in a structural construction based RTL language such as Chisel. First, as mentioned in 2.1, discrete event simulator semantics are not particularly well suited for digital hardware specification and create a host of problems for the designer. Second, creating a node graph representation of the base datapath is much easier in structural construction based HDL languages such a Chisel as there is a one to one mapping between language construct and nodes in the node graph. In the case of Chisel, the automatic tools can directly operate on Chisels internal node graph data structure. In discrete event simulation based RTL languages, we have to first send the RTL specification through a gate level synthesis tool before we can construct the node graph data structure. This makes preserving names difficult and prevents the designer from using RTL level simulations of the automatically optimized designs. All of the automatic optimization tools discussed in the following sections apply transformations to base datapaths specified in Chisel or a reduced Chisel like RTL specifically designed to demonstrate AFDO that will be referred to as Scalpel.

AFDO increases logic designer productivity by addressing both limitations of RTL specification covered in 2.1. By using a structural construction based RTL language like Chisel to specify the base functional datapath, the problems caused by discrete event simulator semantics are eliminated. By allowing the designer to specify only the base functional datapath, which is generally much less complex than the optimized design, and applying optimizations automatically, source code verbosity and algorithm obfuscation is eliminated. Now the designer can specify designs much faster and with fewer errors as he only has to specify the simple base functional datapath and use automatic tools to systematically apply the desired optimizations. Additionally, AFDO allows designers to more easily do design space exploration as they can produce new design points by simply selecting different optimization options for the automatic tools to apply instead of rewriting obfuscated RTL for each new design point. For example, adding a pipeline stage to a processor design in RTL is generally non-trivial in RTL because the pipeline control logic is entwined within the datapath specification. If the designer specifies a base functional 1 stage version of the processor and uses an automatic tool to pipeline the processor, adding a pipeline stage would be trivial on the part of the designer as he would simply have to tell the automatic pipeline tool to generate one more stage.

At the same time, AFDO retains performance, power, and area benefits of RTL specification by preserving a high level of correlation between language constructs and the generated hardware. The generated datapath is simply the base functional datapath modified with the designer specified optimizations. Unlike HLS, the designer knows the number and types of functional units used in the datapath as well as the state elements that are present in the datapath along with how those state elements are updated.

4. Evaluation

I implement two automatic optimization tools, AutoMArch and AutoFAME, to demonstrate how the AFDO specification system could work. The first tool AutoMArch automatically applies in order pipelining and multi-threading to a base functional datapath. The second tool AutoFAME automatically applies the FPGA emulation optimizations introduced in the *A Case for FAME: FPGA Architecture Model Execution paper* [11] to a base functional datapath.

5. AutoMArch

AutoMArch is capable of creating multi-threaded in-order designs of any number of threads and any number of pipeline stages that is functionally equivalent to n-copies of the input base functional datapath, where n is the number of threads. The functionality of the input base datapath is defined by its input output behavior and its next state update behavior. Two designs are defined as functionally equivalent if given the same starting architectural state and sequence of inputs, the two designs transition to the same ending architectural state and produce the same sequence of outputs. Architectural state elements are defined as the state elements present in the input base functional datapath. AutoMArch always preserves the state elements present in the input base functional datapath, but may add additional state elements such as pipeline registers that are not considered architectural state. AutoMArch takes an input base functional datapath specified in Scalpel and outputs the multi-threaded in-order design as a Chisel source file, which can be feed into various simulation, FPGA, and ASIC flows.

5.1 Input Datapath Restrictions

The input base functional datapath can be an arbitrary FSM specified in Scalpel with the following restrictions:

(1) The input FSM must communicate to the outside world through ready/valid ports

(2) The designer cannot use input valid or output ready signals as inputs to any part of their circuit

(3) Any functional units that may take more than one clock cycle to return responses(caches, multipliers, dividers, etc) must be accessed through the Variable Latency Unit Interface discussed below.

Condition **(1)** is necessary because the transformed design will have different input to output latency than the original design depending on what stages the input and output ports are placed in the pipeline. Additionally, some pipeline stages may contain bubbles or be stalled and it would be incorrect to receive inputs or produces outputs under these

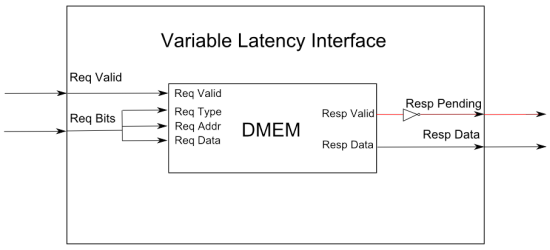

Figure 2: Single Thread View of Variable Latency Unit Interface Black wire are user facing IO, red wires are tool facing IO

conditions. Thus by enforcing that only ready/valid IO ports are used, correct IO behavior can be defined by only the sequence of valid inputs tokens accepted and sequence of valid outputs tokens produced, without any restriction on the exact timing of when the input tokens are accepted and output tokens are produced.

5.1.1 IO Semantics

When input ready or output valid is driven high by the input FSM, this implicitly signals to the tool that the input FSM requires the use of the input or output port on the current state update. Thus, the tool generates logic that examines the input ready or output valid and stalls the pipeline if the corresponding input valid or output ready is not driven high by external modules. The designer should design the input FSM so that input readies and output valids are only driven high when absolutely necessary to avoid unnecessary stalling the automatically multi-threaded and pipelined version of the circuit.

5.1.2 Variable Latency Unit Interface

In order to accommodate caches and long latency arithmetic units in the base functional datapath specification that does not allow any optimization or control logic to be implemented, the tool provides the Variable Latency Unit Interface. The designer should access any caches or long latency arithmetic units through a Variable Latency Unit Interface and treat that Variable Latency Unit Interface like a piece of combinational logic in the input FSM specification. The designer should not use the Resp Pending port of the Variable Latency Unit Interface to drive any part of their circuit and should pretend that the Variable Latency Unit Interface always gives a valid response immediately. The tool will automatically generate control logic that deals with the Variable Latency Unit Interface not immediately outputting a valid response.

When the tool performs the multithreading transformation, each Variable Latency Unit Interface in the input minimal FSM is replicated n times, where n is the number of threads. It is up to the designer to create glue logic that deals the copies of the Variable Latency Unit Interfaces in a top level module that instantiates the automatically multi-threaded module. The designer can instantiate n copies of the functional unit, one for each Variable Latency Unit Interface, or can create additional logic that multiplexes all copies of the Variable Latency Unit Interface onto a single functional unit.

5.2 Conventions

State Element Partitioning

State elements in the input FSM will be considered as multiple circuit components. For singular registers, the read output is considered a separate component from the write data and write enable inputs and will be referred to as the register's read port. The write data and write enable input is considered together as a single circuit component and will be referred to as the register's write port. For memories, which could be either an array of registers or a SRAM block, each read port and each write port is considered a separate component.

This partitioning of the state elements into separate read and write ports make it easier for AutoMArch to analyse the input datapath because it allows the datapath to viewed as a acyclic circuit component graph in which data flows from the state element read ports and input pins through combinational logic nodes into the state element write ports and and output pins.

