Copyright © by

Yuxiong Zhu

2017
# Table of Contents

List of Figures v

Abstract vi

Dedication vii

Acknowledgments viii

1 Introduction 1

2 The Live Graph Structure 3

3 Dead Code Elimination 5
  3.1 The Algorithm 5
  3.2 Contributions 6

4 Global Value Numbering and Partial Redundancy Elimination 7
  4.1 Global Value Numbering (GVN) 7
  4.2 Partial Redundancy Elimination (PRE) 8
  4.3 GVN-PRE 11
  4.4 HDL GVN-PRE
     4.4.1 Observations 12
     4.4.2 The Algorithm 16
  4.5 Contributions 19

5 Fluid Pipelines 21
  5.1 Double Buffering 21
     5.1.1 Stalling 21
     5.1.2 The solution: Double Buffering 23
  5.2 Elastic Circuits 26
  5.3 Fluid Pipelines 28
  5.4 Comparing Fluid Pipelines to Elastic Systems 32
5.5 Deadlock Avoidance In Fluid Pipelines .......................... 34
5.6 The Algorithm ....................................................... 35
  5.6.1 Find Joins ..................................................... 35
  5.6.2 Find Forks ..................................................... 37
  5.6.3 Add Operators ................................................. 39
5.7 Contributions ....................................................... 41

6 Conclusion ............................................................. 42

Bibliography ............................................................. 43
List of Figures

4.1 Example before PRE .................................................... 9
4.2 Example after PRE ...................................................... 9
4.3 Example of Second Scenario of PRE .................................. 10
4.4 Example of PRE in SSA ................................................. 11
4.5 Synthesized logic before and after optimization ....................... 13
4.6 Original Synthesized logic ............................................... 16
4.7 "Optimized" Synthesized logic using Option 1 ......................... 17
5.1 A Stall Implementation .................................................. 22
5.2 Double Buffering ......................................................... 24
5.3 Double Buffering Control Logic ....................................... 25
5.4 Elastic System Example .................................................. 27
5.5 Elastic Buffer ............................................................. 27
5.6 Retiming Example ......................................................... 28
5.7 Recycling Example ....................................................... 28
5.8 Fork Operator ............................................................. 29
5.9 Join Operator (with an adder) ........................................... 30
5.10 Branch Operator ......................................................... 31
5.11 Merge Operator .......................................................... 31
5.12 Toy case to compare Elastic and Fluid Pipelines ....................... 32
5.13 Execution traces of Elastic System vs Fluid Pipelines ............... 33
5.14 Deadlock avoidance example in Fluid Pipelines ....................... 35
5.15 Fluid Pipelines Conversion Example ................................... 37
Abstract

The Live Graph

by

Yuxiong Zhu

Live Graph (lgraph) is a graph optimized for live synthesis (Live Synthesize Graph or lgraph for short). By live, we mean that small changes in the design should have results in few seconds. The goal is that any code change can have its synthesis and simulation setup ready under 30 seconds with a goal of under 4 seconds in most cases.

Notice that this is a different goal from having an incremental synthesis where many edges are added and removed. The typical graph reconstruction is on the order of thousands of nodes.

lgraph is optimized for synthesis, allowing forward and backward traversals in the nodes (bidirectional graph).

Once the graph is built, some optimization can be applied to the input design, including Dead Code Elimination (DCE), Common Sub-expression Elimination (CSE), Copy Propagation, Fluid flop insertion and so forth.
To my wife, parents and in-laws who took care of my baby daughter Gloria so I could finish this thesis.
Acknowledgments

I would like to thank Rafael and my adviser, Professor Jose Renau, who initiated and built the foundation of Lgraph. They also provided me great help on solving problems in this project.

Also thanks Professor Jishen Zhao and Professor Martine Schlag who gave me valuable feedback and suggestions.
Chapter 1

Introduction

Logic Synthesis is an integral part of the ASIC/VLSI Design Flow which maps Hardware description language (HDL) code to digital circuits/components on Field Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs) [14]. Logic Synthesis is one of the signature steps that separates Register Transfer Level (RTL) design from software design.

The first step of RTL Synthesis is generally called elaboration, this includes reading in the RTL file and translating the design to a form that facilitates later steps [8][10].

In lgraph, the design being read is translated to a graph which consists of nodes and edges. By transferring verilog code to a graph form, the characteristics of the design are collected during the process. In graph form, a design is convenient to analyze and modify.

Optimal human written code is hard to produce because sub-optimal code may function correctly. For example, a line of code may be unused or repetitive. This problem has been addressed a long time ago in software programs, but it has greater impact in RTL design: the extra lines of code imply extra logic gates and thus area on chip if they are not recognized and handled properly. Thus, various optimization algorithms are often used in synthesis. In lgraph, common algorithms such as Dead Code Elimination
(DCE), Common Sub-expression Elimination (CSE), Copy Propagation can be used to optimize a design.

In addition, some useful techniques may be tedious to implement manually, inserting Fluid Flops \cite{16}\cite{17} for example may take a great deal of work. However, if the insertion can be done automatically, the technique will be much more easily adopted and evaluated. For this purpose, Fluid flop insertion is integrated in lgraph and can be optionally done when optimizing the design.

The elaboration function was implemented by other colleagues in the lab, so only an overview and related content are covered in this thesis. The optimization functions are the main contribution of the author and thus the focus of this thesis.
Chapter 2

The Live Graph Structure

The graph structure is based on synthesis graph requirements. Each conceptual graph node has many inputs and outputs as in a normal graph, but the inputs and outputs are numbered. For example, a node can have 3 input ports and 2 output ports. Each of the input and output ports can have many edges from/to other graph nodes.

