The University of New South Wales

School of Computer Science and Engineering

VPR Assessment of a Novel Partitioning Algorithm

David Munro

z3218420

A thesis submitted for partial requirement of the degree:

Bachelor of Engineering (Computer)

Submitted: June, 2013

Supervisor: Oliver Diessel

Assessor: Sri Parameswaran

Acknowledgements

I would like to thank my supervisor Oliver Diessel for his assistance and advice throughout the entire process. I also wish to thank Patricia Munro, Salima Yeung, Michelle Kopac, David Ma and Anushka Wikramanayake for their proofreading.

Abstract

Field-Programmable Gate Array (FPGA) systems would be well suited to space-based applications except for their vulnerability to space-based radiation. Various techniques for dealing with their susceptibility have been discussed in the literature. A key part of a theoretical technique to protect against radiation-induced Single Event Upsets (SEUs) was developed and the implementation assessed using open-source tools. Results indicate that the developed algorithm exhibited respectable performance compared to existing alternatives while providing theoretically greater fault tolerance, although further work is required to implement the algorithm for current commercial devices.

Contents

List of Tables

List of Figures

Glossary

ABC A System for Sequential Synthesis and Verification.

ASIC Application-Specific Integrated Circuit.

BLE Basic Logic Element.

BLIF Berkeley Logic Interchange Format.

CAD Computer-Aided Design.

CLB Configurable Logic Block.

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

DFG Directed Flow Graph or Data Flow Graph.

DICE Dual Interlock Storage Cell.

FPGA Field-Programmable Gate Array.

ICAP Internal Configuration Access Port.

IO Input/Output.

LUT Lookup Table.

MBU Multi-Bit Upset.

MCNC Microelectronics Centre of North Carolina.

mux Multiplexer.

NRE Non-Recurring Engineering.

primitive Most basic circuit element. Either a latch or a Lookup Table (LUT).

SAT Boolean Satisfiability Problem.

scrubbing Refreshing an FPGA's configuration memory to purge accumulated errors.

SEU Single Event Upset.

SRAM Static RAM.

TMR Triple Modular Redundancy.

VHDL VHSIC Hardware Description Language.

VPR Versatile Place and Route.

VTR Verilog To Routing Project.

Chapter 1

Introduction

1.1 Overview

Space plays an increasingly important role in the functioning of modern societies, being vital for fields including navigation, meteorology, and communications [20]. Field-Programmable Gate Array (FPGA) systems have many beneficial features, such as their flexibility and low Non-Recurring Engineering (NRE) costs which make them highly desirable for space-based applications. Unfortunately they have far greater susceptibility to space radiation. Hardened FPGAs offer only a fraction of the gate counts (and hence capability of implementing large or complex circuits) of non-hardened offerings prompting a search for a solution to the radiation susceptibility of FPGAs using mainstream hardware, one of the most popular of which is Triple Modular Redundancy (TMR) [17]. In TMR, vulnerable components are triplicated allowing for errors to be detected and mitigated. This thesis is based on the work of [9] which introduces an approach to TMR, and aims to develop a key part of the approach and assess the implementation with the aid of an open-source Computer-Aided Design (CAD) toolchain for FPGAs.

The remainder of this chapter provides an overview of these technologies, discusses alternatives to our approach, and details why we have chosen the technique we have. Chapter 2 discusses our high level design goals and provides some derivation of numbers used in our implementation, Chapter 3 describes our implementation and design choices made in the implementation, Chapter 4 presents our results and Chapter 5 briefly discusses some limitations and possible directions of future work.

FPGAs

FPGAs are popular devices capable of implementing a wide variety of circuits. Unlike Application-Specific Integrated Circuits (ASICs) which must be specially designed and manufactured for an application—a lengthy and expensive process—FPGAs are a generic off-the-shelf device which can be mass produced by manufacturers and then adapted for an individual user's needs. Their flexibility, low cost, and faster development time make them the most economic for a range of applications.

There are three main components to an FPGA: Input/Output (IO) blocks, usually around the edge, allowing for input and output from the FPGA; Configurable Logic Blocks (CLBs) containing all

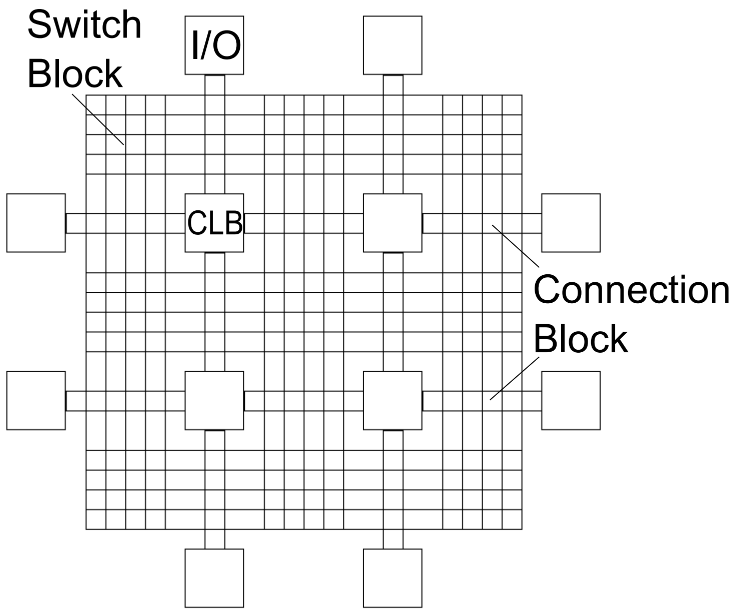

Figure 1.1: Island Style FPGA

the logic elements or *primitives*; and the routing between all the components. Most FPGAs also contain other structures embedded in the CLB array to provide commonly used resources such as multipliers. While they can be implemented using registers and Lookup Tables (LUTs), embedding them as discrete components allows for denser designs. The routing between components consists of channels running horizontally and vertically with a number of wires and programmable switches connecting the wires to each other and to CLBs allowing for configurable paths between arbitrary components. A typical switch or connection block has a configuration cell storing the state, and a connection can be made or unmade by writing a new value to the cell for that switch. The most common style of routing is known as island style (as the CLBs are located as islands in a sea of routing) with the routing area making up some 80%-90% of the FPGA's area [11]. Each CLB is a cluster of smaller blocks, called Basic Logic Elements (BLEs), with each BLE containing the logic primitives, typically a programmable LUT to implement combinational logic, a register for register operations and implementing sequential logic, and a Multiplexer (mux) to switch between the two.

The values for the LUT, whether the mux is selecting the register or LUT output, and other component states are all stored in configuration memory like the routing switches and are typically implemented in Static RAM (SRAM).

Programming an FPGA involves loading in a bitstream which describes all the component values (i.e. contents of the configuration memory for each cell) for a circuit, accomplished through writing the bitstream to a special configuration port on the FPGA. A number of FPGAs also allow for run time programming, or reconfiguration, of parts of a circuit through loading the bitstream for only the section of interest while the rest of the FPGA keeps running.

There are three main technologies used to implement the configuration memory in FPGAs:

- SRAM, which gives the highest density devices and includes the Virtex-5 family this thesis focuses on. These are volatile and must be reprogrammed every power up from an external configuration memory.

- (Anti)fuse, which are only one-time programmable.
- Flash, which is non-volatile (thus not does not require an external configuration memory) and reprogrammable. These have a lower density than SRAM based FPGAs [11].

Latch vs Flip-Flop

Typically, the register can implement either a latch or a flip-flop, as such future references to either latches or flip-flops both refer to a sequential logic element implemented by a register. Generally the term latch is used for consistency with the language used by the Berkeley Logic Interchange Format (BLIF) specification, and by Versatile Place and Route (VPR) and ABC, although when discussing or referring to sources which use the term flip-flop the term flip-flop is also used.

Partial Reconfiguration

Partial reconfiguration involves loading configuration information for part of a circuit during operation. Much like the complete configuration described above, it involves writing a configuration bitstream to one of the available configuration ports, in this case also including the location of the region to reconfigure. The configuration memory of recent Virtex devices is subdivided into frames, and one can only reconfigure entire frames. A configuration frame is 41 (32-bit) words long on a Virtex-5 device. The larger the area being reconfigured the more frames required, and consequently the larger the bitstream and hence the longer the time to reconfigure. The main configuration ports used are the external SelectMAP interface or the equivalent Internal Configuration Access Port (ICAP), with a bandwidth of 400MB/s in all Virtex devices [9, 25]

Space Based Applications

Space is quite different from a terrestrial environment, and FPGAs have a number of advantages due to their lower NRE costs and flexibility. As FPGAs can be reconfigured during a mission, faulty or outdated designs can be replaced remotely; however, there is a significant downside: as systems go further into space and are no longer protected by the earth's atmosphere they become increasingly likely to suffer from radiation-induced errors where ionising radiation impinging on a component causes charge build up, potentially triggering incorrect operation [23]. As outlined in Table 1.1, for higher orbits the mean time to upset is on the order of only a second, and this rate increases as technology advances and chip density further increases. Of the potential effects, which range from unnoticeable to device destruction, this thesis is concerned with mitigating Single Event Upsets (SEUs), where an incorrect signal is triggered but the underlying circuitry is not damaged. We also concern ourselves primarily with errors affecting only single bits or components rather than Multi-Bit Upsets (MBUs) in which multiple components are affected at the same time.

Orbit	SEUs per device/day	Mean time to upset (s)
LEO (560 km)	4.09	2.11×10^4
Polar (833 km)	1.49×10^4	5.81
GPS (20,200 km)	5.46×10^4	1.58
Geosynchronous (36,000 km)	6.20×10^4	1.39

Table 1.1: SEU Rate Predictions for a Virtex-4 XC4VLX200 device at various orbits [9]

In an ASIC, while SEUs may be picked up and latched or otherwise continue affecting the circuit in future, the component itself continues operating normally.

FPGAs on the other hand are vulnerable to configuration errors as well. When the charged particle impacts configuration memory it can flip the state of that cell changing the implemented circuit. Unlike transient errors, these functional errors persist until corrected.

Additionally for SRAM devices, the off-chip configuration memory itself can be affected, so the next time the chip is reprogrammed (e.g. after power cycling), an incorrect circuit configuration will be loaded.

(Anti)fuse devices, being non reprogrammable, are immune to configuration errors, though both SRAM and flash-based FPGAs are vulnerable and all three are susceptible to transient SEUs [7].

How We Deal With FPGA Downsides

Clearly, in order for FPGAs to be viable in space-based systems the effects of SEUs must be mitigated. A number of technologies and techniques are available, each with their own advantages and disadvantages. A number of options exist which detect errors but are unable to determine the correct result, requiring a reload of the configuration memory while the circuit is non operational until the reconfiguration completes. For many applications this downtime is impractical, thus we will be looking at options which allow the circuit to continue operating correctly. There are three main categories of SEU hardening techniques for FPGAs [5]:

- Charge Dissipation, which aims to keep the effect of the radiation below the level where it would have an effect. This includes techniques such as increasing the drive current. These methods typically require custom hardware (increasing costs) and usually increase power usage.

- Temporal Filtering, which aims to filter out transient SEUs, includes methods such as delay-and-vote [5]. These techniques often slow down operation and are ineffective against configuration errors.

- Spatial Redundancy, which uses multiple redundant circuits to detect errors and be able to continue operating. Spatial redundancy techniques include Dual Interlock Storage Cell (DICE) [8] and TMR and can be implemented either in hardware, or at the design level not requiring any custom hardware. These methods typically increase area and power usage.

	Power (μW)	Speed (ns)	Hardness (errors per bit-day)	Node Failures Required for Device Failure	Area (μm^2)
Standard	Rise – 0.7 Fall – 0.2	Rise – 0.21 Fall – 0.27	10^{-8}	1	360
Increased Drive Current	Rise – 1.0 Fall – 0.2	Rise – 0.16 Fall – 0.15	2×10^{-9}	1	460
TMR	Rise – 1.72 Fall – 1.27	Rise – 0.2 Fall – 0.27	10^{-11}	2	1200
DICE	Rise - 1.4 Fall - 1.1	Rise - 0.96 Fall - 0.97	1.6×10^{-10}	2	520

Table 1.2: Comparison of hardening techniques [5]

While hardened FPGAs are available, they typically lag well behind mainstream commercial offerings [17], thus solutions which can be implemented on mainstream commercial FPGA hardware are desirable. Additionally, there is very little point hardening an FPGA and not its configuration buffers and memory which take up far more surface area [11] and are thus even more vulnerable. For these reasons TMR, requiring no custom hardware and providing SEU protection against both transient and functional errors, is one of the more popular SEU hardening techniques even though it comes at the cost of more than tripling area and greatly increasing power usage. Table 1.2 details power usage, operating speed, hardness, and required area for flip flops which have been hardened using the techniques listing within the table. As can be seen, TMR provides the greatest hardness (measured as the greatest average time between errors) at the cost of the highest overhead in power and area usage. Additionally, for SRAM based FPGAs the off-chip configuration memory must also be protected as SRAM is volatile and loads the state from this memory at start up. This can be accomplished by incorporating error detection and correction techniques in the RAM, something already in place on a number of mainstream FPGAs such as the Virtex-4 and -5 [10].