Pipeline Terminology

Traditionally, in order pipelining is discussed with regards to processors or stateless combinational units. Pipelining stateless combinational units is trivial as there are no inter-pipeline stage dependencies. When pipelining processors, we consider the dependencies between each instruction flowing down the pipeline when creating pipeline control logic. However, pipelining as discussed in this thesis can be applied to arbitrary finite state machines. In this context, it does not make sense to talk about instruction to instruction dependencies as we are not only dealing with arbitrary FSMs. Instead, we will consider there to be a series of next state updates flowing down the pipeline. When constructing pipeline control logic, we will consider dependencies between each next state update flowing down the pipeline.

5.3 Automatic Pipelining

I will first cover the specification and node graph transformations required to produce a single thread in-order pipelined datapath from the input base functional datapath. The specification and node graph transformations needed to create multi-threaded in-order pipelined datapaths will build upon

the framework established in this section and will be discussed in 5.4.

5.3.1 Pipelining Options and Specification

First the designer must select the number of pipeline stages the optimized design should have. This is the minimum specification needed. If no additional specification is provided, the tool will automatically place all of the pipeline registers and default to generating an inorder pipeline that resolves all pipeline hazards by interlock.

Pipeline Stage Placement By default, the tool places all of the architectural state element read ports in the first pipeline stage and the architectural state element write ports along with the output ready/valid ports in the last pipeline stage. The tool then automatically places the rest of the circuit components in a pipeline stage that minimizes the critical path delay.

If this default placement is not suitable, the designer is given the option to manually place specific circuit components. For example, in a RISC processor design, the PC register write port should be placed as early as possible in the pipeline to minimize branch penalty. The designer does this by annotating circuit components with the desired stage number in the RTL source file of the base functional datapath. Then when the tool does the automatic placement of the circuit components, it will first place the user annotated components in their specified pipeline stage before it automatically places the rest of the unannotated circuit components.

Thus, the designer has the flexibility to manually place none, some, or all of the circuit components in the base functional datapath. The designer can initially let to the tool place all of the circuit components and gradually manually place components as performance issues are discovered. The designer should be aware that manually placing components constrains the automatic placement algorithm and could potentially lead to suboptimal critical path delays. Thus, the designer should have a good understanding of the design before he starts manually placing components.

Pipeline Hazard Resolution In the traditional framework of inorder processor pipeline design, there are three types of hazards - data hazards, control hazards, and structural hazards. Only data hazards need to be dealt with by the tool. Data hazards arise when the next state update in pipeline stage X reads a state element R that could be written by a next state update in pipeline stage Y, where Y > X, but Y <= the stage of the R's write port. This situation would make the next state update in pipeline stage X use stale read data and result in a non functionally correct design. In traditional pipelined processors, control hazards arise when the next PC information is not available soon enough for branch and jump instructions. This is really just a data hazard on the PC register and does not need to be considered separately. Likewise, in traditional pipelined processors structural hazards arise when instructions in different stages of the pipeline need to write to the register file at the same time, but there is only one register file write port. This cannot happen in AFDO because the input base functional datapath does not have multiple pipeline stages and the tool will not generate logic to have instructions at different pipeline stages write to the same architectural state element write port.

There are three common ways to resolve data hazards - interlocking, bypassing, and speculation. Each option has different performance and area characteristics. The tool generates logic that implements interlocking by default. The designer may choose bypassing or speculation on a read port by read port basis by annotating the particular architectural state element read port they want to be bypassed or speculated in the base functional datapath RTL source. Details of each hazard resolution option will be discussed in 5.3.4

5.3.2 Circuit Node Graph Creation

Because AutoMArch uses Scalpel to specify the input base functional datapath, it does not need to do any additional work to create a node graph data structure representing the input base datapath. Scalpel's internal data structure already uses a node graph to represent the design. All IO pins, wire, logic gates, and state element read and write ports are represented as individual nodes and all nodes maintain a list of their inputs and a list of their consumers. This node graph is convenient for AutoMArch to use directly because it follows the conventions in 5.2 as it uses separate graph nodes to represent every state element's read and write ports and by extension has a acyclic node graph. AutoMArch modifies Scalpel's internal node graph data structure directly and uses Scalpel's built in elaboration methods to produce the Chisel source file that represents the final multi-threaded and pipelined design.

5.3.3 Pipeline Register Placement

The automatic pipelining tool first creates a legal placement of pipeline registers and then creates a optimized placement by balancing the critical path delay in each stage. The automatic placement process preserves the user specified pipeline stage of user annotated base datapath components.

Pipeline Legality

For a pipeline register placement to be legal, it must satisfy the following 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 equal to the shared stage number of its input signals.

(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) All the read ports of a state element must have the same stage number and all the write ports of a state element must have the same stage number. The stage number of a state element's read port(s) must <= that state element's write port(s).

(5) All of the user facing IO pins of a Variable Latency Unit Interface must have the same pipeline stage number, as the designer treats a Variable Latency Unit as a combinational logic unit.

(6) Every input ready/valid IO must have a stage number <= the minimum stage number of every output ready/valid IO. (inputs must be placed before outputs)

It is possible that the designer manually annotates the base datapath in such a way that no legal pipeline register placement can be produced without changing the stage of the user specified components. If this is the case, an error is triggered.

Obtaining a Initial Legal Placement

To make pipeline register placement easier, the tool creates a slightly modified node graph to represent the input base datapath. In this modified node graph, all of the read ports of a state element are merged into one node and all of the write ports of a state element are merged into one node, in order to maintain pipeline legality condition **(4)**. Then, all of the user facing IO pins of a Variable Latency Unit Interface are merged into one node in order to maintain pipeline legality condition **(5)**. Then all of the pins associated with a ready/valid port are merged into a single node. Additionally, a node representing a wire is inserted between all 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. All other aspects of the Scalpel node graph remain the same.

In this node graph, the register readports are source nodes(which have no inputs) and all architectural state element write ports and output ready/valid ports are sink nodes(which have no consumers). All nodes in the input base datapath can be reached through the source nodes' consumer pointers, with the exception of nodes driven only by constants, which don't need to be considered in the pipeline register placement process. All nodes in the input base datapath can be reached through the sink nodes' input pointers. The tool produces the initial legal pipeline placement by propagating stage numbers out from the source nodes to their consumers and the sink nodes to their inputs 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. The source and sink nodes are guaranteed to have pipeline stage numbers at the beginning of the process, whether through user annotation or through the defaults mentioned in 5.3.1

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, 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.

Obtaining an Optimal Placement

After a legal pipeline register placement is obtained, the tool automatically optimizes the pipeline register placement with regards to critical path delay by balancing out the critical path length in each pipeline stage.

It does this by first assigning a delay value for each node in the node graph. In the current implementation, a naive assignment is used. All combinational logic nodes are assigned a delay of 1.0 and all wires are assigned a delay of 0.0. Memory read ports as well as Variable Latency Units are also assigned a delay of 1.0. A more sophisticated assignment based on the delay data collected from ASIC tools could be used, but this naive assignment is mostly sufficient because VLSI retiming can be applied on top of the pipeline register placement performed by the tool. Thus, the pipeline register placement optimization only has to be approximately correct to produce good critical path delay once the design is pushed out through the ASIC design flow. Additionally, real ASIC delay characteristics cannot be fully captured by a static assignment of delay values to each node because they are highly dependent on gate level optimizations performed by the ASIC tools, which varies per design, as well as wire delays, which are dependent the layout of the circuit. So putting too much effort into assigning highly accurate delay values to nodes is futile.

Then, the tool uses the following pressure based algorithm to move the pipeline stage boundaries:

(1) The tool finds the critical path in each pipeline stage and calculates their delays.