Each graph edge is between a specific graph node/port pair and another node/port pair. The graph supports adding meta-information on each node and node/port pair. The port identifier is an integer OK up to 1024 (10 bits) values per node. In the code, the port is identified by a Port_ID.

The graph is built over a table structure. Each table entry is 64 bytes and contains full or part of a graph node’s information. To access the information, we use the table entry number of Index_ID.

When a new node is added to the graph a new Index_ID is generated. The node always has a Index_ID for the port zero, different Index_ID for other node/port pairs, and potentially additional Index_ID for extra storage to keep the graph edges. Each Index_ID can be used to store and maintain information in additional tables such as the delay, or operation, but in reality we only store information for the whole node or for each node/port pair.
The Index_ID that uniquely identifies the whole node is called Node_ID in lgraph. This is typically accessed with methods such as get_nid().

The Index_ID that uniquely identifies a node/port pair is called Outp_ID (Output Pair ID). This is typically accessed with methods such as get_oid(). The Node_ID and the Outp_ID is the same number when the port is zero.

When traversing the edges in the graph, it is possible to ask for:

- `get_nid` # Node_ID, Index_ID that uniquely identifies the node
- `get_oid` # Outp_ID Index_ID that uniquely identifies the node/port pair
- `get_idx` # Index_ID raw index pointer where the info is stored
- `get_inp_pid` # Port_ID Input port for this edge
- `get_out_pid` # Port_ID for the output port driving this edge

These IDs have the following meaning:

- Index_ID # 37 bit index. Either a Outp_ID, Node_ID, or additional storage
- Outp_ID # 37 bit index, uniquely identifies a node/port pair
- Node_ID # 37 bit index, uniquely identifies a node
- Port_ID # 10 bits, per node input/output port identifier

A graph edge does not have a unique id. lgraph does not support meta-information for generic edges. It can store meta-information for Outp_ID. This is different than the edge because the same node/port output can have many destinations and all have to share the same meta-information. This would be an issue if we want to store information such as resistance/capacitance or distance per edge. If this becomes necessary, a potential solution would be to modify the graph so that at most one output edge is inserted for each Index_ID.
Chapter 3

Dead Code Elimination

In a software program, a variable is defined as *live* at a point if its value is used afterward; otherwise, it is defined as *dead* at that time [1]. A related concept is *dead code* which is a section in the source code that compute values that never get used in any other computation. Although no programmer is likely to introduce any dead code intentionally, it does exist in many programs.

3.1 The Algorithm

Dead Code Elimination (DCE) in the HDL domain is very similar to traditional DCE, however it is special in that any signals not used for calculating the output port of the module are considered as *dead*. This is where the graph form becomes very useful; we can just traverse the graph from the outputs going backward, looking for nodes that are not used in the output generation. These nodes are marked for deletion. The node itself is not deleted, but all the edges which leave the node disconnected (not seen in traversal).

The pseudo code of the implementation of the algorithm is shown below:

Create bit array output_used
For the index idx of each operation node in a backward traversal of the graph

set_bit(idx) in output_used

for all In edge in_edge of idx do
    set_bit in_edge$get_idx in output_used
end for

For the index idx of each operation node in a forward traversal of the graph

if output_used[idx] == Null then
    for all In edge in_edge of idx do
        Delete in_edge
    end for

    for all Out edge out_edge of idx do
        Delete out_edge
    end for
end if

3.2 Contributions

• Traditional DCE is introduced.

• The DCE in hardware context is given.

• The algorithm of HDI DCE is provided.
Chapter 4

Global Value Numbering and Partial Redundancy Elimination

4.1 Global Value Numbering (GVN)

Global value numbering (GVN) is a common compiler optimization [4]. GVN assigns a value number to variables and expressions so that the same value number is assigned to those variables and expressions that are equivalent. The value number could be a structure, a hash key or literally a number, as long as each value number is unique. That way a later discovered operation is identified as redundant if it has the same global value number as an earlier stored expression, and this operation can be eliminated.

One big virtue of GVN is that it does not depend on lexical equivalences, whereas traditional CSE or PRE matches lexically identical expressions [20]. So sometimes GVN can discover redundancies that can’t be identified by other algorithms. This is illustrated in Listing 4.1.

Listing 4.1: GVN Example

a = 2
\[ b = 2 \]
\[ y = a + 3 \]
\[ z = b + 3 \]

A good GVN algorithm can detect that \( a \) and \( b \) have the same value, and \( y \) and \( z \) have the same value. Thus it would assign the same value number to \( a \) and \( b \), and the same value number to \( y \) and \( z \). So the redundancy is eliminated. The result is shown in Listing 4.2.

Listing 4.2: GVN Example Result

\[ a = 2 \]
\[ b = a \]
\[ y = a + 3 \]
\[ z = y \]

However, traditional GVN is usually restricted to only basic blocks [20]. Because traditional GVN does no code motion, it does not eliminate some operations that are either fully or partially redundant on some branches. For example, traditional GVN is not able to find the redundancy in Figure 4.1 which can be easily handled by PRE.

### 4.2 Partial Redundancy Elimination (PRE)

Partial redundancy elimination (PRE) is another compiler optimization, PRE eliminates an expression that is redundant on some but not necessarily all paths through a program that include that operation [2]. This kind of operation is defined as partially redundant. PRE is a form of common subexpression elimination (CSE) [9].

In order to eliminate partial redundancy, PRE inserts the partially redundant operation on the paths where originally the operation is not computed. The result is that all
paths now include the operation, thereby making the partially redundant operation fully redundant.

The strength of PRE is clear: it can handle partial redundancy which is difficult for other algorithms to eliminate.