One additional type of hardening is physical shielding i.e. surrounding the FPGA with a material to block incoming radiation. Unlike the above approaches this requires no modification to the FPGA hardware or implemented circuit. Unfortunately, it increases cost, weight and size, and may not always be practical.

1.2 TMR

Triple Modular Redundancy is a commonly used method for creating fault tolerant systems in which a given circuit is implemented three times with independent components, with the outputs feeding into a voter circuit to determine the majority value. As an SEU affects only a single component or bit of data it will affect the output value of at most one version, so the majority vote is still correct. For transient errors that are not in a feedback loop correcting the output is enough to fix the error;

however, SEUs in feedback paths or in the configuration memory will persist, and this necessitates some method for eliminating them. One possible approach is resetting the system but while this occurs the system is unavailable, so a reset may not be a feasible solution. Instead, partial reconfiguration could be used to reconfigure only the faulty circuit while the redundant circuits continue operating and providing output. After reconfiguration the circuit must then be resynchronised to the same state as the other two. We use the approach presented by [9] which involves running the circuit until the state converges, which is guaranteed (for acyclic circuits) to occur within a time frame given by the number of register stages (also referred to as critical path length) and the clock period. In order for this approach to always resynchronise correctly the circuit must have no feedback loops which could carry incorrect data. To solve this we simply ensure that all feedback loops are *cut*, that is, the value is voted on before being passed back into the circuit. This has the additional benefit of correcting transient errors which would otherwise be caught in a feedback cycle by ensuring the cycle data is correct.

This approach requires three times as many circuit elements (as the circuits are triplicated) plus whatever is required for voters. By minimising the number of voters, we can thus reduce the overhead of our approach.

Error Recovery Time for TMR

Once an error occurs it takes up to T_{path} to reach the voter and be detected, where T_{path} is given by the clock period and number of register stages. This is called the *error detection time*. Detection of an error can then be used to trigger reconfiguration.

Sending a request to the reconfiguration controller goes through a token ring network connecting together the other voters and the reconfiguration controller. In the worst case it takes one full cycle of the network to receive the token, one full cycle to reach the reconfiguration controller, and three cycles to transmit the request, giving $5 \times CyclesPerHop$. Benchmarks of a sample voter indicate 50 clock cycles per hop is a good estimate.

Reconfiguration time is dependent upon the circuit size. For our target device based on Virtex-5 we round the circuit's area usage up to an allocatable area of 20 CLBs (representing one column of CLBs in a reconfiguration row). Each CLB consists of 8 BLEs (each BLE having one LUT and one latch) giving us a target reconfiguration area that consists of 160 BLEs and requiring 36 frames of 41 32-bit words each. The bitstream size for this area is 1476 words which takes $14.8\mu s$ to reconfigure at 100MHz [2]. Once the error has been detected and the circuit reconfigured it must then be resynchronised with the other partitions, which takes up to T_{path} using the previously described technique.

The error recovery time consists of the time to detect the error, send a request to the configuration controller, and then reconfigure and resynchronise the circuit, thus is a function of the circuit area, clock period, and number of register stages. Therefore it is required that the number of register stages and area are small enough, that our error recovery time is within a user specified limit.

$$
\begin{aligned}
\text{Error Recovery Time} &= \text{Error Detection Time} + \text{Communication Time} \\
&\quad + \text{Reconfiguration Time} + \text{Resynchronisation Time} \\
\text{Error Detection Time} &\leq T_{path} = \text{Clock Period} \times \text{Register Stages} \\
\text{Communication Time} &\leq 5 \times \text{Cycles per hop} \times \text{Number of Hops} \times \text{Clock Period} \\
&= 50 \times 5 \times (\text{Number of Partitions} + 1) \times \text{Clock Period} \\
\text{Reconfiguration Time} &= \frac{\text{Bitstream Size}}{\text{Reconfiguration Speed}} \\
&= \left\lceil \frac{\text{Number of BLEs}}{160} \right\rceil \times 1.48 \times 10^{-5} \\
\text{Resynchronisation Time} &\leq T_{path} = \text{Clock Period} \times \text{Register Stages}
\end{aligned}
\tag{1.1}
$$

While most of the values used are directly calculable, the number of partitions and the circuit clock period are not known until after partitioning and after routing respectively and must therefore be estimated. The estimations used in our implementation are described in Section 2.2.

Additionally, as each voter circuit adds some constant overhead in terms of area, power usage and clock period slowdown it is desirable to have each partition as large as possible. This thesis is concerned with implementing and assessing this TMR design; a discussion of other TMR methods and our reasons for not using them is included below.

This method only works when at most one SEU occurs within the error detection and recovery time; should SEUs occur in two of the three partitions then it is impossible for the voter to determine the correct value necessitating a complete reload of the configuration memory (*scrubbing*). Therefore, we require the error detection and recovery time to be sufficiently small that the likelihood of multiple events occurring within that time period is minimised.

Additionally, as mentioned earlier, it is also desirable to minimise the number of voters to reduce the overhead of this approach. To that end, having larger (and hence fewer) partitions is preferable to smaller partitions provided we still stay within our target recovery time.

TMR Implementations

This thesis builds on the work of [9] which describes a partitioning algorithm that traverses a circuit represented as a Directed Flow Graph or Data Flow Graph (DFG) in a breadth-first manner, creating partitions that stay within our constraints. Our goal is to create an algorithm which stays within a user-specified error recovery time, doesn't require existing code to be rewritten, allows for both custom voting and reconfiguration logic to be added, can use industry standard FPGAs rather than custom hardware, and effectively protects the entire system from SEUs with as close to no downtime as achievable. Additionally, it is desirable to limit the overhead of implementing TMR through minimising the number of voters required. There are a number of existing TMR solutions, but none quite meet our requirements. Our first requirement is that standard FPGA hardware can be used, with

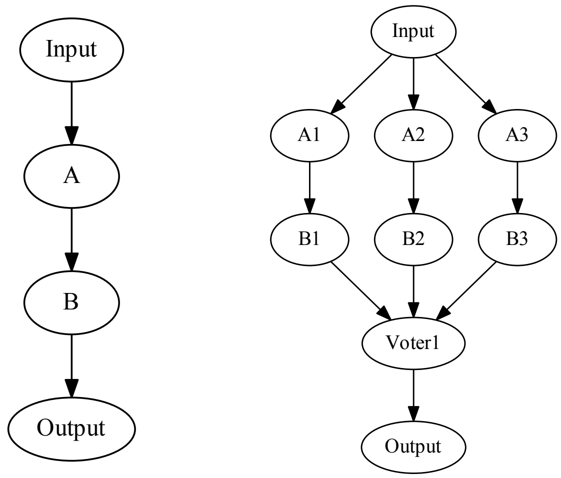

Figure 1.2: DFG before and after partitioning

our implementation specifically targeting Virtex-5 chips. Options with custom hardware such as [17] (with three FPGAs and an ASIC voter in a single package), are often prohibitively expensive, and prevent us from using our existing boards. Many FPGAs marketed specifically at space-based applications are, in addition to featuring specialised hardware, only latchup immune or only include inbuilt TMR on registers, leaving them still vulnerable to SEUs [13]. Non-hardware solutions are typically implemented pre-synthesis, such as [12] (which introduces a VHDL library featuring triplicated components), and require existing code to be rewritten, or during synthesis such as [1] and [3], neither of which supports specifying an error recovery limit, nor for adding reconfiguration logic. Other options look at using partial TMR (such as [21]) which, while it does reduce the overhead of TMR, means the entire circuit is no longer protected, or have excessive downtimes to recover from errors such as [4], which uses idle cycles in a data path to calculate redundant results. One approach similar to ours is presented by [14] who also partition a post-synthesis netlist (represented by a DFG), but their focus is on evaluating techniques for cutting feedback loops, while we focus on partitioning circuits into smaller sub circuits. Cutting feedback loops is however a part of this thesis, and their work could be incorporated in, although for our current implementation a simple depth-first traversal described later in Chapter 3 was chosen.

Our Algorithm

Given a netlist description of a circuit, it is possible to represent the circuit as a DFG [11]. Our goal is to split a DFG into a number of smaller subgraphs, triplicate the components of each subgraph, and insert voting and recovery logic, with each subgraph having independent components and an error recovery time within our threshold. We can then proceed to implement our graph, made up of our new subgraphs, as normal. To do so we traverse the DFG in a depth-first manner, keeping track of the number of register stages and area, extending our partition area as we do so, until our recovery time constraint would be violated. As we extend our partition area we must detect any cycles within our current partition and cut them, joining them back up after the output has been voted on. We thereby ensure that each partition is acyclic and guarantee that the circuit will resynchronise and not get incorrect data trapped in a feedback loop. At that point we cleave off our partition and write it to a file, open a new empty partition, and repeat for all circuit elements. Once this is done, we have a set of subcircuits. We now triplicate each partition and insert our additional voting logic, then join each subcircuit back together.

1.3 CAD Flow

FPGAs are typically programmed in a hardware description language such as VHDL or Verilog, and then a number of programs (collectively making up the CAD flow or development toolchain) turn the source into a bitstream to program a target FPGA. The design flow process can be split into a number of subprocesses as illustrated in Figure 1.3 [6, 11, 16].

1. The synthesiser turns a hardware description language such as VHDL or Verilog into a netlist of basic gates and flip-flops.

2. The optimiser removes redundant logic, and attempts to simplify logic.

3. The mapper maps logic elements to primitives, the basic logic elements contained on the FPGA.

4. The packer combines logic elements into CLBs.

5. The placer locates each CLB within the FPGA architecture, deciding which physical block implements which logic block.

6. The router makes the required connections between each element by deciding which switches are on or off. This includes the connections within each CLB (local routing) and between CLBs (global routing).

For our partitioner we will insert an additional step into the design flow between mapping and packing, which operates directly on a netlist. The additional steps are detailed in Chapter 3.

How VPR Works

For this thesis we will be assessing the results of our algorithm implementation after processing by VPR, an open-source packer, placer and router. VPR was chosen as the algorithms used are public and well-documented, it is open source allowing modifications to be made if necessary, and it is well-documented and popular in research, making it much easier for us to determine what's happening and why, rather than relying on proprietary black-box processes from commercial vendors. Additionally, BLIF (the format used by VPR) benchmarks are readily available. A brief understanding of the algorithms used in VPR and the effects of different settings is useful, though not critical, for understanding the results. [16] has a more detailed list of all the options VPR takes. Unless otherwise specified, all values are at their defaults.