(2) On each pipeline stage boundary, the side with the longer critical path delay "pushes" the boundary towards the other side. This "push" is accomplished by taking the end node from the critical path on the side with the longer critical path delay and moving it across the pipeline boundary.

(3) Repeat until the maximum critical path of all the stages stops decreasing.

5.3.4 Data Hazard Resolution Option Details

Data hazard resolution options are selected on a read port per read port basis. Every read port of every architectural state element can have a different data hazard resolution option. By default, data hazards on every read port is resolved by interlocking and the designer can annotate the read ports whose data hazards they want to be resolved by bypassing or speculation.

Interlocking

Interlocking is the simplest way to resolve data hazards. When the next state update at stage X has the possibility of using stale read data due to a next state update at stage y > x writing to an architectural state element being read by the next state update at stage X, we simply stall the pipeline at stage X and inject bubbles into stage X + 1 until the next state update at stage y flows down the pipeline and finishes writing to the architectural state element. Interlocking has the lowest cost in terms of area and cycle time, but has the highest cost in terms of performance because it inserts bubbles into the pipeline whenever a data hazard occurs.

Bypassing

Bypassing allows data hazards to be resolved with fewer bubbles inserted into the pipeline if the write data and write enable signal of the contested architectural state element is earlier than the stage of the write port by inserting muxing the read data pin of the read port with the write data signal as soon as it is available. If the write data and write enable signals are available in the stage immediately after the read port, no bubbles need to be inserted at all. If the write data and write enable signals are only available at the stage of the write port, then bypassing does not produce fewer bubbles than interlocking. Compared to interlocking, bypassing has higher cost in terms of area and cycle time due to the bypass network, but has lower cost in performance because inserts fewer bubbles into the pipeline.

In order to select bypassing as the data hazard resolution option, the designer simply annotates the desired read port as bypassed in the RTL source file of the base functional datapath.

Speculation

For certain architectural state elements such as the PC register in RISC processors, interlocking is not acceptable because of the performance penalty and bypassing does not help because the write data signal of the PC register's write port is not immediately available. In these cases, it is desirable to speculate on the next read value of the architectural state element and then kill next state updates present in the pipeline between the stage of the read port and the stage of state element write port, including the next state update present in the stage of the read port, if the speculated value is determined to be incorrect later.

Control logic generation for arbitrary speculation is very complex, so the tool places the following restrictions on what kind of speculation maybe performed:

(1) Only the read ports of registers maybe speculated upon.

(2) There may only be one speculated register read port per pipeline stage.

(3) Assuming the stage of a speculated read port is stage X and the stage of the corresponding write port is stage Y, there may not be IO ports, Variable Latency Units, architectural state element write ports, and other speculated read ports present in the stages between stage X and stage Y, inclusive of stage X.

The user specifies that a register read port should be speculated upon in by annotating the desired read port as speculated in the RTL source file of the base functional datapath. The user creates a speculative clone of the register associated with the read port speculated upon and creates the update logic for that speculative clone register in the RTL source of the base functional datapath. Then the user uses some annotations to label this clone register as the speculative clone of the original register. The tool will generate logic that muxes the read data port of the original register with the read data port of the speculative clone register. The read data from the speculative clone register will be chosen except on the clock cycle immediate after a mispredict. During that cycle, the data held in the speculative clone register is synced with the correct data in the original register. Mispredicts are detected by pipelining the speculative read data until the stage of the register write port. If the speculative read data at the stage containing the next state update immediately after the next state update present in the write port state is different than the actual write data in the write port stage, all stages between the read port stage and the write port stage, inclusive of the read port stage, are killed.

5.3.5 Pipeline Control Logic Generation

After the tool places the pipeline registers, it then automatically generates the pipeline control logic that keeps the functional behavior of the pipelined data path the same as the functional behavior of the base input circuit. The pipeline control logic will need to deal with both the availability of valid tokens at the ready/valid IO ports as well as the data hazards introduced by pipelining. I will first present the scheme necessary to generate control logic that deals with the availability of IO ports and handles all data hazards through interlocking. Then I will present the control logic

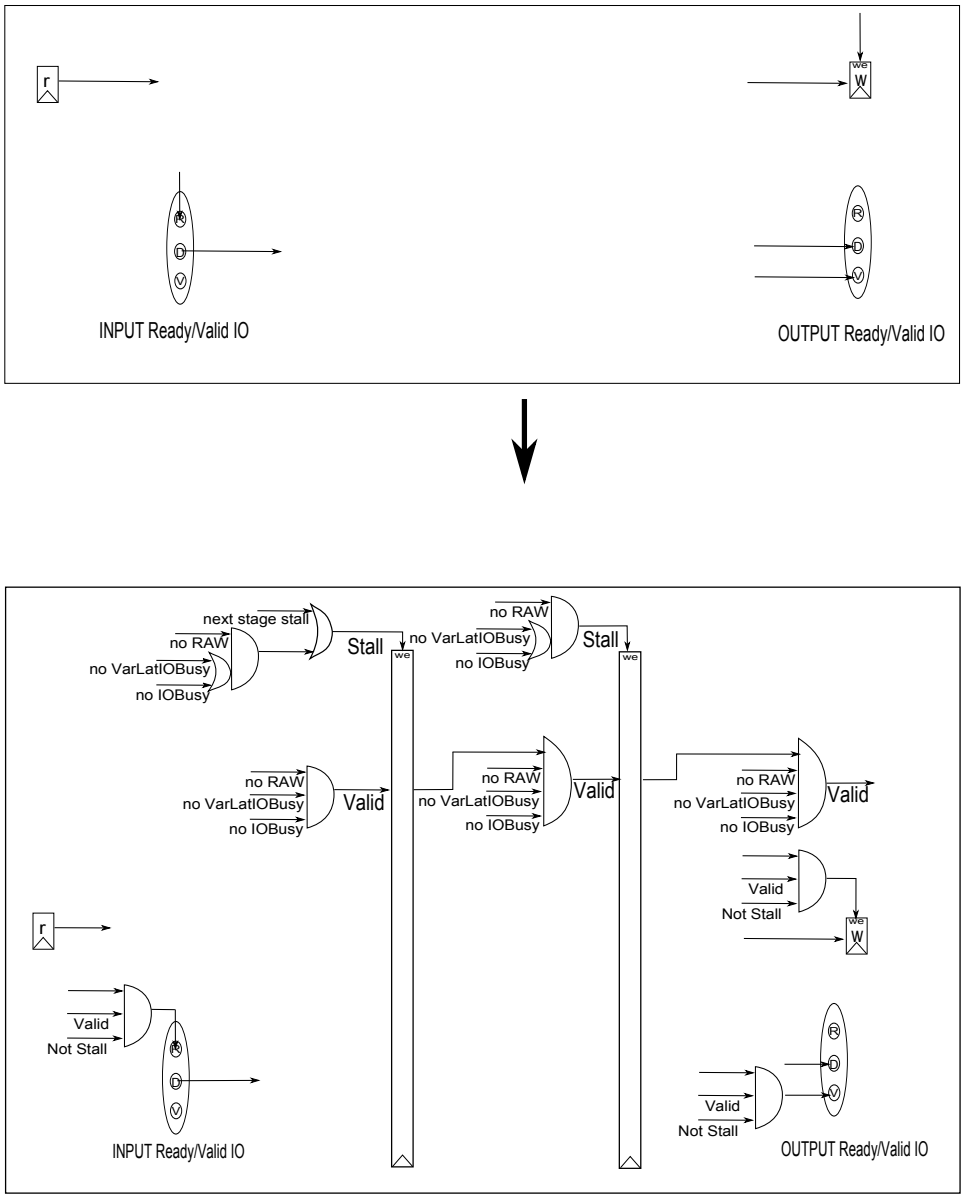

Figure 3: Basic interlocking control logic generated for a 3 stage pipeline. The combinational logic of the original input datapath is not shown for clarity.

needed for bypassing and speculation as modifications on top of the basic interlocking scheme.