Although not generally discussed in other articles, there are two slightly different scenarios of PRE. The first scenario is simple: all branches of a merge point don’t have a common destination variable, an example is given in Figure 4.1. The destination variable $y$ only appears in the left branch.

The expression $a + b$ assigned to $z$ is partially redundant because it is computed twice if the left branch is taken. PRE can hoist the operation $a + b$ to the right branch where originally the operation was unknown. So a good PRE would perform code motion on the expression to yield the optimized code as shown in Figure 4.2. By hoisting and preserving the result of the operation $a + b$ in temporary variable $t$, PRE can eliminate the redundancy at $z$ which exists in the original code.
Figure 4.3: Example of Second Scenario of PRE.

The second scenario is more complicated compared to the first one: some branches of a merge point may have a common destination variable, an example is given in Figure 4.3. The destination variable $a$ appears in both branches. Here the expression $a + b$ is also partially redundant because it is computed twice if the left branch is taken.

The PRE algorithm in static single assignment form (SSA) can be used to handle this scenario [19]. First we transfer the original code into its SSA form as shown in Figure 4.4a. Note SSA uses the $\Phi$-function [1] to combine the two definitions of $a$. $\Phi(a1, a2)$ has the value $a1$ if the left branch is taken and the value of $a2$ if the right branch is taken.

The result of PRE is shown in Figure 4.4b. Similarly PRE can hoist the operation $a + b$ to the right branch where originally the operation was not computed. So by hoisting and preserving the result of the operation $a + b$ in temporary variables $t1$ and $t2$, PRE can eliminate the redundancy at $z$ which exists in the original code.

These two scenarios are not so different here as they are in the HDL context, as we will see in Section 4.4.

The main drawback of traditional PRE is that it is based on lexical equivalences [20]. That means the set of operations computing the same value but differing lexically can not be identified. Hence traditional PRE cannot identify the redundancy in Listing 4.1.
4.3 GVN-PRE

One observation is that GVN and PRE are functionally complementary; neither one can completely replace or include the other but they can eliminate some operations the other one can’t. So it is natural to come up with the idea of combining these two algorithms. Thomas VanDrunen proposed an effective algorithm which combines GVN and PRE in his PhD dissertation [21]. His GVN-PRE is a powerful value based Partial Redundancy Elimination algorithm for programs in static single assignment form (SSA). It extends dominator based global value numbering to eliminate partially redundant operations.

The GVN-PRE algorithm has three steps: BuildSets, Insert, and Eliminate. Step 1, BuildSets, provides the foundation for the other steps. While it is analyzing the program, it forms the value table and the leader and antileader sets. The expressions which compute the same value are assigned by the same leader, this leader is effectively the value number in GVN terms. Step 2, Insert, places new operations in the program to make partially available instructions fully available. This step converts the partially redundant operations into fully redundant ones. Step 3, Eliminate, removes operations whose values are already available. This is done by traversing the program, for any operation, finding the leader of the target’s value in the value table. If the leader differs
from that target, then there is an earlier computed variable with the same value. Thus the target could be replaced by the leader.

4.4 HDL GVN-PRE

4.4.1 Observations

In the context of an HDL, things become quite different. The first observation is that the Φ in SSA form becomes a MUX operation, also the logics on all branches always exist even though they should not depend on each other. This is a significant difference from the software context since in software when a certain branch is taken, only the operations on that branch are visible after the merge point. That is the main reason why GVN cannot handle partial or fully redundancy. Whereas for HDL, the logics on all branches are visible after the merge point i.e. the MUX. Thus GVN in HDL context can eliminate some cases of partial redundancy. One example is shown in Listing 4.3. This is actually one example of PRE scenario 1 where the destination register \( d \) appears only in one branch, also \( a \) and \( b \) are not assigned in any branch. Note \( a \& b \) is redundant at \( z \), because it’s calculated in one branch and assigned to \( d \). An HDL GVN can detect this and then eliminate the AND gate at \( z \). The original and optimized synthesized logic is given in Figure 4.5.

Listing 4.3: GVN in HDL example

```verilog
module mux_pre1 ( 
  input sel , a , b ,
  output z
);

reg c , d;
```
always @ (*) begin
    d = a;
    if (sel)
        d = a & b;
        c = a;
    else
        c = b;
    z = a & b;
end
endmodule

In addition, the logic on the branches could be made visible to each other if desired even if they were not originally visible to each other. The Verilog code in Listing 4.4 below demonstrates this point. In this example, a MUX is inferred because of the if-else statement. The select pin of the MUX is connected to the input port sel, and the output of MUX is connected to the output port z. The value of z is either branch B1:(a AND b) or branch B2:(a AND b OR c) depending on the value of sel. Note (a AND b) is a

![Diagram](image1)

(a) Original Synthesized logic.

![Diagram](image2)

(b) Synthesized logic after optimization.

**Figure 4.5:** Synthesized logic before and after optimization.
common sub-expression shared by the two branches of the MUX, so it can be removed in B2 if we add an edge from the output of B1 to B2. Thus we can eliminate one AND gate. This is something that cannot be done in software code compilation.