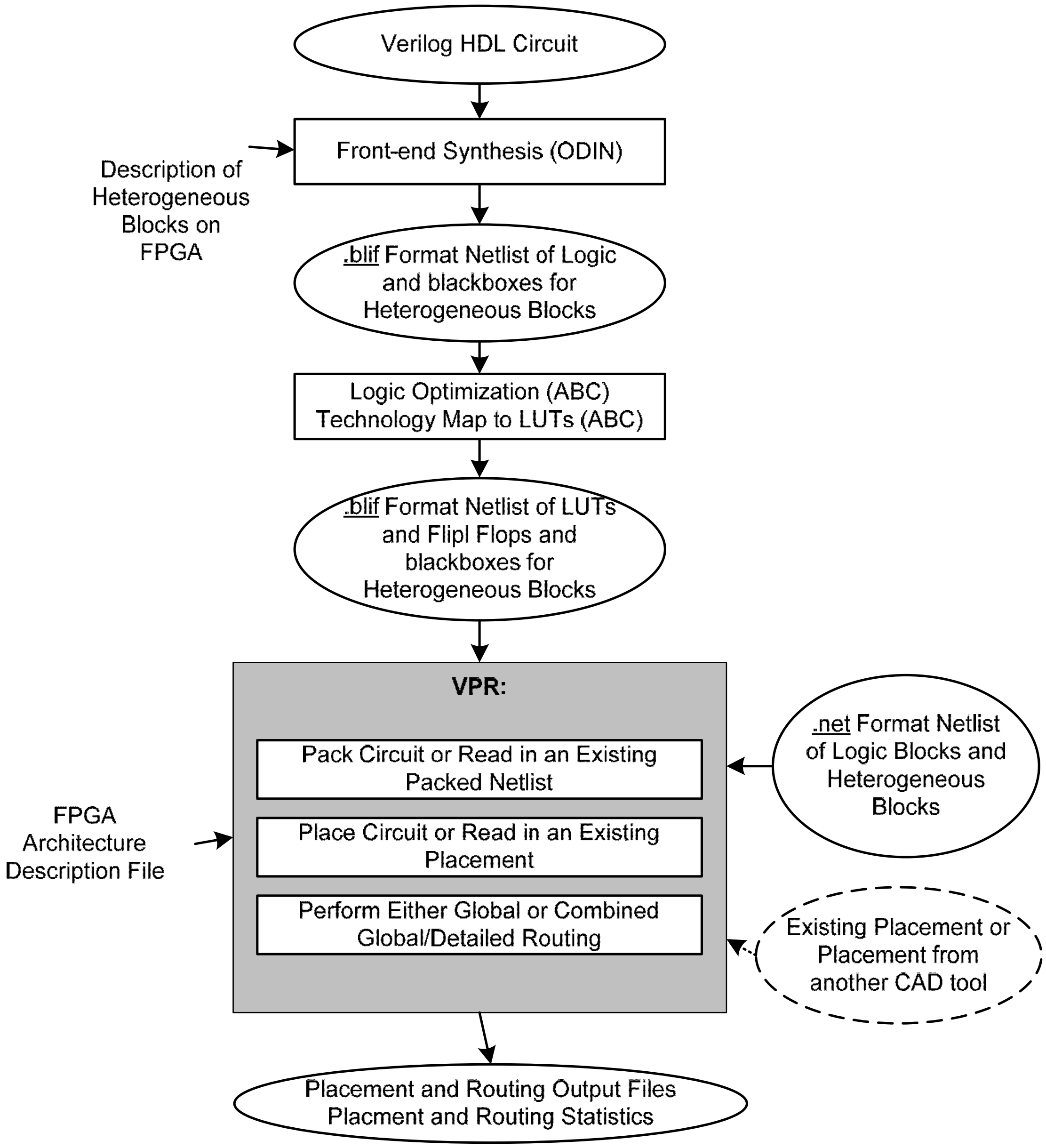

Figure 1.3: Cad Design Flow. [16]

Packer

VPR uses the AAPack algorithm described by [15]. This is a greedy algorithm which operates on blocks sequentially, starting with an FPGA area of 1 block by 1 block. For each block it greedily adds *primitives* (latches or LUTs) based on a configurable cost function until no more primitives can be added. It then repeats for the next block, and the next after that, until every primitive has been packed. As it runs out of blocks in the current FPGA area it expands the FPGA area used until it reaches the physical limit specified in the architecture file (or grows indefinitely if no limit is specified). This means that even if the device is of area 40 by 40, if the packer can fit everything in a 30 by 30 area it will do so, and VPR will treat the FPGA as being only 30 by 30. The cost function can be configured through options passed to VPR, to [16]:

- prioritise optimisation of timing or area (default is prefer timing)
- prioritise absorbing nets with fewer connections over those with more (default is yes)
- when prioritising absorbing nets with fewer connections, focus more on signal sharing or absorbing smaller nets (default is greatly prefer absorbing smaller nets)
- determine the next complex block to pack based on timing or number of inputs (default is timing).

The main thing to note, as relates to our results, is that as much as possible AAPack will never leave blocks partially packed while there is still a primitive which will fit. Even when optimising timing exclusively, it will still attempt to maximally pack each cluster even if it negatively impacts circuit performance.

Placer

VPR's placer uses a simulated annealing algorithm where the options allow us to specify annealing schedule parameters and cost function. The default options were chosen via experimentation and are likely superior to custom options we may choose to use, and affect the average quality of the result rather than materially affecting the behaviour [6, 16]. For these reasons we will be leaving them at their default. Section 4.3 discusses the variation in results due to the stochastic nature of the placement algorithm.

Router

VPR's router supports three different algorithms: breadth_first, which focuses solely on routing a design; timing_driven, the default, which tends to use slightly more tracks (5%) than breadth_first while providing much faster routes ($2\times$–$10\times$) with less CPU time; and directed_search, which like breadth_first is routability driven however uses A* to improve run time. We will be using the default timing_driven algorithm. There are a number of other options setting algorithm parameters, all of which we will leave at their defaults. Additionally, we can set the width of the architecture's routing channels through the route_chan_width parameter. If omitted VPR will perform a binary search on channel capacity to determine the minimum channel width.

Chapter 2

Project Outline

2.1 Project Objectives

The objective of this thesis is to create an implementation of the algorithm outlined by [9] and assess the overheads of this method. As such, we need to create a *correct* implementation, that is, one which :

1. Correctly implements TMR.
2. Preserves the original inputs and outputs. Signals should retain the same names, and for a set of inputs, the circuit should have the same output as the original circuit.
3. Is accurate in partitioning, such that subpartitions are all within the target recovery time.

and then evaluate the overhead of this algorithm in terms of algorithm running time, and how it affects the performance of the final circuit.

2.2 Design of Partitioning Algorithm

Chapter 3 describes the partitioning algorithm fully; in brief our design is to:

1. Split a larger input circuit into smaller partitions;
2. Triplicate each partition;
3. Join them back together.

While splitting our design into smaller partitions we need to cut any cycles internal to a partition such that they pass through the voter circuit and then the corrected output is fed back into the partition.

The size of each partition is set such that each partition has an error recovery time less than a user specified target. The error recovery time is calculated as per Section 1.2 with estimates for the number of partitions and final circuit clock period as follows: An initial guess for the final number of partitions is set at 1, and the partitioner is run to completion. The guess is then updated to the actual number of generated partitions and the partitioner is rerun. This repeats until the guess is the same or greater than the actual generated number of partitions, or partitions within the target recovery time were unable to be created. In practice, it generally only takes two or three runs to converge to the actual number of partitions for the twenty largest Microelectronics Centre of North Carolina (MCNC) benchmarks although some tested corner cases took up to ten repetitions on increasing estimates before the partitioner determined that the target recovery time was unable to be met. As all target benchmark circuits completed partitioning relatively quickly and it is impossible for the partitioner to be stuck in an infinite loop of revising its estimate optimising this estimation was not considered necessary.

In addition to an estimate for the number of partitions, to calculate the recovery time the partitioner requires an estimate for the final clock period. This is derived as $1.8\times$ the original circuit's clock period, as reported by VPR upon processing the original circuit. 1.8 was experimentally chosen, as the all the circuits were under an average $1.8\times$ slowdown, and the average case was well under.

2.3 Assessment of Partitioning Algorithm

Assessment is based on the fulfilment of the criteria outlined earlier in this chapter, in Section 2.1 as they relate to a set of benchmark circuits, the twenty largest MCNC circuits from LGSYNTH'93. Generated circuits were verified against and compared with the original circuits to confirm that the algorithm operated correctly, as described further in Section 3.4. The CPU time of the algorithm as it runs against the benchmarks was recorded and compared with the running time of VPR as detailed in Section 4.6. And lastly, the performance of the generated circuits was compared to the original untriplicated circuits in Chapter 4.

Chapter 3

Algorithm

For our partitioner, we operate on a netlist in BLIF format (described in Section 3.6) after optimisation and technology mapping, but before packing. Our goal is to take an input netlist and transform it into a netlist in the same format, with the same set of outputs for each set of inputs, but with redundant components.

Figure 3.1 illustrates a typical CAD toolchain with our custom partitioner added and the substeps expanded (c.f. Figure 1.3 for an example without). The steps below are explained in more detail in Section 3.2.

- Partition - Take an input circuit and split it into multiple smaller circuits, one per file;
- Triplicate - Take an input circuit and transform it into a TMR'd version;
- Join - Take a set of input files, one circuit per file, and join them into one larger circuit by joining corresponding signals;
- Flatten - Use ABC to transform a hierarchical circuit into a format supported by VPR;
- Test - Use the verification capability of ABC to verify that the generated circuit is equivalent to the original.

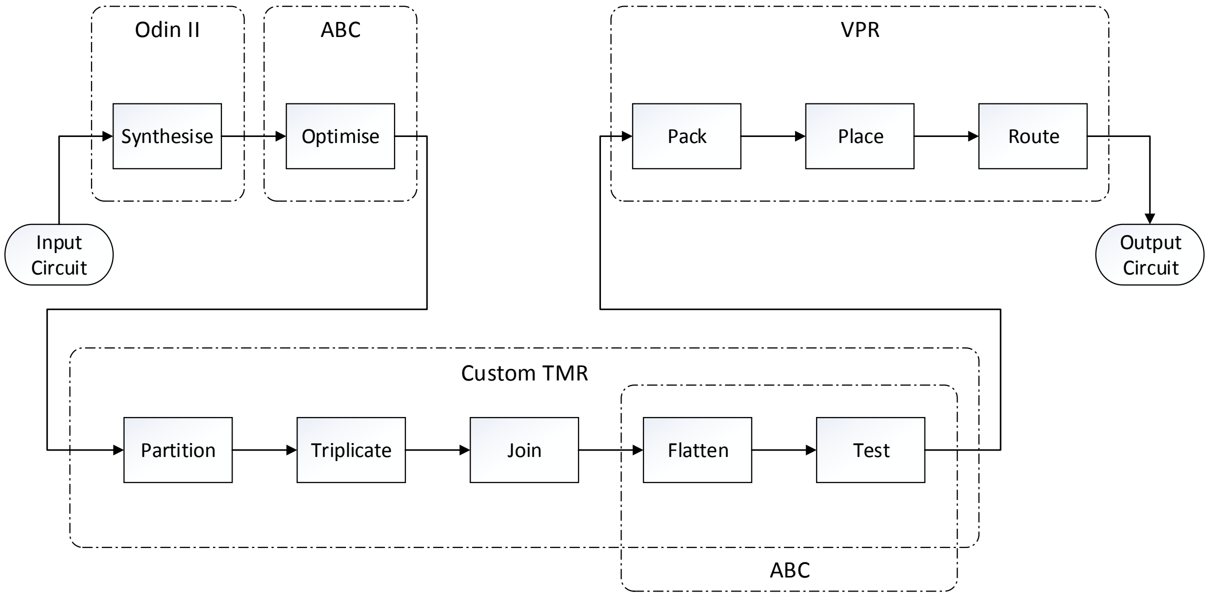

Figure 3.1: Custom Tool Flow

3.1 Data Structures

Basic Types

Table 3.1 lists the basic types, out of which others are built. There is generally, but not always, a direct relationship to a C++ primitive. Table 3.2 contains an overview of the custom complex types, which are further explained below.

Blif

Contains helper functions to read in a BLIF and represent it as a DFG. The circuit itself is represented as a Model within Blif.

Model

Represents the circuit as a DFG, with a list of BlifNodes and the Signals both between nodes, and the primary inputs/outputs of the circuit. Also contains a mapping from signal name to Signal.