The following signals are generated in the basic interlocking scheme:

IO Busy Signals

An IO port is considered busy if the input ready/output valid signal is driven high, which implicitly signals that the data from this port is needed to be consumed/produced, and the input valid/output ready signal is driven low by an outside module, which indicates that the port is not available.

VarLatIO Busy Signals

A Variable Latency Unit is busy if its RespPending port is driven high.

RAW Hazard Signals

A read port belonging to stage X which has corresponding write ports in stage Z has a RAW hazard signal for every stage Y, write port W pair, where X < Y <= Z and 0 < W < `number of corresponding write ports`. A read port is consider to have a RAW hazard for stage Y, write port W if there is a valid next state update in stage Y and there is a possibility that the next state update in stage Y will do a write to the state element associated with the read port at the same address as the read address at stage X. For registers, the read port has a RAW hazard for stage Y, write port W if the version of the write enable signal of write port W at stage Y is high or the write enable signal is produced after stage Y.

For memories, the read port has a RAW hazard for stage Y, write port W if the read enable is high AND (the version of the write enable in stage Y is high or the write enable signal is produced after stage Y) AND (the read address equals the version of the write address in stage or the write address is produced after stage Y).

Valid Signals

The tool generates a valid signal for each pipeline stage. Stage X is currently valid if stage X-1 was valid on the last clock cycle, no state element read ports in stage X has a RAW hazard, no IO port in stage X is busy, and no Variable Latency Unit in stage X is busy. The input ready signals, output valid signals, Variable Latency Unit Req Valid signals, and state element write enable signal are masked with the valid signal for the pipeline stage they belong in.

Was Valid Signals

In order to determine if any pipeline stage was valid on the last clock cycle, the valid signal of every stage is fed into a register. The output of this register is known as the was valid signal. Stage X's was valid signal is fed into the AND gate driving the valid signal of stage X+1.

Stall Signals

The tool generates a stall signal for each pipeline stage. When a pipeline stage is stalled, the contents of that pipeline stage do not progress into the next pipeline stage. The boolean equation for stage X's stall signal = stage X + 1 is stalled OR (stage X was valid last cycle AND (a read port in stage X+1 has a RAW hazard OR a IO port in stage X+1 is busy OR a Variable Latency Unit in stage X+1 is busy)). The input ready signals, output valid signals, Variable Latency Unit Interface Req Valid signals, and state element write enable signal are masked with the valid signal for the pipeline stage they belong in.

The above scheme is illustrated in Figure 3

In order to implement bypassing, the above scheme is modified in the following way. The RAW hazard signals belonging to read ports that are specified to be bypassed are not fed into the AND gate driving the stage valid signal. Instead, the RAW hazard signals are used as the select signals of the bypass muxes placed in front of the bypassed read ports.

In order to implement speculation, RAW hazard signals belonging to read ports that are specified to be speculated upon are removed. Kill signals are generated for every stage between the read port stage and the associated write port stage, inclusive of the read port stage. Every kill signal is driven high when a mis-speculation is detected. The kill signals of each stage are then fed into the AND gate driving the stage valid signals.

5.3.6 Differences From VLSI Retiming

The automatic pipelining ability of AutoMArch appears to be similar to well known VLSI Retiming methods, but it is much more powerful. VLSI Retiming methods are only capable of adding pipeline stages to purely combinational datapath or moving registers around in a manually pipelined sequential datapath. It is not capable of adding pipeline stages to an unpipelined sequential datapath and the way it can move registers around in a manually pipelined sequential datapath is constrained by the placement of the manually constructed pipeline control logic. AutoMArch is capable of adding pipeline stages to arbitrary sequential datapaths as long as they follow the restrictions in 5.1. Additionally, AutoMArch has full freedom to place the pipeline registers in sequential datapaths because it can generate the correct pipeline control logic for any legal pipeline register placement.

5.4 Automatic Multi-threading

The specification and node-graph transformations needed for producing a multi-threaded in-order datapath from a base functional datapath builds upon the framework established in 5.3. The multi-threaded pipelines produced by AutoMArch have all of their architectural state elements, the IO ports, and the Variable Latency Unit IOs replicated n times, where n is the specified number of threads, and the combinational logic is not replicated. From the outside world, the optimized datapath will be functionally equivalent to n-copies of the original base datapath, but the optimized datapath requires much less than n-times the area, where n is the number of threads. The multi-threaded datapaths produced in this manner also obtain better throughput/area than the base functional datapath in the right conditions because multi-threaded datapaths can get rid of next state update to next state update data dependencies and can hide the long access latencies of functional units wrapped in Variable Latency Unit Interfaces.

5.4.1 Multi-threading Options and Specification

In addition to the pipeline configuration options discussed in 5.3, the designer can specify the number of threads they want and whether to use fixed or dynamic interleave thread scheduling policies.

5.4.2 Fixed vs Dynamic Interleave

In fixed interleave, the tool uses a fixed counter to select the next thread to wake up. The tool enforces that the number of threads is greater than or equal to the number of pipeline stages, so there can not be any data hazards between any two transactions in the pipeline because there is always at most one transaction from each thread in the pipeline. This allows the tool to not generate the data dependency check and resolution logic. Additionally, in fixed interleave, the tool does not generate logic to retry a transaction if it encounters a Resp Pending from a Variable Latency Unit Interface. Instead, the whole pipeline stalls. Fixed interleave provides good performance at low area and cycle time cost if the majority of the pipeline stalls will be caused by data hazards in the pipeline and Variable Latency Unit stalls are short or infrequent.

In dynamic pipelining, the tool uses a user supplied thread scheduler that selects the next thread to wake up based on the status of any long latency functional units in that thread. In this mode, even if the tool restricts the number of threads to be greater than or equal to the number of pipeline stages, data hazards between two transactions in the pipeline cannot be avoided because the user supplied scheduler is free to choose any thread order and therefore can place more than one transaction from each thread in the pipeline. Additionally, in dynamic interleave, the tool generate logic to retry a transaction if it encounters a Resp Pending from a Variable Latency Unit Interface and does not stall the pipeline. Dynamic interleave is required to get good performance if Variable Latency Unit stalls are long or frequent.

5.4.3 Circuit Node Graph Creation

The circuit node graph creation process discussed in 5.3.2 does not need to be modified to accommodate multi-threading.

5.4.4 Pipeline Register Placement

The pipeline register placement process discussed in 5.3.3 does not need to be modified to accommodate multi-threading.

5.4.5 State and IO replication

Given a user specification of n threads, the tool first replicates the IO ports the architectural state elements n times. The input ready signals, output data signals, and the output valid signals of the replicated IO elements are driven by the corresponding original IO port signals. The state element write data signals, and state element write enable signals of the replicated state elements are driven by the corresponding original state element write signals.