Listing 4.4: MUX example

```verilog
module mux_vs_phi (
    input sel, a, b, c,
    output z
);

always @ (*) begin
    if (sel)
        z = a & b;
    else
        z = a & b | c;
end
endmodule
```

The second observation is about overhead; in hardware the goal of an elimination algorithm is to reduce the total number of gates. So if any optimization step introduces more gates after all, it is better to avoid optimizing that part. Listing 4.5 is one good example of the second scenario of PRE that we discussed before. Here a is assigned a new value in both branches, thus we can’t directly reuse the AND gate for z. There are two options, the first one is that we optimize it in the same way as in software PRE. That means we insert a new AND gate which calculates a & b in the else branch. Then we can safely eliminate the AND gate at z. However, a MUX must be inserted for the z to choose different versions of a & b. The second option is that we do not try eliminate this redundancy; that is we do nothing for the second scenario of PRE.
Listing 4.5: MUX example 2

```verilog
module mux_example ( 
    input sel, a, b, c, d, 
    output z
);

reg e, f;

always @(*) begin
    if (sel)
        e = c;
        f = e & b;
    else
        e = d;
        f = a;
    z = e & b;
end
endmodule
```

The original synthesized logic of Listing 4.5 is given in Figure 4.6, there are two AND gates and two MUXs. If we try optimize it using the first option as we discussed, the result is shown in Figure 4.7. The highlighted AND gate is inserted to the other branch, this way we can remove the final AND gate which was originally connected to $z$. As we discussed before, a MUX must be inserted before $z$ so that $z$ can choose the different results from the two AND gates. One straightforward metric to evaluate the hardware complexity is gate counts. When we count the gates in the optimized logic we can see that there are still two AND gates and two MUXs. That means this optimization
does not improve anything at all. That is the reason we chose to use the second option in this situation; we ignore the redundancy in scenario 2 of PRE.

### 4.4.2 The Algorithm

In our HDL GVN-PRE algorithm, there are also three steps: BuildSets, Insertion, and Elimination. There are three maps needed for the algorithm: expression leader map, operand leader map and index-replace-map. Each expression is assigned a leader node pin in the expression leader map, the expressions that compute the same value are assigned the same leader node pin. Similarly, the operand leader map is used to map result operands. Each result operand is assigned a leader node, the result operands that compute the same value are assigned the same leader node pin in operand leader map. The third map, index-replace-map, stores the index of a node which can be replaced by its leader.
**Figure 4.7:** "Optimized" Synthesized logic using Option 1.

**BuildSets**

In this step, we forward traverse the graph and we process one node at a time. For each node, there could be multiple input operands. For each input operand, we lookup in the operand leader map to see if it has a leader, then its leader is returned. We assert that there must be a leader for an input operand since an input operand must come from somewhere and if an input is not connected then that input operand will not be seen during the traversal. After the leaders of all input operands are collected, we have the new expression for that node where all input operands are replaced by the associated leaders. Here if the node is an operation that has the commutative property such as AND, OR, XOR, we sort the leader operands by their node pin id. This way we can preserve the commutative property for that operation. Next we lookup in the expression leader map to check if the new expression was defined. If so the leader node pin is returned, and this node is marked replaceable by adding it to the index-replace-map. If there is no leader for the new expression, we create a leader in the expression leader map by mapping the new expression to the result node pin of the current node.

The pseudo code of the algorithm is given below:

For the index idx of each operation node in a forward traversal of the graph
for all Source operand node pin op_np of idx do
    op_leader ← operand_leader_map[op_np]
    Push back op_leader to new_expression
end for
if Node idx is commutative then
    Sort new_expression
end if
new_expression_leader ← expression_leader_map[new_expression]
if new_expression_leader != Null then
    index_replace_map[idx] ← new_expression_leader
else
    expression_leader_map[new_expression] ← dst_np
    ▷ dst_np is the destination node pin of the current node
end if

Insertion

For Insertion, we need to iterate the index-replace-map. For each node in the index-replace-map, we grab its leader. Then for each out edge from the node, we add a new edge from the leader node pin to the original destination node pin.

The pseudo code of the algorithm is given below:

For the index idx of each operation node in a forward traversal of the graph
leader_np ← index_replace_map[idx]
if leader_np != Null then
    for all Out edge out_edge of idx do
        Get the destination node pin dst_np of out_edge
        Add edge from leader_np to dst_np in the graph
end for
end if

Elimination

The final step is more straightforward: we eliminate all the replaceable nodes in the graph. This cannot be done directly since removing a node without removing its edges will cause problems. So for each node in the index-replace-map, we remove all in edges and out edges of the node. By doing this we effectively eliminate the nodes.

The pseudo code of the algorithm is given below:

For the index idx of each operation node in a forward traversal of the graph

\[
\text{leader}_{\text{np}} \leftarrow \text{index}_{\text{replace}}_{\text{map}}[\text{idx}]
\]

if leader_{np} != Null then
    
    for all In edge in_edge of idx do
        
        Delete in_edge
    
    end for

    for all Out edge out_edge of idx do
        
        Delete out_edge
    
    end for

end if

4.5 Contributions

- Traditional PRE and GVN are introduced.
- Several observations of GVN-PRE for hardware are given.
- Examples are shown to help understanding.
• The algorithm of HDI GVN-PRE is provided.
Chapter 5

Fluid Pipelines

5.1 Double Buffering

Double buffering is an important technique used in Fluid Pipelines systems. It is originally introduced for solving the timing issues of stall signals in deep pipelines [7].

5.1.1 Stalling

An ideal pipeline does not need a stall signal; all pipeline stages are connected smoothly and there are no events that interfere with the operation of the stages. In reality, however, some events such as data hazards may occur and complicate the pipeline. For example, in a computer pipeline, the dependence of one instruction on an earlier one that is still being processed in the pipeline causes data hazards [15]. That means in some situations, stage B depends on the result of stage C which is processing an earlier instruction. Thus stage B cannot proceed until stage C generates the result. Here we need a stall signal sent from stage B to all of its previous stages. Assume stage A is the adjacent stage right before stage B. Without this stall signal, the register from stage A feeding the stalled stage B would still accept new inputs from its earlier stage. The
incoming data would overwrite the previous inputs stored in A. As a consequence, the old task that should be retried later would be lost.