Name	Closest C++ Equivalent	Description
Integer	int	Whole number
Boolean	bool	True or False
Float	float	Floating point number
Queue	std::list	FIFO queue
List(type)	std::list⟨type⟩	
String	std::string	String object that provides operations to manipulate itself
File	std::iostream	Abstract type to represent simple I/O operations
Map(KeyType → ValueType, DEFAULT: DefaultValue)	std::unordered_map⟨KeyType, ValueType⟩	A map to translate values of type KeyType to values of type ValueType. If the key isn't present, returns DefaultValue

Table 3.1: Basic Data Types

Name	Description
Blif	Parent object, contains all information about a BLIF file and provides useful operations
Model	Represents a circuit within a BLIF file, and provides methods to manipulate said circuit
BlifNode	A circuit element, or node in the DFG representing the circuit
Signal	A signal within a specific circuit, or Model, representing a set of edges with common source

Table 3.2: Complex Data Types

Field Name	Type	Description
masterOutputs	**List(String)**	List of outputs for the original file
masterInputs	**List(String)**	List of inputs for the original file
main	**Model**	The main circuit in the BLIF file

Table 3.3: Fields in Blif object

Field Name	Type	Description
name	**String**	Name of the circuit
signals	**Map(String → Signal, DEFAULT: NULL)**	Map from signal name to Signal object
outputs	**List(Signal)**	List of output Signal objects
inputs	**List(Signal)**	List of input Signal objects
nodes	**List(BlifNode)**	List of all nodes in a circuit
numLatches	**Integer**	Number of latches within a circuit
numLUTs	**Integer**	Number of LUTs within a circuit

Table 3.4: Fields in Model object

Field Name	Type	Description
output	**String**	Name of output signal
clock	**String**	Name of clock signal
inputs	**List(String)**	List of input signal names
cost	**Integer**	How many clock cycles this node contributes to the critical path. 0 for LUTs and 1 for latches.
type	**String**	Type of node, "latch" or "names" (LUT)
contents	**String**	Parameters describing node which are not used by partitioner but required to recreate BLIF file e.g. initial latch state

Table 3.5: Fields in BlifNode object

Field Name	Type	Description
name	**String**	Name of the signal
source	**BlifNode**	Pointer to source node which drives this signal
sinks	**List(BlifNode)**	List of pointers to this node's sinks.

Table 3.6: Fields in Signal object

BlifNode

Contains the names of the input and output Signals, as well as the properties of the node (type, etc). Does not contain direct references to Signals, merely their names.

Signal

Contains references to the signal source, and a list of its sinks. Also stores the signal name.

DFG Traversal

BlifNodes represent nodes in the DFG while Signals represent a collection of edges with common source. Traversing the network is thus achieved through traversing from node→ signal→ node. However, BlifNodes do not store a pointer to the Signal, just the name of the Signal. The actual Signal object, being specific to a partition while BlifNodes are not (nodes can be added to and removed from Models with no issue, and can even exist in multiple at once e.g. original circuit and subpartition). This means that signals must be looked up in the partition by name. To this end, Model contains a field *signals* which is a map from signal name to Signal.

Thus, an example which recursively traverses from a node to its children would be:

Given a Model which represents the circuit as a DFG and contains a list of nodes, map of signal name → Signal, and lists of primary inputs and outputs for the circuit, each node contains the names of its input and output signals, allowing the Signal to be looked up, and the Signal contains pointers to its source and sink nodes. This allows the DFG to be traversed by going from node, to signal, to node, etc. A BlifNode represents the information in a circuit element declaration within a BLIF file,

Algorithm 1 Example Traversal

```
procedure ExampleTraversal(startNode, partition)
    ▷ Get the name of our output signal
    outputName ← startNode.output

    ▷ From a signal name, get the Signal object
    outputSignal ← partition.signals[outputName]

    ▷ Retrieve the sinks of a signal, which are also the immediate children of our start node
    for all childNode ∈ outputSignal.sinks do
        Print("Reached childNode from startNode")

        ▷ Recursively visit the child
        ExampleTraversal(childNode, partition)
    end for
end procedure
```

which includes only the name of its input and output signals. The actual Signal itself is a separate circuit specific construct designed to allow for ease of traversal of the circuit as a DFG. As such, we don't directly point to signals from a BlifNode, as the Signal depends on the circuit context.

3.2 Algorithm

Main

Partition, Triplicate, Join and Flatten are all implemented in separate programs. Main is responsible for taking an input file and running it through our toolchain to produce a TMR'd output file.

Algorithm 2 Main Algorithm

Variable	Type	Description
$input$	**File**	Input BLIF file
$targetRecoveryTime$	**Float**	Per partition recovery time (in seconds)
$files$	**List(File)**	circuit partitions, one per file
$file$	**File**	
$header$	**String**	string containing the first three lines of the input file
$output$	**File**	output file

1: **procedure** MAIN($input$, $targetRecoveryTime$)
2:     $baseClockPeriod \leftarrow$VPR$(input)$
3:     $files \leftarrow$ Partition$(input, targetRecoveryTime, baseClockPeriod \times 1.8)$
4:     **for all** $file \in files$ **do**
5:         $file \leftarrow$ Triplicate$(file)$
6:     **end for**
7:     $header \leftarrow input.lines[0 \rightarrow 3]$
8:     $file \leftarrow$ Join$(files, header)$
9:     $output \leftarrow$ Flatten$(output)$
10: **end procedure**

We're given a BLIF file as input. First, in line 2 we run the original circuit through VPR to determine the clock period of the base circuit. In line 3 we partition the input circuit into a number of sub circuits, each in a separate file, as further expanded in Algorithm 3, passing it our target recovery time, and an estimate of the final circuit's clock period. Then in lines 4-5 for each partition file we read it in as a black box, triplicate it, insert voting logic, and write it back out. Next in line 7 we extract the original header, which provides the name, inputs and outputs of the original circuit. We then, in line 8, join all the partitions together with the original name, inputs and outputs (in the same order), as the original circuit, and finally line 8 flattens the circuit, i.e. transforms the generated hierarchical netlist into a flat netlist with only one main model, or circuit, and no submodels.

Variable	Type	Description
$file$	**File**	input file
$targetRecoveryTime$	**Float**	maximum per partition recovery time (in seconds)
$estimatedClockPeriod$	**Float**	An estimate for the final clock period of the partitioned circuit
$blif$	**Blif**	In-memory representation of input BLIF file
$circuit$	**Model**	Main circuit from input file, represented as DFG
targetPartitions	**Integer**	An estimate for the number of partitions
$partition$	**Model**	Circuit, which we are adding nodes to, to make our partition
$queue$	**Queue**	FIFO queue of nodes to visit
$visited$	**Map(BlifNode→ Boolean)**	Map of whether a BlifNode is visited
$signal$	**Signal**	
$circuit.outputs$	**List(Signal)**	List of output Signal of a circuit
$signal.source$	**BlifNode**	Node which drives this Signal
$queue.size$	**Integer**	Number of nodes in queue
$node$	**BlifNode**	
$file$	**File**	
$files$	**List(File)**	
$numPartitions$	**Integer**	Counter of number of partitions
$signalName$	**String**	Name of a Signal
$node.inputs$	**List(String)**	List of names of signals which are inputs to this node
$model.signals$	**Map(string → Signal)**	Map from signal name to Signal representing it in that Model

Table 3.7: Variables for Partition

Partition

Given an input file, Partition reads it in, and splits it into a number of smaller subcircuits, each of which has a maximum recovery time of our target recovery time or less. Each subcircuit is then output to its own separate file, each of which is a valid BLIF circuit on its own.

Algorithm 3 Partition

```
procedure PARTITION(file, targetRecoveryTime, estimatedClockPeriod)
    blif ← new Blif(file)                                ▷ Read in file
    circuit ← blif.main                                  ▷ The actual circuit within the BLIF file
    numPartitions ← 1
    repeat
        targetPartitions ← numPartitions
        numPartitions ← 1
        partition ← new Model                            ▷ Empty Circuit
        queue ← new Queue                                ▷ Empty Queue
        visited ← new Map(BlifNode → bool, DEFAULT: false)
        for all signal ∈ circuit.outputs do
            queue.Enqueue(signal.source)
        end for
        while queue.size > 0 do
            node ← queue.Dequeue()
            if visited[node] = true then
                continue                                 ▷ Handle each node once and only once
            end if
            visited[node] ← true
            partition.AddNode(node)
            if RecoveryTime(partition, targetPartitions, estimatedClockPeriod) > targetRecoveryTime then
                partition.RemoveNode(node)
                MakeIOList(partition, circuit)
                file ← partition.WriteToFile()
                files ← files ∪ file
                numPartitions ← numPartitions + 1
                partition ← new Model                    ▷ Empty Circuit
                partition.AddNode(node)
            end if
            for all signalName ∈ node.inputs do
                signal ← model.signals[signalName]
                queue.Enqueue(signal)
            end for
        end while
        if partition.size > 0 then
            MakeIOList(partition, circuit)
            file ← partition.WriteToFile()
            files ← files ∪ file
        end if
    until numPartitions ≤ targetPartitions
    return files
end procedure
```

Line 2 reads a BLIF into memory, representing it as a DFG. Lines 14-18 ensure that we visit each node only once, and thus that each node is in exactly one partition, by checking if a node has been visited before and if so, skipping it, otherwise marking it as visited and continuing. Lines 20/28 insert the current node into the open partition, cutting any created cycles and updating values such as critical path length (also known as the number of register stages) as outlined in Algorithm 6. Line 21 tests if the current partition recovery time is greater than our specified limit, with the algorithm used to calculate the recovery time given in Algorithm 5. If the partition's recovery time exceeds our target we execute lines 22-28, where we remove the just added node to bring our recovery time back under the limit, and then write the partition to a file. Line 23 calculates which signals are primary inputs or outputs for the partition, and promotes them accordingly, with more detail given in Algorithm 4. Writing the partition to a file simply involves outputting the name, inputs, outputs, and a list of every node in the partition in BLIF format. RemoveNode, on line 22, merely removes the node from the partition's list of nodes rather than fully reversing everything AddNode does. WriteToFile simply serialises the inputs, outputs and node list. Lines 35-39 write out the final partially full partition, if there is one. Again, WriteToFile simply outputs the circuit name, list of inputs, outputs and clocks, and list of nodes, with no further processing required. In line 40, we now check if our estimate $targetPartitions$ was correct. As long as the actual number of partitions is less than or equal, our recovery time calculation was fine, and we can return the generated files and proceed. If we underestimated the number of partitions repeat the entire process with (assigned on line 6) our new target as the previous number of partitions.

MakeIOList

Given the original circuit and a subpartition, promote any signals which are sourced or sunk outside of the subpartition to a primary input or output of the subpartition.

Algorithm 4 MakeIOList

Variable	Type	Description
$partition$	**Model**	Partition to create list of primary inputs and outputs for
$originalCircuit$	**Model**	Original model
$signal$	**Signal**	
$signal.source$	**BlifNode**	The driver for the signal
$partition.inputs$	**List(BlifNode)**	List of primary inputs for the circuit
$partition.signals$	**Map(String → Signal)**	Map from signal name to Signal
$originalCircuit.signals$	**Map(String → Signal)**	Map from signal name to Signal
$signal.sinks$	**List(BlifNode)**	List of sinks for the signal

1: **procedure** MAKEIOLIST($partition, originalCircuit$)
2:     **for all** $signal \in partition.signals$ **do**
3:         **if** $signal.source = NULL$ **then** ▷ If this signal has no driver
4:             $partition.inputs.Add(signal)$
5:         **end if**
6:         $otherSignal \leftarrow originalCircuit.signals[signal.name]$ ▷ Get the corresponding signal in the original circuit
7:         **if** $\text{count}(otherSignal.sinks) - \text{count}(signal.sinks) > 0$ **then** ▷ If the signal has more sinks in the original circuit than it does in this partition
8:             $partition.outputs.Add(signal)$
9:         **end if**
10:     **end for**
11: **end procedure**

We iterate through every signal in our partition. For each one we check if we have a source (line 3), if not it must be a primary input. Similarly, on line 7 we check if we have a sink which is not represented within our partition. If so, promote it to a primary output of the partition.

So for example, in Figure 3.2 signal 2 has no source within the partition, and so is promoted to primary input. Signal 3 and 4 both have outputs outside the partition, and so are promoted to primary outputs.