A mux is placed in front of the input data signals and all consumers of the original input data signal is now driven by this mux. Similarly, a mux is placed in front of the replicated state element read data signals and all consumers of the original read data signal is now driven by this mux. The select pin on these muxes will be driven by the thread sel signal generated by the thread scheduler.

5.4.6 Pipeline Control Logic Generation

The schemes for generating pipeline control logic that deals with both IO port unavailability and the various data hazard resolution strategies discussed in 5.3.5 can be modified to produced multi-threaded in-order datapaths. The following modifications have to be made:

(1) The thread sel signal generated by the thread scheduler has to be pipelined to every pipeline stage.

(2) Input ready signals, output valid signals, Variable Latency Unit Req Valid Signals, and architectural state element write enable signals are masked with the thread sel signal, so that the IO and state elements of thread X are accessed/updated only when thread X is selected.

(3) If dynamic interleave is selected, RAW Hazard signals for read port R at stage Y are masked with the additional condition thread sel at stage of R equals thread sel at stage Y. If fixed interleave is selected, RAW hazards do not need to be generated at all and interlock would be the only necessary data hazard resolution option.

(4) If dynamic interleave is selected, the speculative clone registers need to be replicated just like the architectural registers and the mis-speculation detection logic will need to signal mis-speculate only when the thread sel of the pipelined speculated values match the thread sel of the actual write data and the speculated value does not match the actual write value.

(4) Additional IO Busy and VarLatIO Busy signals need to be generated for the replicated IO and Variable Latency Unit Interface ports for each thread.

(5) IO Busy signals and VarLatIO Busy signals are masked with the thread sel signal of the stage they belong to.

(6) If dynamic interleave is selected, kill signals need to be generated if a Variable Latency Unit signals Resp Pending and the the thread sel signal of the stage of the Variable Latency Unit equals the thread num associated with that Variable Latency Unit.

5.4.7 Example Application

For evaluation of the tool, I created simple RISC processor in Scalpel and explored the design space of number of threads (1 - 4), number of pipeline stages (1 - 4), and fixed vs dynamic interleave. The base RISC processor is a classic RISC machine that contains a ICache with a miss latency of 4 cycles, which is accessed through a Variable Latency Interface, and a magic single cycle DMem.

In order to evaluate how effective the generated multi-threaded designs are at hiding the latencies of long latency functional units in general, which is one of the primary advantages of multi-threading, the ICache is made to pseudo randomly miss based on the output of a LFSR. Each thread accesses its own private ICache through the Variable Latency Unit Interface. This gives a representative evaluation of multi-threaded design effectiveness regardless of the quality of the benchmark program used on the RISC processor.

The metric for performance used is Throughput/Area, which can be broken down into $Task/Cycle{*}(Cycle/Time)/Area$. $Task/Cycle$ and $(Cycle/Time)/Area$ separately are shown separately from the overall Throughput/Area for each design point. Cycle counts are obtained from running an arithmetic benchmark on each design point in Chisel's cycle accurate C++ simulator. The cycle time and area numbers were obtained by pushing each design point through synthesis on a 40nm technology. No VLSI retiming was used.

Dynamic Interleave Results The trends shown in Figure 4. are generally expected. Cycle time should decreases when number of stages increases. Because no retiming was used, this trend shows that the automatic pipeline register placement is working well. Cycle time also should not depend significantly on number of threads. It seems that for

this design, the 3 pipeline stages is the tipping point before the delay added by pipeline register setup and clk-q times out weigh the decrease in delay caused by a shorter combinational path.

The trends shown in Figure 5. are generally expected. Area used should increase when either number of threads or number of stages increase.

The trends shown in Figure 8. are generally expected. Task per cycle increases as the number of threads increases due to fewer data dependency hazards and masking of ICache miss latency. Task per cycle decreases as the number of pipeline stages increases beyond the number of threads because the pipeline becomes over saturated and the threads beging to interfere with each other.

The trends shown in Figure 9. also make sense. For any number of threads, $(Cycle/Time)/Area$ peaks at 3 stages because going from 3 stages to 4 stages with any number of threads causes a increase in area without decrease in cycle time. $(Cycle/Time)/Area$ decreases as number of threads increase because adding more threads increases the area and decreases the clock frequency.

Multiplying Figure 8. and 10. togther, we obtain Figure 10. From this figure, we can see that the 2 thread, 3 stage designs obtains the best overall Throughput/Area.

Fixed Interleave Results The cycle time and area characteristics shown in Figure 6. and Figure 7 are very similar to the cycle time and area characteristics of the dynamically interleaved pipelines. Although we do see slightly smaller area in the fixed interleave pipelines when the number of stages is high, we don't seem to get much savings using fixed interleave for this base design.

Because the fixed interleave policy stalls the entire pipeline whenever a Variable Latency Unit Interface signals Resp Pending, the number of threads and number of pipeline stages makes a negligible impact on Task per cycle as shown in 11.

Figure 12. shows that the $(Cycle/Time)/Area$ statistics are very similar to the $(Cycle/Time)/Area$ statistics for dynamic interleave. This tells us that the additional control logic required by dynamic interleave plays a negligible role in determining area and delay.

Multiplying Figure 11. and 13. togther, we obtain Figure 13. From this figure, we can see that the 1 thread, 1 stage designs obtains the best overall Throughput/Area. However, all of the design points have much lower Throughput/Area than their equivalents with dynamic interleave. Thus, we can conclude that for the example RISC processor design, the latency benefits gained from dynamic interleave outweigh the additional area and delay cost of the more complex control logic in dynamic interleave.

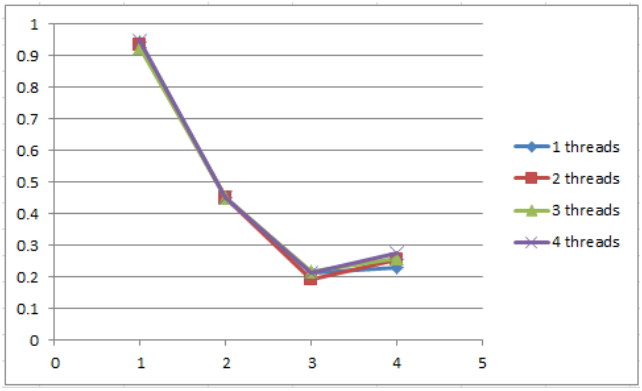

Figure 4: Dynamic Interleave Cycle Time The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of ns.

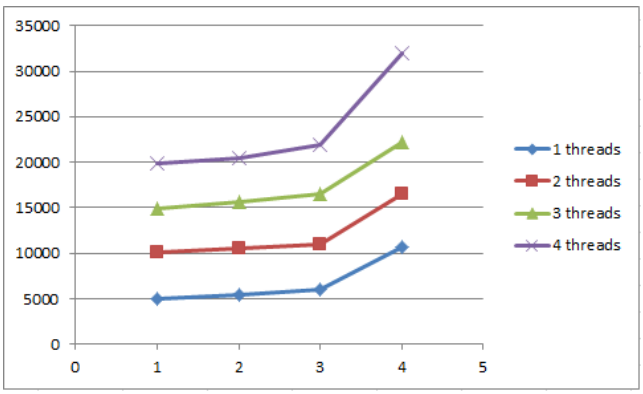

Figure 5: Dynamic Interleave Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of um^2.