To respond to the stall signal, each pipeline register bank includes logic that will hold its data and not accept new data if its stall is asserted. Downstream, when a stall occurs, we still clock the flip-flops, but deassert valid, a control signal, indicating that the result of the previous stage is invalid. An implementation of a pipeline with stall signal is given in Figure 5.1. Each of the three pipeline register banks has a stall pin at the top. When the stall is asserted, the pipeline register bank will not clock in new data in that clock cycle, thus hold its data. The valid signal in each stage is generated by a two input AND gate. The bottom input indicates whether the last clocked in data set is valid, whereas the top input indicates whether this stage does NOT generate a stall in the current cycle. That means, only if the last data set is valid and this stage is not generating a stall, will the valid signal of this stage be asserted. Note, because the valid signals are separated by the stage registers, they will not be the longest delay path. That means the valid signals are not on the critical path. Assuming a valid signal is generated in stage N, the start point of its path is always from stage register N-1 and the end point is always stage N. So it cannot directly interfere with any signals before stage register N-1 or after stage register N.

In contrast, the stall/stop signals tend to be on the critical path. First, the assertion
of a stall signal usually does not occur until a late pipeline stage, this introduces logic delay. Second, the stall/stop signals form a stop chain as shown in Figure 5.1, this is different than the valid logic. These signals must propagate up to the length of the pipeline and combine with stall signals from other stages. The propagation across stages may incur wire delay, and the OR gates that combine the stall signals have a delay that increases linearly with the pipeline depth. These aforementioned timing concerns restrict the number of pipeline stages that a pipeline can have.

5.1.2 The solution: Double Buffering

Now we know that the problem with a global stall signal is the cascaded delay. In modern pipelines where clock period is frequently less than one nanosecond and a large number of stages exist [15], it is impossible to broadcast the stall signal in one cycle. That is when double buffering comes into play. The idea is that we need to clock the stop signal into a flip-flop at every stage and propagate it upstream to only its previous stage. In order to do this while preserving the old data. We must use double buffer where one buffer holds the old data while the other buffer is accepting the new data arriving when a stall occurs until the upstream stage can be stopped.

An example of a pipeline using double buffering is shown in Figure 5.2. At the end of each stage there is a double buffer which contains two registers. When a pipeline stage \( i \) becomes not ready, it asserts the internal stop signal \( IS_i \). This makes the data \( D_{i-1} \) coming from stage \( i-1 \) to be buffered into the other register in the double buffer of stage \( i \). Note, at this point stage \( i-1 \) is unaware of the stall event because it has not seen the stop signal yet. On the next cycle, stage \( i-1 \) sees that \( PS_i \) is asserted, where \( PS_i \) is the registered value of \( Si \) (So that is why \( PS \) is called the Previous Stop). This causes the stop signal \( Si-1 \) of stage \( i-1 \) to be 1. In this way, the stall signal is propagated upstream one cycle at a time.
The control logic of the double buffer can be implemented using a state machine. Let us define four stages as following: State 1: S=0, PS=0. State 2: S=1, PS=0. State 3: S=1, PS=1. State 4: S=0, PS=1.

Also assume that the first register in the double buffer is RegA and the second one is RegB. State 1 is the normal operation state where no stall event occurs. The new data coming from the previous stage will be clocked into RegA. When suddenly a stage generates a stall signal, it asserts its S and enters state 2. The new data will be written into RegB, while RegA holds old data. Now if the stage generates another stall signal in the next cycle for some reason, it has both S and PS asserted and thus enters state 3. Now the new data will NOT be written into either RegB or RegA, but this is OK because the upstream stage has already received the stall signal from the PS of the current stage. So the upstream stage will hold its old data which is the new data with respect to the current stage. Thus we will not lose any data. When the stall event is cleared at the current stage, it has S deasserted but PS is still asserted. So it will enter state 4. Now the old data at RegA is being processed normally while the buffered data at RegB is written into RegA.

An implementation of the double buffer control logic is given in Figure 5.3. When the stage has not been stalled for the two most recent cycles, both S and PS are 0 (State 1). This causes the new data to be written into RegA through MUX2 (select 0) and MUX3 (select 0) every cycle. In the first cycle during which a stall event occurs and
Figure 5.3: Double Buffering Control Logic.

the stop signal $S_i$ is asserted by either IS or PS from downstream stage (State 2), RegA holds its old data by the feedback through MUX3 (select 1). The RegB buffers data from the upstream stage through MUX1 (select 1). In the subsequent cycles until the stall event at stage $i$ is solved, $S$ and PS are both asserted (State 3). RegA holds its data through MUX3 (select 1) and RegB holds its data through MUX1 (select 0). In the first cycle after the stop signal $S$ has been deasserted, it enters State 4 since $S=0$, PS=1. The data of RegB will be forwarded to RegA through MUX2 (select 1) and MUX3 (select 0). Note the data at RegB is also fed back into RegB itself, but the data are not needed any more. This is because the next state is either state 1 or state 2, neither of them reads the content in RegB.

By using double buffering we can decrease the latency and increase the throughput of the pipelines. By removing the stall signal from the critical paths, the timing issues caused by stall signal are solved. However, the additional registers in the double buffers do impose an area overhead compared with the logic of a global stall signal. So a design with low frequency or short pipelines may favor the global stall signals.
5.2 Elastic Circuits

In traditional digital chip design, pipeline depth and clock cycle period are usually decided early in the design process which is before the synthesis and physical design [16]. But their impact on physical design can only be evaluated after synthesis and timing analysis. The later we need to modify the design the more effort and difficulty will be involved. This is because any modification in a step may cause several iterations in other steps in the design flow [23]. So if we want to adjust the pipeline depth when the implementation is mostly done, it will be impractical to add or remove the number of pipeline stages. Because this would require multiple long iterations each time you make a change. For example, a change may introduce a new bug which should be discovered in the verification step. Or a change may turn a new path into the critical path which could cause a setup time violation.