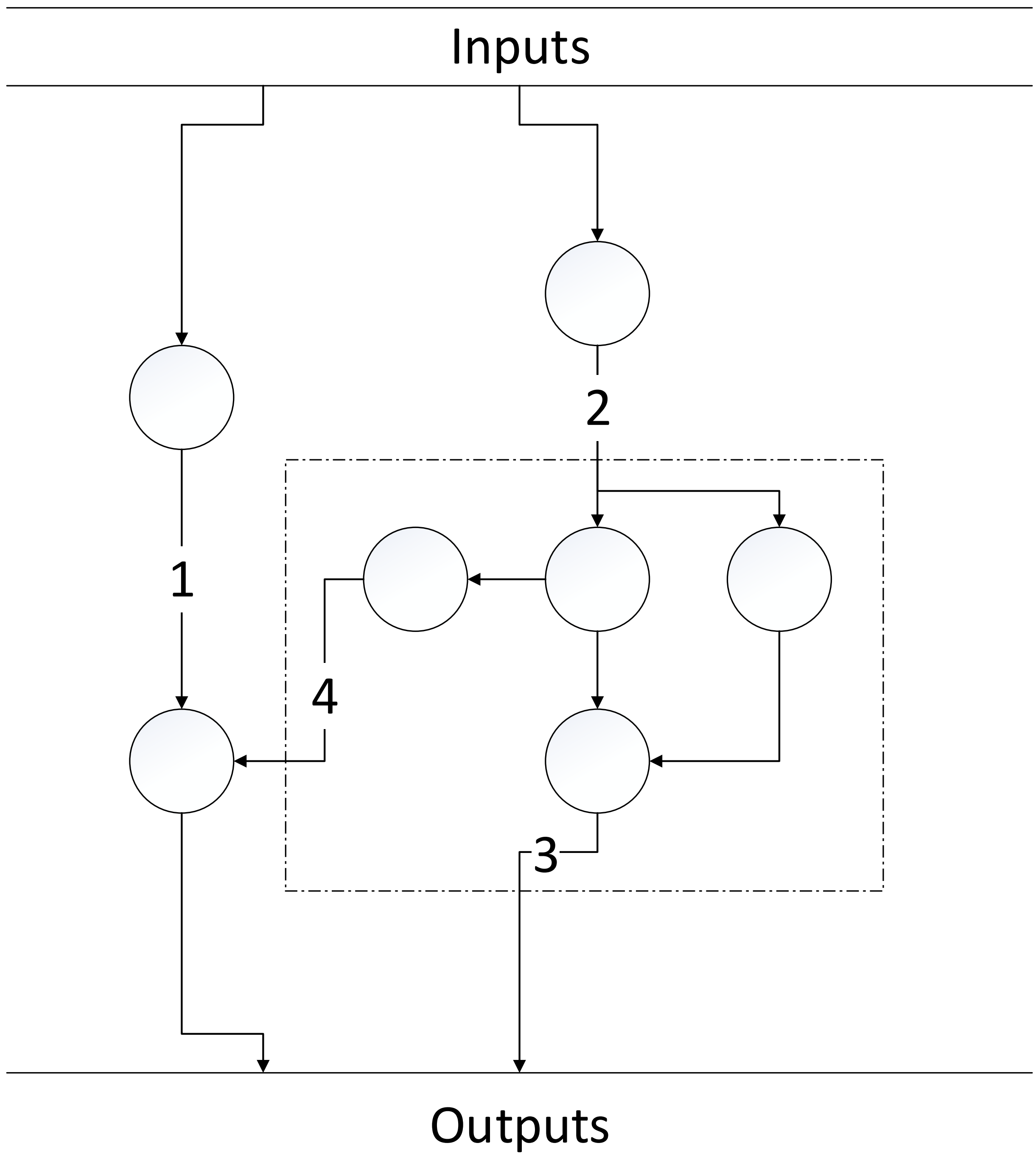

Figure 3.2: MakeIOList

RecoveryTime

For a given partition, calculate its error recovery time. The derivation of this algorithm and the values

Algorithm 5 RecoveryTime

Variable	Type	Description
$partition$	**Model**	The partition to calculate the recovery time for
$numPartitions$	**Integer**	Estimated final number of partitions
$clockPeriod$	**Float**	Estimated clock period of final circuit
$latency$	**Float**	Circuit latency (i.e. time for input to completely propagate to output) in seconds
$clockPeriod$	**Integer**	Estimated period of the final circuit, in seconds. This is estimated as $1.8\times$ the clock period of the original circuit
$criticalPath$	**Integer**	Maximum number of steps between an input and an output
$numFF$	**Integer**	Number of Latches in circuit
$numLUT$	**Integer**	Number of look up tables in circuit
$resynchronisationTime$	**Float**	Time, in seconds, that it takes to resynchronise circuit
$detectionTime$	**Float**	Time, in seconds, that it takes to detect an error
$reconfigurationTime$	**Float**	Time, in seconds, that it takes to reconfigure circuit
$communicationTime$	**Float**	Time, in seconds, that it takes to transmit reconfiguration request to controller

procedure RECOVERYTIME($partition, numPartitions, clockPeriod$)
    $latency \leftarrow clockPeriod \times (criticalpath + 1)$
    $detectionTime \leftarrow latency$
    $resynchronisationTime \leftarrow latency$
    $reconfigurationTime \leftarrow \max(numFF, numLUT)/160 \times 1.48^{-5}$
    $communicationTime \leftarrow 5 \times 50 \times (numPartitions + 1) \times clockPeriod$
    $recoveryTime \leftarrow detectionTime + resynchronisationTime + reconfigurationTime + communicationTime$
    return $recoveryTime$
end procedure

used is fully discussed in Section 1.2. The criticalpath is a measure of the maximum number of latches on a path from input to output. The +1 is to account for the contribution of combinational logic, which may be up to one additional clock cycle of latency. $numPartitions$ and $clockPeriod$ as passed to this function are calculated as per Section 2.2.

AddNode

Insert a node into an existing partition, or circuit, while updating appropriate parameters (i.e. maximum path length and signals) which are depended upon by other components (i.e. recovery time calculation and DFG traversal respectively). Additionally, detect any newly created cycles and cut them. This ensures that the circuit is always an acyclic graph with every node reachable.

Lines 3-13 update the appropriate signals, adding the node as a source or sink to the relevant signals if they exist within the partition, or creating them implicitly if they don't already exist. Line 4 checks if the input signal referred to has been renamed by CutSignal in Algorithm 8. If it has, retrieve the new name for the signal and rename the input signal accordingly. CutSignal only renames inputs, not outputs, thus this check only needs to be performed for circuit inputs. Lines 14-22 then update the maximum path length (or latency in clock cycles) while detecting and cutting any newly created cycles.

Algorithm 6 AddNode

Variable	Type	Description
$partition$	**Model**	Model containing DFG representing partition to add node to
$node$	**BlifNode**	Node to add
$signal$	**Signal**	
$signalName$	**String**	Name of a Signal
$newName$	**String**	The new name of a Signal if and after it's been cut
$partition.signals$	**Map(String $\rightarrow$ Signal)**	Map of signal name to Signal
$signal.sinks$	**List(BlifNode)**	List of sinks for a Signal
$signal.source$	**BlifNode**	Source, or driver, for a Signal
$inCost$	**Integer**	Maximum number of critical path steps to reach node, not counting the node itself
$explored$	**Map(BlifNode $\rightarrow$ Boolean)**	Whether a node has been reached yet in the current iteration

procedure ADDNODE($partition, node$)
    $nodes.insert(node)$
    for all $name \in node.inputs$ **do**
        if IsRenamed($signalName$) **then**
            $newName \leftarrow$ GetNewName($name$) ▷ If this signal has been renamed already to avoid a cycle, rename this occurrence of it.
            Replace($node.inputs, signalName, newName$) ▷ Replace the original name with what it was renamed to
            $signalName \leftarrow newName$
        end if
        $signal \leftarrow partition.signals[signalName]$
        $signal.sinks.Add(node)$
    end for
    $signal \leftarrow partition.signals[node.output]$
    $signal.source \leftarrow node$
    $inCost \leftarrow 0$
    for all $signalName \in node.inputs$ **do**
        $signal \leftarrow partition.signals[signalName]$
        $source \leftarrow signal.source$
        if $partition.costs[source] > inCost$ **then**
            $inCost \leftarrow partition.costs[source]$
        end if
    end for
    UpdateCostsAndBreakCycles($partition, node, NULL, node, inCost, explored, costs$)
end procedure

UpdateCostsAndBreakCycles

Recursively traverse our network to update maximum path lengths to account for our new node and additional paths. While traversing the network, detect and break any cycles we encounter. This turns a possibly cyclic DFG with partially computed path lengths, into an acyclic DFG—or Directed Acyclic Graph (DAG)—with fully computed path lengths.

We care about two things. One, the maximum cost to reach a node, and two, detecting and removing any cycles. Given an existing DAG which we insert a new node into, then

1. The new node is the root node of a subgraph within the DAG.
2. Nodes which are not within the subgraph cannot have the maximum cost to reach them change (as nothing has changed in any path to them).
3. Any cycles must pass through the new node, as all the new edges are to or from the new node.
4. Correspondingly, without any cycles the root node will only be reached once at the start.

Consider Figure 3.3 where every node is a latch with cost to reach indicated. Our new node (filled in) is added to an existing DAG. Our new node should now be the root of a subtree which includes all nodes reachable from our new node i.e. all nodes except those crosshatched which are unreachable from our new node. We now traverse our DFG recursively, updating the maximum cost to reach each node as we travel. Eventually, in our example we reach our newly added node again indicating a cycle. We thus cut the cycle as detailed in Algorithm 8, recurse back a step, and continue until the entire DFG has been traversed, at which point all cycles have been cut, and all nodes have the maximum path length to them updated.

Using this information we develop our traversal algorithm. Line 2 demonstrates an optimisation, in that once a path has been checked we need not recheck it unless we have found a more expensive path to it as otherwise nothing will change. Lines 5-9 check if we have detected a cycle. If so, cut it through cutting the signal, which splits the signal into two: A primary output with the same source, and a primary input with the same sinks, as detailed further in Algorithm 8.

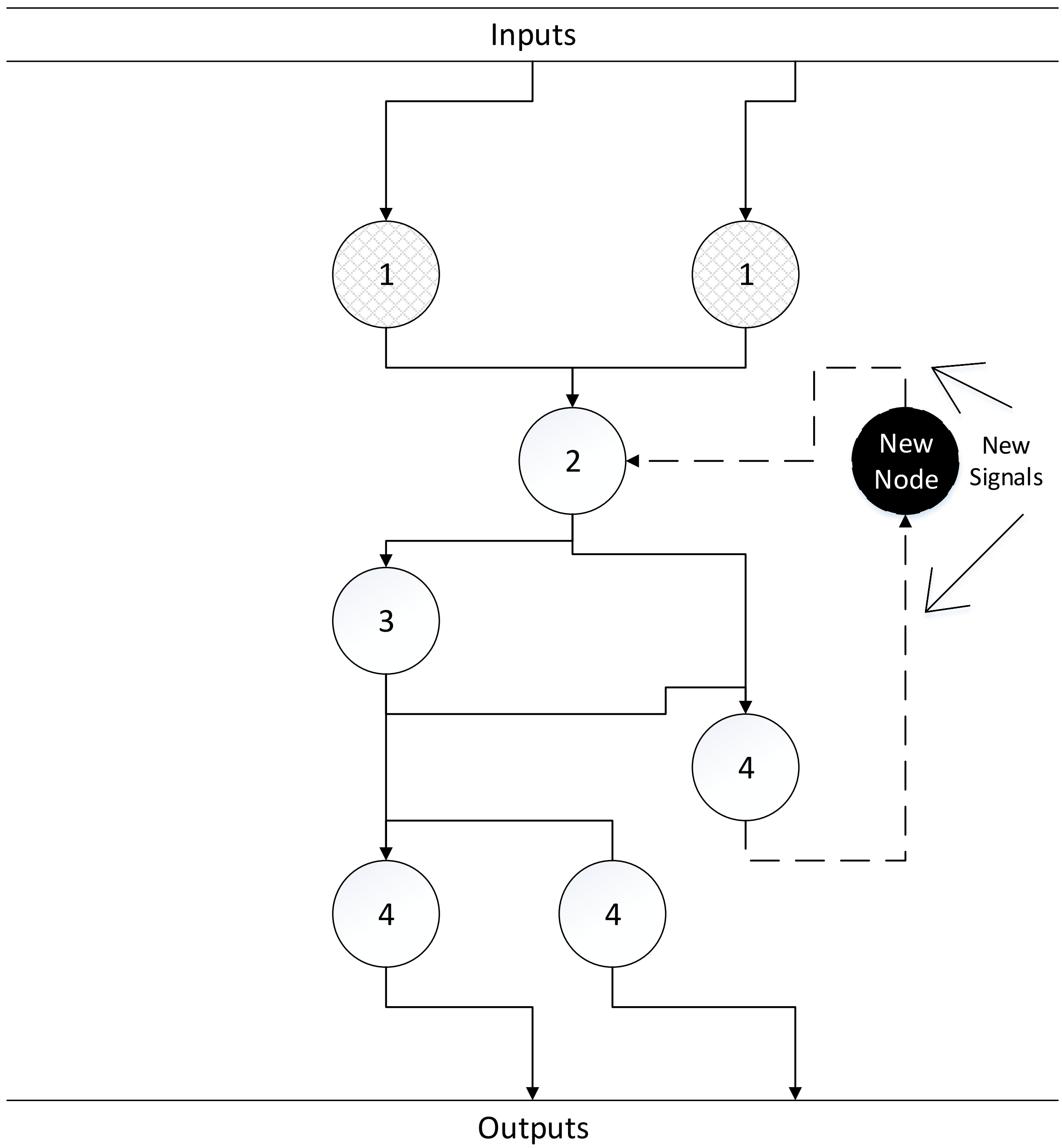

Figure 3.3: AddNode

Algorithm 7 UpdateCostsAndBreakCycles

Variable	Type	Description
$partition$	**Model**	Model containing DFG representing partition to add node to
$root$	**BlifNode**	Newly added node
$parent$	**BlifNode**	Node we just came from
$costToReach$	**Integer**	Maximum number of critical path steps to reach node, not counting the node itself
$explored$	**Map(BlifNode $\rightarrow$ Boolean)**	Whether a node has been reached yet in the current iteration
$partition.signals$	**Map(String $\rightarrow$ Signal)**	Map of signal name to Signal
$parent.output$	**String**	Name of the signal the parent nodes drives i.e. the signal we reached this node from
$signal$	**Signal**	Signal we reached this node from
$node.cost$	**Integer**	1 for latches, 0 for LUTs
$costs$	**Map(BlifNode $\rightarrow$ Integer)**	Map of the cost to reach each node
$node$	**BlifNode**	
$signal.sinks$	**List(BlifNode)**	List of sinks for a Signal
$cost$	**Integer**	Number of critical path steps to reach node, including the node itself