6. AutoFAME

6.1 FAME Introduction

FAME, or FPGA Architecture Model Execution is a system for efficiently emulating digital circuits on a FPGA introduced in the *A Case for FAME: FPGA Architecture Model Execution paper* [11]. The paper introduces a system in which the concept of the emulated digital circuit, hence known as the target machine, is separated from the concept of the digital circuit that does the emulation of the target machine, hence known as the host machine. By extension, in FAME, the concept of time passing in the target machine, hence known as target time, is separated from the concept of time passing in the host machine, hence known as host time.

In naive FPGA emulation, the target machine and the host machine are the same digital circuit. the RTL specification of the target machine is mapped directly to an FPGA through vendor tools with no change in the logic design. Because the characteristics of a FPGA are sometimes significantly different from the characteristics of a ASIC in timing and area characteristics, it is desirable to use a modified implementation of the design for emulation on a FPGA. In an mod-

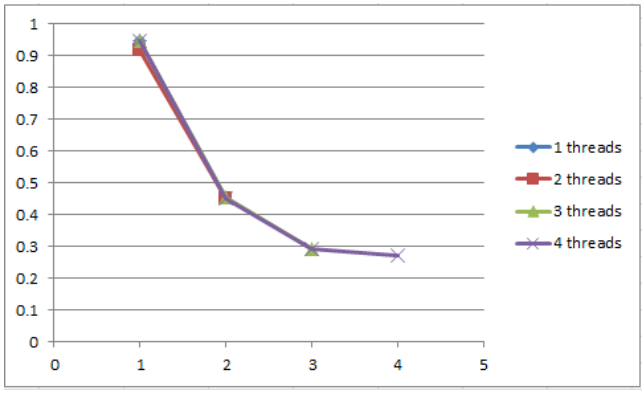

Figure 6: Fixed Interleave Cycle Time The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of ns. The data points where number of stages > number of threads are not valid design points for fixed interleave, so they do not exist on this chart.

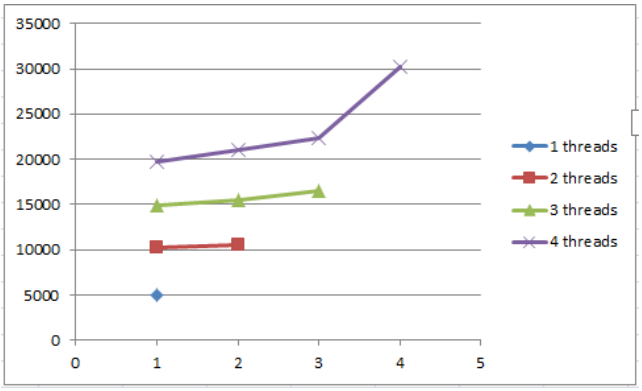

Figure 7: Fixed Interleave Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of um^2. The data points where number of stages > number of threads are not valid design points for fixed interleave, so they do not exist on this chart.

ified implementation optimized for an FPGA, the host machine maybe different from the target machine and it may take more than one or less than one host clock cycle emulate one target clock cycle.

Using FAME, large digital designs can be broken up into modules that talk to each other in a decoupled manner. The partitioned system maintains the same target time behavior as the original design even though the modules are communicating in a decoupled manner. Once the original design is partitioned into decoupled modules, modules can be individually optimized for FPGA emulation, with no restrictions on the host time to target time relationship in each module, while preserving the same target time behavior of the whole system as the target time behavior of the original design.

The original design to be emulated is called a FAME0 level design. The original design should be viewed as a set of FAME0 level modules connected together by registers and

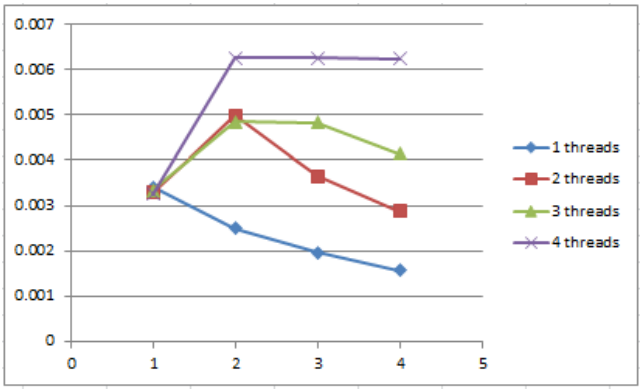

Figure 8: Dynamic Interleave Task/Cycle The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of $task/cycle$.

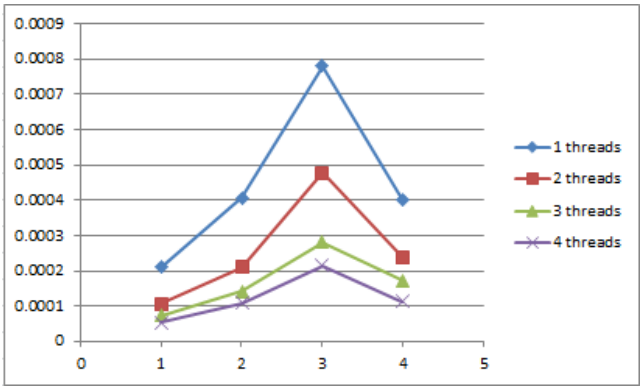

Figure 9: Dynamic Interleave (Cycle/Time)/Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of GHz/um^2.

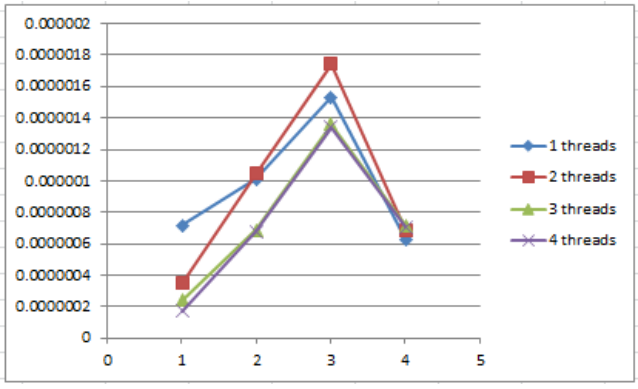

Figure 10: Dynamic Interleave Throughput/Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of $(task/ns)/um^2$.

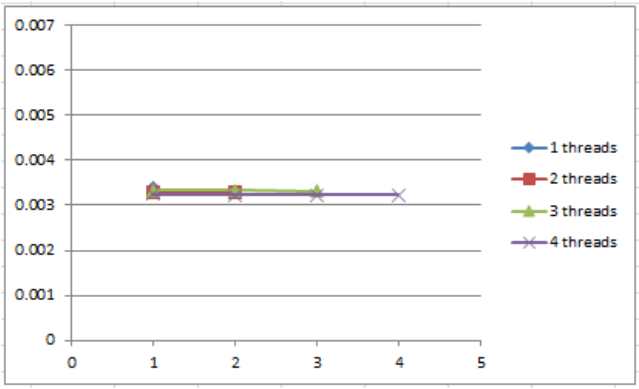

Figure 11: Fixed Interleave Task/Cycle The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of $task/cycle$. The data points where number of stages > number of threads are not valid design points for fixed interleave, so they do not exist on this chart.