Elastic Circuits [5] could be a solution in this case. Because they are insensitive to latency, they can support changing the pipeline depth late in the design time with relatively low effort.

Elastic circuits provide the ability to tolerate timing variations in the computations and communications of a circuit and its environment. The "elasticity" here refers to the property of a circuit to respond to the timing requirements to adapt its activity. This elasticity is achievable without throughput penalty. Elastic systems assume that the correctness of the functions in the system is dependent on the order between two subsequent events instead of the latency. This also helps the design phase, because it is a more natural way for people to think and design. This feature provides the flexibility: the insertion of new pipeline stages later in the design steps does not alter the system correctness.

An elastic execution example is given in Figure 5.4 each colored cell in the A and B rows represents the arrival of a valid token to that operand. The empty cells in the
two rows mean that no new data arrived in that cycle. Assume the calculation of the result C of the addition has one cycle latency. The cells with the same shading form a complete data set for the addition. Note that the latency between any events is arbitrary; the correctness of execution only depends on the order of the events.

Elastic circuits are built on an elastic buffer (EB) which is shown in Figure 5.5. They are also referred to as double buffering technique which was originally invented for timing issues caused by stall signals in pipeline as aforementioned.

Retiming [12] can be used in Elastic Systems by moving EBs across combinational logic without affecting the functionality of the design to fix timing issues or improve performance. Retiming does not change the sequential behavior of the circuit, so it can be leveraged in Elastic Systems without penalties. The reason retiming works is based on the following observation. Critical paths (with negative slack) always exist along with their non-critical adjacent paths (with positive slack). The idea is to leverage
positive slack on one side of the register to balance negative slack on the other side \cite{13}. One such example is shown in Figure 5.6. F1 and F2 are two functions which are implemented by combinational logic. After the transformation by retiming, the computation is more balanced and the critical path is much shorter.

The technique "recycling" which changes the number of pipeline stages is also possible in Elastic Systems \cite{18}. One recycling example is given in Figure 5.7. Note the number of registers is changed by recycling. However one constraint of using recycling in Elastic Systems is that it cannot be used in sequential loops. Traditional Elastic Systems depend on an automated flow which transforms regular synchronous circuitry into elastic. Because the flow is unaware of the intended behavior of the circuit, it must maintain the completion order of events. This can reduce the overall throughput of the system. This also significantly reduces the performance on transforming designs with Out-of-Order (OoO) execution.

5.3 Fluid Pipelines

Fluid Pipelines \cite{16} \cite{17} extend traditional Elastic Systems to allow breaking the relative completion order. Because the flow cannot change the functional behavior of a design, Fluid Pipelines rely on designer annotations in the code to indicate where
ordering can be changed. Fluid Pipelines should be considered as a generalization of Elastic Systems, since without user annotations, they behave exactly like traditional Elastic Systems. In order to support OoO execution, we have to know the intended behavior and when the relative completion order between operations does not affect functional correctness. This can be achieved by exposing the Merge and Branch to the designer.

The basic construct blocks of Fluid Pipelines are still Elastic Buffers plus four types of operators: Branch, Merge, Fork and Join. Examples of their implementations are given in Figures 5.8, 5.9, 5.10 and 5.11. All of the operators can be extended to handle more than two inputs or outputs.

When the data output of one stage needs to be fed to multiple output paths, fork is used. The implementation of the fork operator is shown in Figure 5.8. The valid signal from the current stage literally forks to multiple downstream branches. If one of more stop signals are received from the downstream branches, the current stage will generate a stop.

Join is used when parallel data paths join at a point, thus the inputs of a stage come from multiple upstream stages. The use of the join operator is quite frequent, since any operation that has more than one register as its input will need a join operator. A typical example is an adder that requires all the inputs to be present before the execution can
Figure 5.9: Join Operator (with an adder).

start. The implementation of the join operator with an adder example is shown in Figure 5.9. Note that the valid of the current stage is the AND of all its input valid signals; this forces the join operator to wait to proceed until all the inputs are valid.

Branch is an operator that propagates data to exactly one of multiple output paths, this is different than sending data to all the output paths as in Fork. The implementation of the Branch operator is shown in Figure 5.10. The Selection signal (Select) is data dependent; it selects the branch (from multiple output paths) to which the data will propagate.

Merge is used when the execution can proceed as long as at least one of inputs has valid data. As opposed to Join, only data from one of the input paths can be consumed at each cycle. The function of Merge is like an arbiter: multiple sender stages compete for one receiver stage, while the receiver selects one winner of the arbitration whose data is accepted. For simplicity, the proposed implementation in Figure 5.11 has fixed
Figure 5.10: Branch Operator.

Priority and may cause starvation. But when Merge is used, the arbitration scheme can be replaced by any other scheduling scheme, such as Round-Robin.

Note Merge and Branch cannot be automatically inserted like Fork and Join, because they change the relative order between events. So the designer is responsible for

Figure 5.11: Merge Operator.
adding annotations to the code when they want to insert Merge and Branch.

5.4 Comparing Fluid Pipelines to Elastic Systems

To demonstrate the power of Fluid Pipelines, an example is given in Figure 5.12. The circles labeled with F represent combinational logic, the rectangles represent EBs, and the dots inside rectangles mean that EB contains valid data. In Figure 5.12b, any T in the figure is an abbreviation of Top branch and it means the Top branch is taken, any B is an abbreviation of Bottom branch and it means the Bottom branch is taken.

The execution traces of the given instructions for both systems are shown in Figure 5.13. As we discussed before, the elastic system has to maintain the completion order of events. So from cycle 3 to 4, it must hold I3 at "In" until I2 is finished in cycle 5. The last column of elastic execution trace clearly shows that all instructions are completed.