procedure UPDATECOSTSANDBREAKCYCLES($partition, root, parent, node, costToReach, explored$)
  if $explored[node] = true$ and $costs[node] \geq costToReach$ **then** ▷ No need to continue down this path
    return
  end if
  if $parent \neq NULL$ and $node = root$ **then** ▷ We have a cycle
    $signal \leftarrow partition.signals[parent.output]$ ▷ The signal edge we came in on
    CutSignal($partition, signal$)
    return
  end if
  $cost \leftarrow costToReach + node.cost$
  if $cost > costs[node]$ **then**
    $costs[node] = cost$
  else
    $cost = costs[node]$
  end if
  for all $child \in partition.signals[node.output].sinks$ **do**
    UpdateCostsAndBreakCycles($partition, root, node, child, cost, explored$)
  end for
  $explored[node] = true$
end procedure

CutSignal

Given a signal, cut it by splitting it into two signals, of which one is a newly named primary input with the same sinks as the cut signal had, and the other of which is a primary output with the same source and name as the original signal. Figure 3.4 demonstrates this transformation in action.

Algorithm 8 CutSignal

Variable	Type	Description
partition	**Model**	Model containing DFG representing partition to cut signal within
signal	**Signal**	Signal to cut
newInputSignal	**Signal**	New primary input signal with the sinks of the original
newOutputSignal	**Signal**	New primary Output signal with the source of the original

1: **procedure** CUTSIGNAL($partition, signal$)
2: $newInputSignal \leftarrow newSignal()$
3: $newInputSignal.source \leftarrow NULL$
4: $newInputSignal.sinks \leftarrow signal.sinks$
5: $newInputSignal.name \leftarrow MakeNewName(signal.name)$ ▷ Create a unique signal name through a reversible transformation
6: $newOutputSignal \leftarrow newSignal()$
7: $newOutputSignal.source \leftarrow signal.source$
8: $newOutputSignal.sinks \leftarrow newList$ ▷ No inputs, so assign an empty list
9: $newOutputSignal.name \leftarrow signal.name$
10: $partition.signals[newInputSignal.name] \leftarrow newInputSignal$
11: $partition.signals[newOutputSignal.name] \leftarrow newOutputSignal$
12:
13: **for all** $node \in signal.sinks$ **do**
14: replace($node.inputs, signal.name, newInputSignal.name$) ▷ Replace input signal names in nodes with the new signal name
15: **end for**
16: **end procedure**

Lines 2-5 create a new primary input. It has no source (as it is a primary input of the partition) and the sinks of the original signal. Line 5 generates a new globally unique name for this signal which can be reversed to give the original signal name. In our implementation we prepend a constant string "qqrin" and specify that signal names of this form are reserved, as no benchmarks used signal names in that format. Lines 6-9 create a new primary output. It has no sinks and the source of the original signal. This signal retains the same name as the original signal. Lines 10-14 update all references to the old signal to refer to the appropriate new signal.

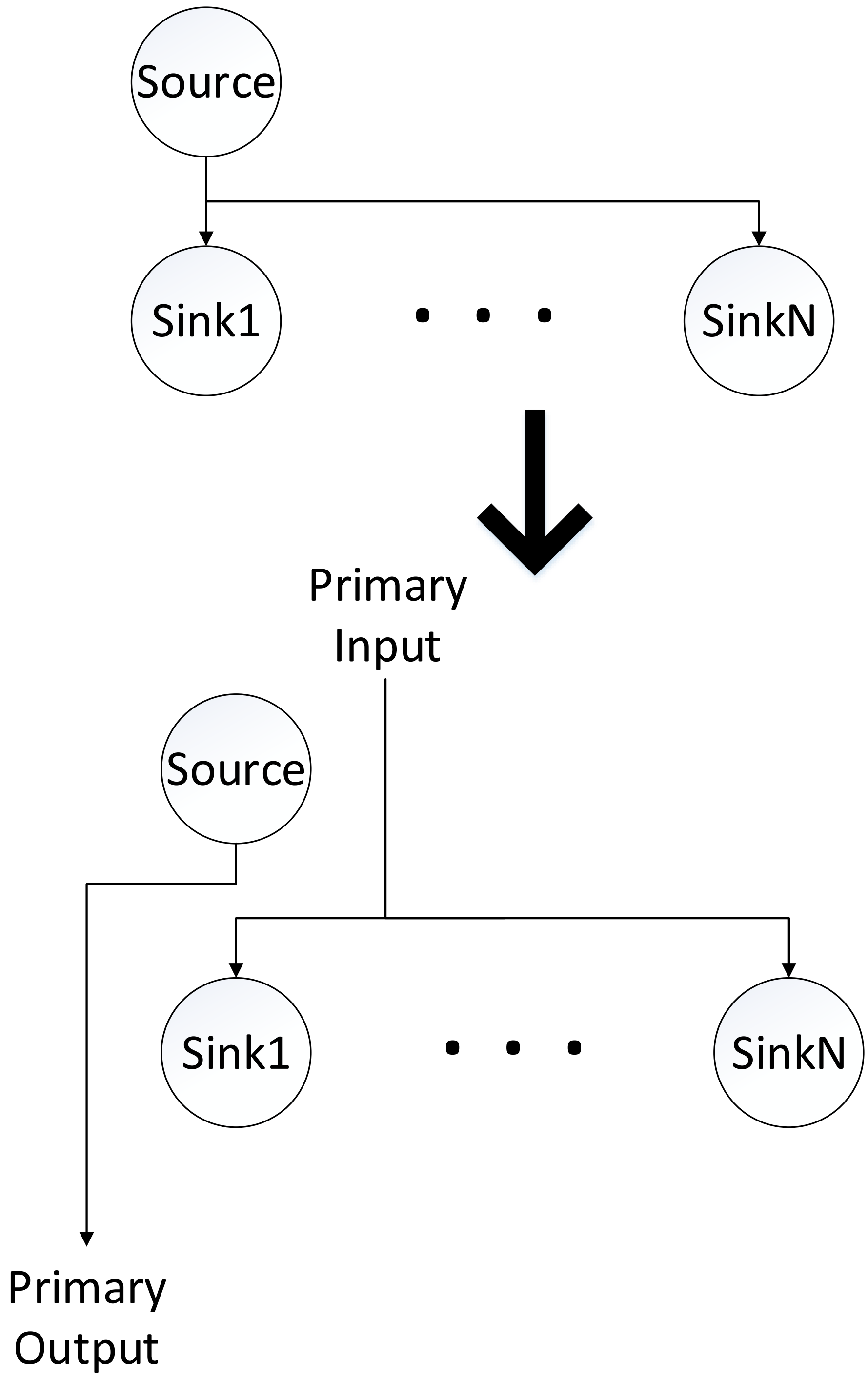

Figure 3.4: CutSignal

Triplicate

Given a file containing a partition, read it in as a black box, triplicate it, add voter logic and write it back out to file.

This method operates on the BLIF in a low level way, dealing with manipulating the actual file contents, rather than operating on an abstract circuit representation, as we transform a flat circuit into a hierarchical circuit, in which our original flat circuit remains untouched but we insert voting and similar logic around it.

This is done through reading in the circuit, instantiating three copies of it, and inserting these copies into a template which includes a voter circuit. We then generate the appropriate signals to connect the circuit inputs to each of the three identical partitions, the partition outputs to the voter, and the voter output to the circuit output. This new circuit is then written back out to file. Figure 3.5 represents this process graphically.

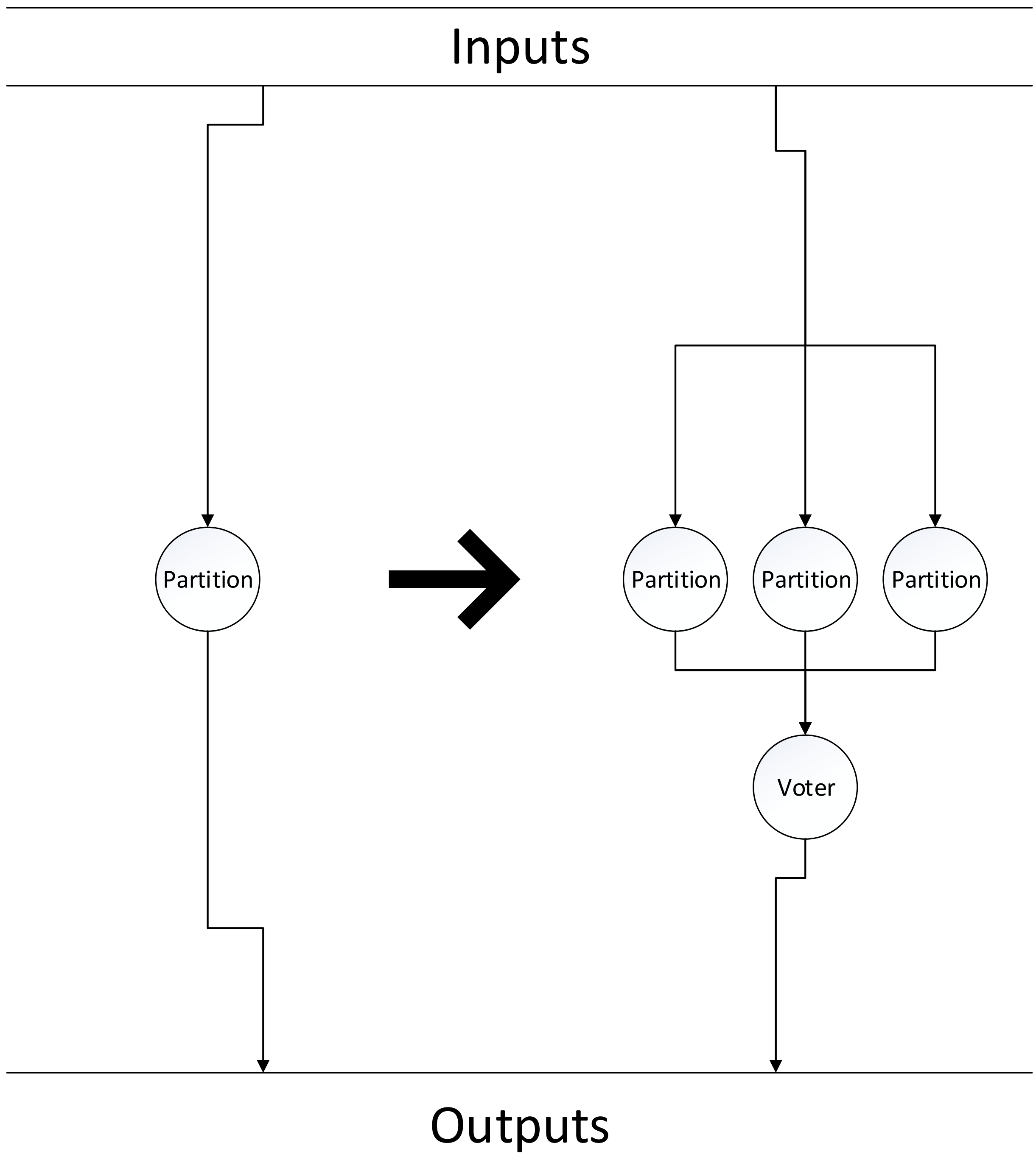

Figure 3.5: Triplicate

Join

Given a list of BLIF files, joins them all together into one circuit. Figure 3.6 represents the transformation graphically.