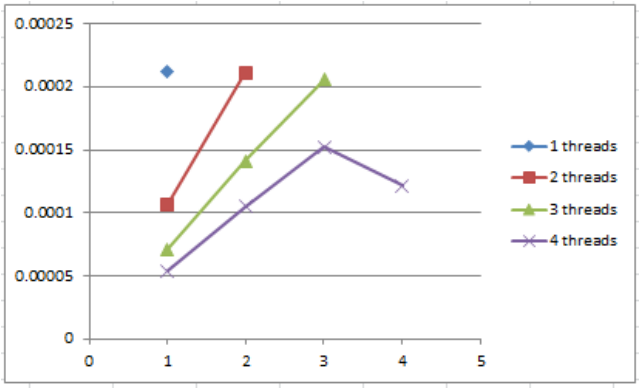

Figure 12: Fixed Interleave (Cycle/Time)/Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of GHz/um^2. The data points where number of stages > number of threads are not valid design points for fixed interleave, so they do not exist on this chart.

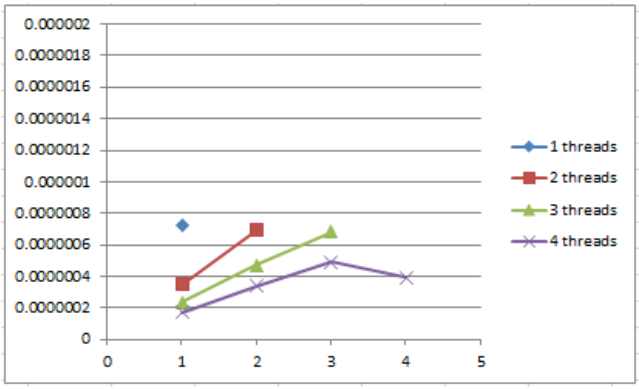

Figure 13: Fixed Interleave Throughput/Area The X-Axis indicates the number of pipeline stages. The Y-Axis is in units of $(task/ns)/um^2$. The data points where number of stages > number of threads are not valid design points for fixed interleave, so they do not exist on this chart.

queues. A FAME0 module naively modified to work as a module that can be inserted into the partitioned system of decoupled modules is called a FAME1 level module. Figure 14 shows a FAME0 design being transformed into a FAME1 design. A module that can be inserted into the partitioned system of decoupled modules and emulates a FAME0 module in an abstract manner, such as with a split functional model/timing model implementation of the FAME0 module, is called a FAME3 level module. A single module that can be inserted into the partitioned system of decoupled modules and emulates n copies of a FAME0 module through multithreading is called a FAME5 level module.

6.1.1 FAME Design Partitioning Details

Target time behavior is maintained in the partitioned system in the following manner. The original design is partitioned by placing module boundaries across registers and queues in the target machine. The target machine registers are replaced by FAME Registers and the target machine queues are replaced by FAME Queues. The FAME Register is a FIFO containing tokens that represent the state of the target register at particular target clock cycles. Tokens further ahead in the FIFO represent the state of the target register at earlier target clocks. The FAME Queue is a FIFO containing tokens that represent the state of the target queue at particular target clock cycles. Tokens further ahead in the FIFO represent the state of the target queue at earlier target clocks. Both FAME Registers and FAME Queues are initialized with one token.

It is important to separate these tokens, which represent one target clock cycle's worth of information about the target register or target queue, from the entries in the target queues. Enqueuing/dequeuing tokens from FAME Registers/FAME Queues will be referred to as host enqueue/host dequeue and the host enqueue/dequeue operations will be performed through manipulating host ready and host valid signals. If a FAME Register/FAME Queue does not contain any tokens, it will be referred to as host empty and if a FAME Register/FAME Queue cannot accept any more tokens, it will be referred to as host full. In contrast, enqueueing/dequeue entries from the target queues will be referred to as target enqueue/target dequeue and the full/empty status of the target queues will be referred to as target full/target empty.

For every advance of the target clock, each module consumes a token from its input FAME Registers/FAME Queues and outputs a token to its output FAME Registers/FAME Queues. A module may not advance its target clock if any of its inputs are host empty or if any of its outputs are host full. Since FAME Registers and FAME Queues are FIFOs of tokens, tokens are allowed to accumulate within FAME Registers and FAME Queues. Thus, the target clock can advance in a decoupled manner while still remaining functionally the same as the original design.

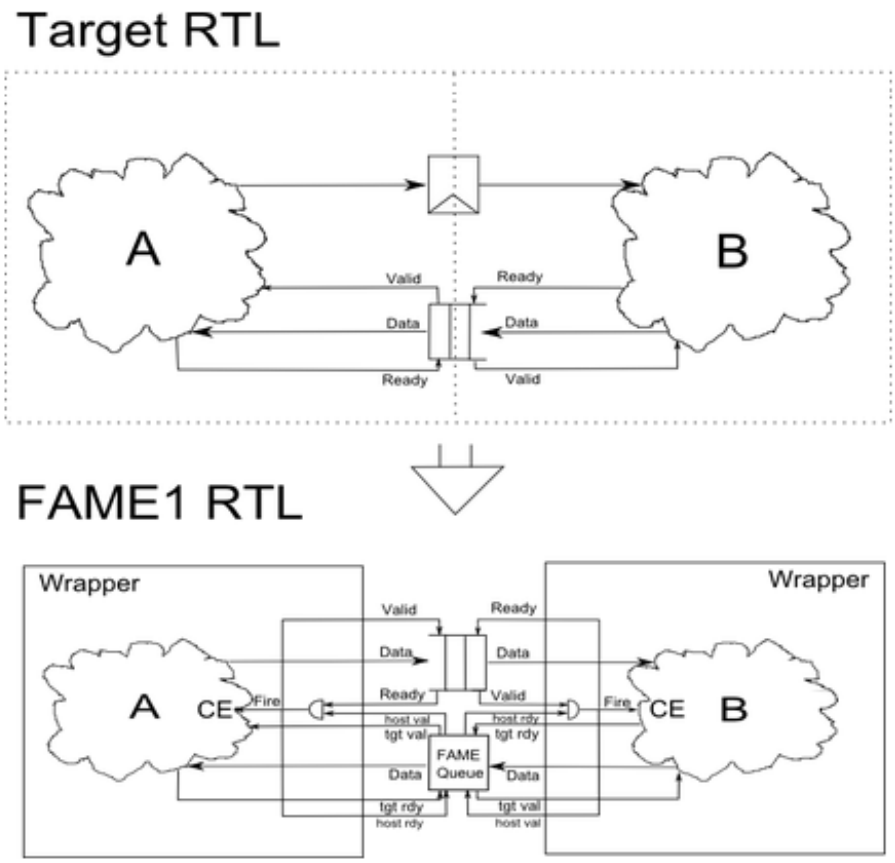

Figure 14: Transforming a FAME0 Design into a FAME1 Design

6.1.2 AutoFAME Features

AutoFAME is capable of creating FAME1 and FAME5 level FPGA optimized designs given a base functional datapath specified in Chisel. This is equivalent of automatically transforming a FAME0 level design into a FAME1 or FAME5 level design. The tool performs the required circuit transformations on the Chisel internal node graph and leverages Chisel's elaboration steps to output the optimized design as either a cycle accurate C++ emulator or as a Verilog source file.

6.2 Input Datapath Restrictions

Because the original design to be emulated should be split by having module boundaries placed across registers or queues, the input FAME0 module should have IO ports of the type RegIO, which consists of a single data pin and indicates that the IO port should be connected to a register in the original design, and QueueIO, which consists of a ready pin, a valid pin, and a data pin, and indicates that the IO port should be connected to a queue in the original design.