(a) Toy case example.

<table>
<thead>
<tr>
<th>Instruction ID</th>
<th>I1</th>
<th>I2</th>
<th>I3</th>
<th>I4</th>
<th>I5</th>
<th>I6</th>
<th>I7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Taken</td>
<td>B</td>
<td>T</td>
<td>B</td>
<td>B</td>
<td>T</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

(b) The instructions to execute.

Figure 5.12: Toy case to compare Elastic and Fluid Pipelines.
in order strictly. In contrast, the execution order of Fluid Pipelines can be altered. The first time this reordering happens is in cycle 3, where I3 moves to the bottom branch while I2 is being executed in the top branch. This is behavior that must be known and specified by the user, and not arbitrarily changed by the tool. If a user did not indicate this in the code, no reordering would be performed. In that case, the order would be maintained and the result would be similar to a traditional Elastic System. The last column of fluid execution trace shows that instructions are completed out of order. For this example, using Fluid Pipeline can save 3 cycles compared to Elastic System, thus throughput improvement is achieved.

**Figure 5.13:** Execution traces of Elastic System vs Fluid Pipelines.
5.5 Deadlock Avoidance In Fluid Pipelines

Deadlocks are a concern in a design that contains loops where circular dependencies may exist [6]. It is shown that in Elastic Systems extraneous dependencies can cause deadlocks [22]. That means one output of a module waits for an input that it does not depend upon to fire. Another issue is the creation of a token in the output before the consumption of one in the input. This is specially a problem in Fluid Pipelines due to the additional freedom and flexibility compared to traditional Elastic Systems [16]. Deadlocks can be avoided by adhering to the following two properties. First, the No-Extraneous Dependencies (NED) property. If an output O of a module M does not depend on an input I of M, then O should be produced regardless of the existence of a valid I. In addition, the dependency list of an output of a module should be a subset of the inputs of the module. Second, the Self-Cleaning (SC) property: A circuit has the SC property if whenever it produced N tokens on its outputs, it must also have consumed N tokens from its inputs. These two properties are not meant to restrict which designs are possible, but rather to guide how to implement each design.

One example is given in Figure 5.14 to demonstrate how to avoid deadlocks. This module has 4 registers A, B, C and D, where the value of C depends on the values of A and B, and the value of D depends on the value of B only. When we want to convert this module to a Fluid Pipelines circuit, there are different implementations of the resulting circuit.

One straightforward implementation is to wait for all the input EBs (A and B) to become valid and all the output EBs (C and D) are ready to accept data, then simultaneously produces both C and D. However, this implementation forces D to be produced when A is valid, whereas D does not depend on A in the original design. Thus, it violates the NED property. This may lead to deadlocks in some situations. For example, if we connect D to A from the outside when using this module, the module will deadlock.
Figure 5.14: Deadlock avoidance example in Fluid Pipelines.

because A becomes valid only when D is produced.

To solve this issue, we can use a Fork operator at B. So the implementation becomes the following: C is produced when A and B are valid and it can accept data, whereas D is produced when B is valid and D can accept data. In this implementation, D does not depend on A any more. Thus the deadlock situation is avoided.

To explain why violating the SC property can cause deadlocks, let us see another example. Imagine a circuit that consumes N inputs per token and assume the buffer size is M. If the output of this circuit is fed back to its input, then after M/N cycles, the buffer will be full and a deadlock situation occurs. So the SC property is necessary to prevent deadlocks caused by buffer overflow.

5.6 The Algorithm

The main steps of the Fluid Pipelines conversion are Find Joins, Find Forks, and add operators.

5.6.1 Find Joins

In this step, we forward traverse the graph. For each node (say node A) visited, we check if there is any in edge from a Flop node. If so, we map A to the join-index-has-flop-map to indicate this node has a Flop node as its input, also we append the Flop
node id to the array which is mapped to the current node A in the join-index-flop-map. If A has a node (say node B) other than Flop as its input, we copy and append the array which is mapped to B in the join-index-flop-map to the array which is mapped to A in the join-index-flop-map. At the end, if we found that node A is a Flop node and join-index-has-flop-map[A] is empty or false, then we mark node A as In Flop in the vector in-flop-vec to indicate that its input paths do not contain any Flops. That means the input of A is connected to the module input port.

In Figure [5.15] for example, after we visited XOR, the content of join-index-flop-map[XOR] is B and C. When we are visiting F, the content of join-index-flop-map[XOR] is copied and appended to F. So after F is visited, join-index-flop-map[F] has B and C. Thus in the later steps we know that F joins B and C, so a Join operator should be inserted.

The reason we want to traverse the graph forward is to make sure that the input paths of a node have been visited before the node. For example, in Figure [5.15] when F is being visited in2, B, in3, C and XOR must have been visited already. Otherwise F will not gather the complete information of its input paths.

The pseudo code of the algorithm is given below:

For the index idx of each node in a forward traversal of the graph

for all Input nodes inp_idx of idx do

if inp_idx is a Flop node then

if inp_idx is not in join_index_flop_map[idx] then

join_index_has_flop_map[idx] ← True

Push back inp_idx to join_index_flop_map[idx]

end if

else

if join_index_has_flop_map[inp_idx] then

end if

end if

end for all

36
Figure 5.15: Fluid Pipelines Conversion Example.

Push back `join_index_flop_map[inp_idx]` to `join_index_flop_map[idx]`

end if

end if
end for

if idx is a Flop node then

Push Back idx to `in_flop_vec`

end if

5.6.2 Find Forks

In this step, we traverse the graph backward. For each node (say node A) visited, we check if there is any out edge from A to a Flop node. If so, we map A to the fork-index-has-flop-map to indicate A has a Flop node as its output, also we append the Flop node id to the array which is mapped to A in the fork-index-flop-map (i.e.
fork-index-flop-map[A]). If A has a node (say node B) other than Flop as its output, we copy and append the array which is mapped to B in the fork-index-flop-map (i.e. fork-index-flop-map[B]) to the array which is mapped to A in the fork-index-flop-map (i.e. fork-index-flop-map[A]). At the end, if we found that node A is an Flop node and fork-index-has-flop-map[A] is empty or false, then we mark node A as Out Flop in the out-flop-vec to indicate that its output paths do not contain any Flops.