This is accomplished through reading in each circuit from the input files, and instantiating each of these circuits as a black box subcircuit in a hierarchical design. The top level circuit is created with the same inputs and outputs as the original circuit, and contains a reference to each subcircuit. The inputs and outputs from each circuit are promoted up to the top circuit level, and all signals with same same name are tied together. As signal names are required to be globally unique, and are preserved through the partitioning process (recall that while cutting signals renames them, these renamed signals are internal to a set of triplicated partition and voter and thus not visible to the top level circuit) this then joins all the subcircuits up correctly, and to the appropriate circuit primary inputs and outputs. This process is representing graphically in Figure 3.7.

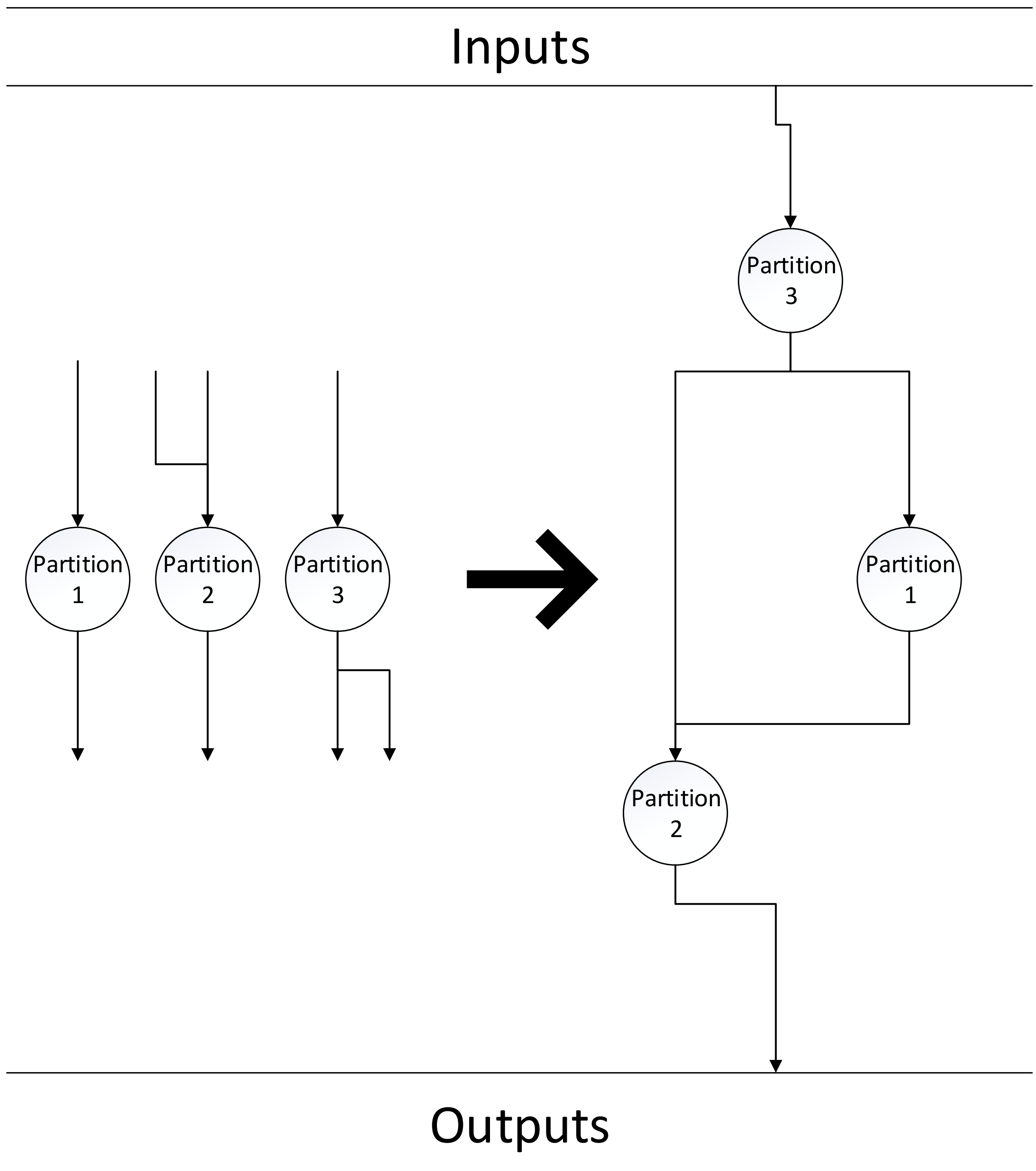

Figure 3.6: Join high level overview

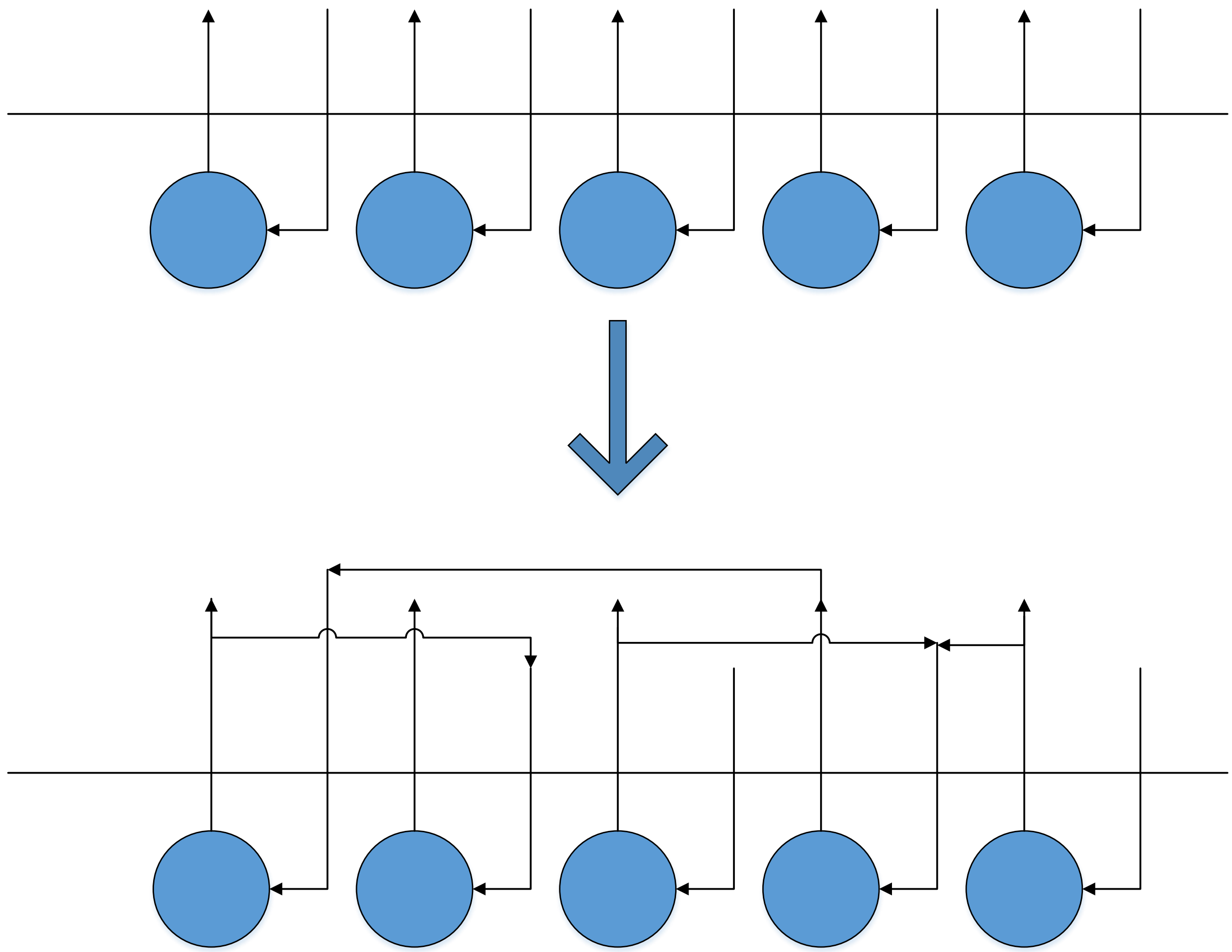

Figure 3.7: Joining Signals

Flatten

Given a hierarchical BLIF file, run it through ABC to flatten it, and postprocess if necessary.

By default ABC flattens input files but performs no other optimisations, therefore we can call `./abc -o output -c echo input` to read in `input`, flatten it, and write it to `output`. Unfortunately, there exists a bug in ABC where clock information is stripped from latches. To circumvent this we require that all latches have the same clock information (clock name, trigger, initial state), which holds for all of the twenty largest MCNC benchmarks, and then use grep and sed to extract the clock information from the original circuit and edit it back into the flattened circuit.

Test

ABC is also used to optionally test the generated circuit to verify that it is equivalent to the original. It does this by creating a miter circuit, which is derived by pairing inputs for the two circuits, and feeding output pairs into an XOR gate which are then OR'd to produce the single output. For any given input, the miter circuit output is 0 if both circuits produce the same set of outputs for the input

Algorithm 9 Flatten

Variable	Type	Description
$file$	**File**	File to flatten
$clockInfo$	**List(String)**	List of latch parameters, including clock name, etc

procedure FLATTEN($file$)
    `./abc -o output -c echo file`
    $clockInfo \leftarrow$split(`grep -m 1 '.latch' file`)
    if $clockInfo$ **then**
        `sed -ri 's/(\.latch.+)(2)/\1 '` + $clockInfo[3]$ + `' '` + $clockInfo[4]$ + `' 2/' output`
    end if
end procedure

set, and 1 if the outputs differ, which turns verification into a Boolean Satisfiability Problem (SAT). The circuits are then simplified by merging equivalent nodes, removing redundant logic and testing inputs. This proceeds until a counter example is found, or the circuit is shown to have constant output 0 for all possible inputs [18,19]. While solving a SAT is NP-complete, in practice the large amount of redundancy in TMR'd circuits allows testing to complete in only a few seconds for the twenty largest MCNC benchmarks.

3.3 Performance

The algorithm must visit each node in the input circuit once to add it to a partition, giving a factor of n. Additionally, for each node added to a partition, in the worst case every other node already in the partition must be visited to detect cycles and update costs, making AddNode worst case linear in the number of nodes in the partition. Constructing the list of inputs and outputs takes time proportional to the number of signals in the partition. In practice, the number of signals will be approximately equal to the number of nodes (each node drives one signal, plus the number of inputs to the circuit). This gives us worst case $O(n^2)$.

Note that this does not include the contribution from rerunning the partitioner as we update our estimate for the number of partitions. This depends on the maximum number of partitions (as the estimate can only be revised upwards) which is a function of the number of circuit elements, giving worst case of at most $O(n^3)$.

Triplicating is linear in the number of inputs and outputs, joining is $O(nk)$ where n is the number of circuits, and k is the number of inputs and outputs for each circuit. In practice, for the twenty largest MCNC benchmark circuits each step is sub-second compared to VPR's running time of up to an hour for some TMR'd benchmark circuits, as outlined in Section 4.6.

3.4 Correctness

A threefold approach to verifying the correctness of the implementation was taken. Firstly, small sample circuits were partitioned and the resulting circuits were examined manually to verify correct operation. Manual verification is, however, not practical for all but the smallest circuits so the small sample circuits were generally just used to test specific corner cases, while two other methods were used to check the benchmarks. As detailed earlier in this section, ABC was used to verify that the generated circuits were functionally equivalent. That is to say, for any set of inputs both the original and TMR'd circuit had identical outputs. Next, circuit properties such as number of elements could be examined and compared to expected results, as is done in Section 4.2. One additional incidental test was verification that the generated file is a valid BLIF file. VPR and ABC are both quite picky and generally either error out or crash on circuits which don't exactly match the expected format.