6.3 FAME1 Transform

Given any FAME0 module that follows the restrictions in 6.2 and a user annotation in the Chisel source file containing the module instantiation of the input FAME0 module that a FAME0 to FAME1 transformation should be applied, AutoFAME will produce a FAME1 version of that module.

Automatically transforming a FAME0 level design into a FAME1 level design is useful because it allows the FAME0 level design to be interfaced with FAME3 or FAME5 level designs at the cost of no extra work by the designer.

6.3.1 Transformation

In order to make a FAME0 module work in the partitioned system of decoupled modules, its RegIOs need to be converted into FAMERegIOs, which attach a host ready and a host valid pin to the RegIO port in order to interface with the FAME Registers. Its DecoupledIOs also need to be converted into FAMEDecoupledIOs, which also attach a host ready and a host valid pin to the DecoupledIO in order to do host enqueue/dequeues.

Then every state element write enable signal is masked with a fire target clock signal. The fire target clock signal is driven low whenever any of the input FAME Reg-

isters/FAME Queues are host empty or any of the output FAME Registers/FAME Queues are host full.

Then combinational logic is generated to host dequeue the input FAME Registers/FAME Queues and enqueue the output FAME Registers/FAME Queues whenever fire target clock is high.

6.3.2 Example Application

Automatic FAME0 to FAME1 transformation was used to interface a FAME0 level high performance research RISC processor used by UC Berkeley's ASPIRE Lab with a FAME3 level hardware DRAM model for FPGA emulation. The hardware FAME3 level DRAM model is necessary to get accurate results for the processor in FPGA emulation because the relative DRAM to processor clock rate on a FPGA is much higher than the relative DRAM to processor clock rate on a ASIC. In order to obtain performance figures accurate to the ASIC implementation in emulation, the FAME3 hardware DRAM model is used as a intermediary between the processor and the FPGA DRAM and makes the FPGA DRAM appear slower to the processor. Additionally, the FAME3 hardware DRAM model can be adjusted to simulate a variety of DRAM configurations, which would not be possible if the processor interfaced directly with the FPGA DRAM.

6.4 FAME5 Transform

Given any FAME0 module that follows the restrictions in 6.2 and a user annotation in the Chisel source file containing the module instantiation of the input FAME0 module that a FAME0 to FAME5 transformation should be applied along with a specification of how many threads there should be, AutoFAME will produce a FAME5 version of that module.

Automatically transforming a FAME0 module into a FAME5 level design is useful because it allows the designer to conserve area usage on the FPGA if the FAME0 level design is instantiated many times as the multi-threading only replicates the state elements and not the combinational logic in the FAME0 design. Additionally, the multi-threading allows external memory access latencies to be hidden.

6.4.1 Transformation

First, all of the IO ports and the state elements are replicated n times, where n is the user specified number of threads.

Then, like the FAME0 to FAME1 transformation, the RegIOs and Decoupled IOs the input FAME0 design are replaced with FAMERegIOs and FAMEDecoupledIOs.

A IO Ready signal and a Thread Selected signal is generated by each thread. The IO Ready signal of thread m is driven high when all of the input ports associated with thread m are not host empty and all of the output ports associated with thread m are not host full. The Thread Selected signal for thread m is high when the thread select id generated by the thread scheduler equals m.

A fire target clock signal is created for each thread. Thread m's fire target clock signal when thread m's IO Ready signal is high and thread m's Thread Selected Signal is high. The write enables of all the state elements associated with thread m are then masked with the fire target clock signal of thread m.

7. Related Work

Older work in the area [4][6][5][3][10] generally focus on automatically verifying pipelined designs based on a given ISA or describe automatic pipelining system that still require manual intervention on part of the pipelining process.

Nurvitadhi et al [9] present separate tools T-spec for transactional datapath specification and T-piper for automatic pipeline synthesis. T-spec is used to describe transactional datapaths as state elements and acyclic next-state logic blocks that updates those state elements. Users manually annotate all the state elements and next-stage logic blocks with the pipeline stage number. T-piper analyses the T-spec design to identify RAW hazards and generate hazard resolution logic. [8] extends the above tool to be able to generate multi-threaded in-order pipelines. AutoMArch is different in that the pipeline specification does not require a entirely separate language. Additionally, my tool is capable of automatically assigning datapath components to pipeline stage numbers, which saves a lot of designer specification and potentially produces more balanced pipelines.

Galceran-Oms [2] present a method to automatically pipeline synchronous elastic systems. The paper presents a set of provably correct transformations on synchronous elastic systems and show that it is possible to pipeline a micro architecture by applying a sequence of such transformations. AutoMArch is different than the system presented here because it applies pipelining and multi-threading optimizations to ordinary finite state machines with a few IO restrictions rather than to synchronous elastic systems.

8. Conclusion and Future Work

In this thesis, I presented a system for digital circuit frontend specification named Automatic Functional Datapath Optimization. In this system, the designer specifies a base functionally correct datapath without any optimizations applied in a RTL like manner and then selects optimization techniques for automatic tools to apply to the base functional datapath. This system of specification is a middle ground approach between RTL specification and HLS specification and tries to capture the conflicting pros of both approaches, namely high designer productivity and the ability to generate highly efficient designs in terms of performance, power, and area at the same time. I implemented two examples of such a system, AutoMArch and AutoFAME, and presented example applications of both.

I hope that AutoMArch can be extended to support more general speculation and that a more general form multi-

threading, where combinational logic is replicated as well as the state elements can be explored. I also hope that work can be done to catalogue all of the commonly used digital circuit optimization techniques so that they may one day be implemented automatically

9. Acknowledgements

I developed the beginnings of the system presented in this thesis as part of a class project along side of Huy Vo. I would like to acknowledge his work in creating the initial version of the automatic pipelining tool. I would also like to acknowledge my advisers Jonathan Bachrach and Krste Asanović for providing me with guidance and support for this thesis. This research is supported by DoE Award DE-SC0003624, and by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

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] M. Galceran-Oms. *Automatic Pipelining of Elastic Systems*. PhD thesis, UNIVERSITAT POLITCNICA DE CATALUNYA, 2011.

[3] J. Higgins and M. Aagaard. Simplifying the design and automating the verification of pipelines with structural hazards. *ACM Trans. Design Automation of Electronic Systems*, 2005.

[4] D. Kroening and W.Paul. Automated pipeline design. *Proc.ACM/IEEE Design Automation Conf.*, 2001.

[5] M.-C. V. Marinescu and M. C. Rinard. High-level automatic pipelining for sequential circuits. *Proc. Int, Symp. on Systems Synthesis*, 2001.

[6] J. Matthews and J. Launchbury. Elementary microarchitecture algebra. *Lecture Notes in Computer Science*, 1999.

[7] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. In *Proceddings of the IEEE*, 1990.

[8] E. Nurvitadhi, J. C. Hoe, T. Kam, and S. L. Lu. Automatic multithreaded pipeline synthesis from transactional datapath specifications. In *Design Automation Conference*, DAC '10, .

[9] 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, .

[10] A. K. P. Mishra. Synthesis-driven exploration of pipelined embedded processors. *Proc. Int. Conf. VLSI Design*, 2004.

[11] T. Z., W. A., C. H. M., B. S., A. K., and P. D. A case for fame: Fpga architecture model execution. In *ISCA*, 2010.