In Figure 5.15 for example, after we visited INV2, the content of fork-index-flop-map[INV2] is G. Similarly, after we visited INV3, the content of fork-index-flop-map[INV3] is H. When we are visiting D, the content of fork-index-flop-map[INV2] and fork-index-flop-map[INV3] are copied and append to D. So after D is visited, fork-index-flop-map[F] has G and H. Thus in the later steps we know that D forks G and H, so a Fork operator should be inserted.

The reason we want to traverse the graph backward is to guarantee that the output paths of a node have been visited before the node. For example, in Figure 5.15 when D is being visited out3, G, INV2, out4, H and INV3 must have been visited already. Otherwise D will not gather the complete information of its output paths.

The pseudo code of the algorithm is given below:

For the index idx of each node in a backward traversal of the graph

for all Output nodes out_idx of idx do

if out_idx is a Flop node then

if out_idx is not in fork_index_flop_map[idx] then

fork_index_has_flop_map[idx] ← True

Push back out_idx to fork_index_flop_map[idx]

end if

else

if fork_index_has_flop_map[out_idx] then


38
Push back fork_index_flop_map[out_idx] to fork_index_flop_map[idx]

end if

end if

end for

if idx is a Flop node then

Push Back idx to out_flop_vec

end if

5.6.3 Add Operators

The next step is to add global IOs including ValidIn, StopOut, ValidOut, and StopIn to this module. While we are forward traversing the graph, we convert all Flops to Fluid Flops.

Next, all In Flops found in in-flop-vec are connected to global ValidIn and StopOut, while all Out Flops found in out-flop-vec are connected to global ValidOut and StopIn.

Then for each Flop node (say Node A), we check fork-index-has-flop-map[A], if True is returned then we know A forks to one or more Fluid Flops. Then we look up fork-index-flop-map[A] and there are different cases.

Case 1: A only forks to Node B, and B only joins A (this is checked in join-index-flop-map[B]). For case 1, we just need to connect the Valid and Stop signals between A and B.

Case 2: A forks to Node B and Node C, whereas B and C both only joins A. For case 2, we need to insert a Fork operator between A and B, C. One example can be found in Figure 5.15 a Fork operator should be added so that D forks G and H.

Case 3: A only forks to Node C, and C joins A and Node B. In addition, B also only forks to C. This is a typical Join case, we need to insert a Join operator between A, B and C. One example can be found in Figure 5.15 a Join operator should be added so
that F joins B and C.

Case 4: A forks to Node C and Node D, while another Fluid FLop node B also forks to C and D. In addition, C and D both join A and B. In this case, A, B should be merged into a new Fluid Flop node (say Node AB), also C, D should be merged into another new Fluid Flop node (say Node CD). Then the Valid and Stop signals between AB and CD are connected.

Case 5: The deadlock prune case as shown in Figure 5.14. In this case, as aforementioned, we can use a Fork operator at B to create B1 and B2. Then C joins A and B1, while B2 is connected to D. The Valid and Stop signals between B2 and D are connected. We can name this process as Deadlock Avoidance Operator.

The pseudo code of the algorithm is given below:

Add ValidIn as Input port, StopOut as Output port, ValidOut as Output port, and StopIn as Input port of this module

for all Flop node flop_idx in the graph do
    Change the node type of flop_idx to Fluid Flop (Elastic Buffer)
end for

for all in_flop in in_flop_vec do
    Connect in_flop to ValidIn and StopOut
end for

for all out_flop in out_flop_vec do
    Connect out_flop to ValidOut and StopIn
end for

for all Fluid Flop Node idx do
    Check in fork_index_flop_map and join_index_flop_map to decide case_number.

switch case_number do
    case 1
Connect the Valid and Stop signals between the two nodes.

case 2
Add Fork Operator.

case 3
Add Join Operator.

case 4
Combine nodes to form a wider Fluid Flop.
Then move the previous connections to this Fluid Flop.

case 5
Add Deadlock Avoidance Operator.

end for

5.7 Contributions

- Double Buffering, Elastic Systems and Fluid Pipelines are introduced.

- Examples are shown to help understanding.

- The algorithm of Fluid Pipelines conversion is provided.
Chapter 6

Conclusion

This thesis introduces Live Graph which is a graph optimized for live synthesis. Then three optimization techniques are described, including Dead Code Elimination (DCE), Global Value Numbering & Partial Redundancy Elimination (GVN-PRE), and Fluid Pipelines conversion.

For DCE and GVN-PRE, traditional concepts in software compilation context are first introduced, then the ideas and implementations to leverage of them in hardware context are given. The ultimate goal of these two techniques are to reduce gate counts and design area.

For the Fluid Pipelines conversion, a detailed introduction is given to cover all the basic concepts, evolution and usage of Fluid Pipelines. Then the process of Fluid Pipelines conversion is given.

Various examples are given in each part of the techniques to provide better understanding. The algorithms are added for each technique at the end of each section.

This thesis only shows parts of the strengths of the Live Graph; more optimization methods can be added to extend the Live Graph.
Bibliography