3.5 Design Choices

As much as possible, we would like our implementation to be easily extensible to multiple architectures. The actual partitioner operates on a DFG so it can be mostly architecture agnostic, only requiring the estimation functions to be architecture aware. From initial steps in this thesis we wrote Python scripts capable of performing basic operations on BLIF files which were used as the basis for Triplicating and Joining. Given time the functionality of each step (partition, triplicate, etc) could all be combined in one program; however it was considered a much lower priority than creating a working reference implementation.

Other design choices include deciding on VPR due to its open nature as discussed earlier in Section 1.3, and how we traverse our DFG. A depth-first traversal as we ended up using tends to generate long narrow pipelines within each partition, thus increasing the number of register stages but reducing the number of inputs and outputs for each partition, whereas a breadth-first traversal lends itself to fewer register stages for the same number of nodes but more inputs and outputs (and hence voters) for each partition. Benchmark results comparing the two options can be found in Section 4.8. A possible future improvement is implementing a more advanced traversal algorithm, for example A* with an appropriate heuristic could allow for more elements per partition.

Additionally, we are faced with a choice as to when in the CAD process to partition. The closer to the end of the process the more control we have, and the better our ability to estimate area and timing, but the harder it is to partition. As we are inserting new elements we want to partition before packing/placement to allow VPR to pack and place our inserted elements.

Choice of Language

We have used a combination of languages, mainly Python and C++. Language choice primarily came down to preference regarding familiarity and personal taste although a few other considerations were kept in mind. For BLIF joining and insertion of the voting logic Python was used. BLIF files are plain text and the text parsing to join and insert is computationally simple, so the primary concern was short

```
.model voter
   .inputs in1 in2 in3
   .outputs out1 out 2
   .clock clock
   .names in1 in2 in3 out1
   11- 1
   1-1 1
   -11 1
   .latch in1 out2 re clock 1
   ...
   commands
   ...
   .end
```

Listing 3.1: BLIF file layout

Model name:	.model ⟨Name⟩	The name of the model.
Input List:	.inputs {SignalName}	The model inputs.
Output List:	.outputs {SignalName}	The model outputs.
Clock List:	.clock {SignalName}	The model clocks.
LUT:		.names {InputSignals} ⟨OutputSignal⟩ {Line}
Latch:		.latch ⟨InputSignal⟩ ⟨OutputSignal⟩ [Trigger ClockSignal] [InitialState]
Optional End Marker:		.end

Table 3.8: BLIF commands

development time while still being readable and maintainable (although Python's performance on text is still quite reasonable) [22]. For the actual partitioner C++ was chosen for a few reasons. Firstly, it was expected that the area and time estimations could be quite computationally expensive, so a lower level compiled language was chosen for performance reasons [22]. Secondly, VPR is written in C, so using C or C++ allowed for easy code reuse, or merging the partitioner and VPR. Our reason for choosing C++ over C was that we preferred an object oriented language as we felt it would be easier to maintain, and would better lend itself to our goal of extensibility, as well as its libraries making our implementation much easier.

3.6 Input File Format

The BLIF file format is a textual format which describes an arbitrary sequential or combinational network of logic functions [24]. Our partitioner only supports a subset of the BLIF specification, specifically only those elements supported by VPR and used in our benchmark files. A sample BLIF file is included in Listing 3.1 and Table 3.8 lists the supported commands and their meanings.

{Name} indicates 1 or more of Name. ⟨Name⟩ indicates a compulsory field. [Name] indicates an optional field. A combinational logic element (.name) is followed by one or more lines describing the logic function it implements. However, our partitioner only cares about node type and the signal names (named with SignalName above) as it builds and traverses the DFG. All other element information is stored and written back out when the node is written.

VPR only supports flat BLIF files, so only one module declaration is allowed per BLIF file. ABC can be used to flatten BLIF files for use by VPR.

Chapter 4

Results

4.1 Benchmarking Procedure

These results were collected by running benchmark circuits through an automated test suite written in Python by the author. For each benchmark circuit, and each target recovery time, a minimum of 15 repetitions were performed to average out the variability in results due to the stochastic nature of VPR's placement algorithm. The original circuit was run through VPR to collect base results, then the circuit run through our partitioner to TMR it. The TMR'd version was then verified by ABC to check its functional equivalence to the original, and then run through VPR to collect TMR'd results. Each run of VPR used a randomly generated seed for the placer. The mean of the reported values across all successful runs was recorded. The benchmarks used were the 20 largest MCNC LGSynth93 circuits technology mapped to flip-flops and 4-input LUTs, as provided by the open-source Verilog To Routing Project (VTR) project[1] and described in table 4.1. As the number of LUTs is larger than the number of latches for all twenty MCNC circuits we used, the number of BLEs is equal to the number of LUTs. The set of target recovery times used were 10^{-3}, 2.5×10^{-4}, 1.2×10^{-4} and 7.5×10^{-5}s. The voter used is a simple 3-input LUT, which uses one BLE per output signal from each partition. Table B.1 in the Appendix, where each circuit had only one partition, contains equivalent values for area overhead and clock slowdown as if a more traditional TMR approach were used, which simply triplicated the entire circuit allowing our approach the be compared.

Target Architecture

VPR allows us to specify a custom architecture for it to run against in an XML format. We opted for the default architecture detailed in [16] consisting of a grid of CLBs each consisting of ten fully interconnected BLEs, and each BLE having a latch and 6-LUT as illustrated in Figure 4.1. Table 4.2 details the number of primitives (latches and LUTs per CLB. Primarily of interest is that each BLE has 6 inputs and 1 output and each CLB has 33 inputs and 10 outputs.

Our architecture consists of 6-input LUTs while our design has LUTs with fewer inputs so there is potential for multiple LUTs to be packed into one. VPR does not optimise in this way, instead

[1] v1.0: `http://code.google.com/p/vtr-verilog-to-routing/`

	Number of:			
Name	Inputs	Outputs	Latches	LUTs
alu4	14	8	0	1522
apex2	38	3	0	1878
apex4	9	19	0	1262
bigkey	229	197	224	1707
clma	62	82	33	8381
des	256	245	0	1591
diffeq	64	39	455	1494
dsip	229	197	224	1370
elliptic	131	114	1218	3602
ex1010	10	10	0	4598
ex5p	8	63	0	1064
frisc	20	116	924	3539
misex3	14	14	0	1397
pdc	16	40	0	4575
s298	4	6	8	1930
s38417	29	106	1463	6096
s38584.1	38	304	1260	6281
seq	41	35	0	1750
spla	16	46	0	3690
tseng	52	122	385	1046

Table 4.1: Benchmark circuits used

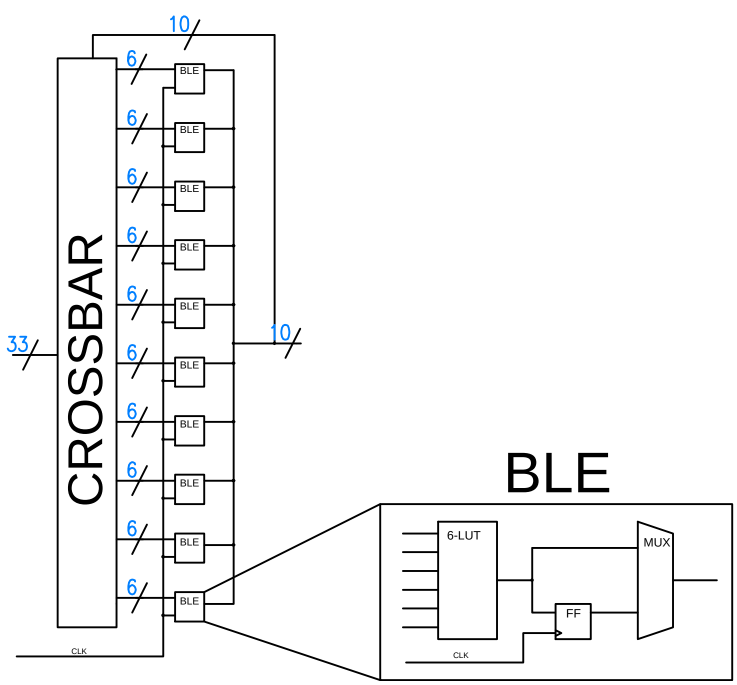

Figure 4.1: CLB Architecture

Component	Number	Notes
Flip Flop	1 per BLE	Shown as FF on Diagram
6-LUT	1 per BLE	
MUX	1 per BLE	
BLE	10 per CLB	
Crossbar	1 per CLB	
CLB	Autosized by VPR	

Table 4.2: Architecture Elements

relying on ABC for optimisation. To confirm this we compared our results to a small set conducted on a similar architecture with 4-input LUTs and found that there while there were slight ($\approx 1\%$) differences in running time and variations between the packed netlist, for the same seed the the final circuit area and clock period are identical.

4.2 Sanity Check

The following results are for the tseng circuit at a target recovery time of 7.5×10^{-5}s. The reported values can be compared to each other as a manual sanity check allowing for additional confirmation of the correct operation of the partitioner.

We are able to confirm all the values which should add up, do. For example:

$$\begin{aligned}
LUTsTMR &= 3 \times LutsBase + \sum PartitionOutputs \\
3730 &= 3 \times 1046 + 305 + 287 \\
3730 &= 3730 \\
LatchesTMR &= 3 \times LatchesBase \\
1155 &= 3 \times 385 \\
NumNodes &= \sum LUTs + \sum Latches = LUTsBase + LatchesBase \\
1431 &= 640 + 406 + 206 + 179 = 1046 + 385 \\
1431 &= 1046 + 385 = 1431 \\
PartitionOutputs &> CutLoops \\
RecoveryTime &= ClockPeriod \times CriticalPath \times 2 + 250 \times (NumPartitions + 1) \times ClockPeriod + \\
&\quad ClockPeriod \times \left\lceil \frac{NumBLEs}{160} \right\rceil \times 1.48 \times 10^{-5} \\
&= 10.9 \times 10^{-9}(2 \times 20 + 750) + 4 \times 1.48 \times 10^{-5} \\
&= 8.649 \times 10^{-6} + 5.92 \times 10^{-5} \\
&= 6.78 \times 10^{-5}
\end{aligned}$$

$LUTs$ and $Latches$ refers to per partition numbers of LUTs and latches respectively. $LUTsBase$ and $LatchesBase$ refer to numbers for the entire circuit. $PartitionOutputs$ is the number of voted-

File	tseng
Number of Nodes	1431
Estimated Latency (ns)	10.9
Partitions	2
Number of Inputs Base	52
Number of Inputs TMR	52
Number of Outputs Base	122
Number of Outputs TMR	122
Number of LUTs Base	1046
Number of LUTs TMR	3730
Number of Latches Base	385
Number of Latches TMR	1155
VPR Duration Base (s)	15.93
VPR Duration TMR (s)	92.99
NetDelay Base (ns)	1.60
NetDelay TMR (ns)	2.30
LogicDelay Base (ns)	4.48
LogicDelay TMR (ns)	6.56
Period Base (ns)	6.08
Period TMR (ns)	8.87

Table 4.3: Detail from one run of tseng, recovery time 7.5×10^{-5}

Recovery Time (s)	Outputs	Inputs	Cut Loops	Latches	LUTs	Critical Path Length
6.78E-05	305	304	206	206	640	20
5.27E-05	287	303	179	179	406	4

Table 4.4: Partition detail from one run of tseng, recovery time 7.5×10^{-5}

on outputs from each partition and is equal to the number of feedforward edges (edges into another partition, or primary outputs from the circuit) and the number of feedback edges, or edges reused within the partition. $CutLoops$ is the number of cut loops on a per-partition basis and is the same as the number of feedback edges. Where a signal is used part of a voted-on cycle (feedback) and in another partition (feedforward) it is only counted once, not twice. $ClockPeriod$ is the estimated clock period and $NumPartitions$ is the number of partitions.

In Table 4.4 Outputs is the number of feedforward edges (signals going to other partitions) $+$ the number of feedback edges (cut cycles). Some other observations from this data: Our estimated clock period was conservative. We estimated 10.9ns when the circuit actually came in at 8.9ns. VPR takes much longer on triplicated circuits than on the original. 6 times longer in this example.

Circuit	Number of Successful Runs
clma	0
s38584.1	0
s38417	0
ex1010	22
pdc	7
spla	25
elliptic	12
frisc	0
s298	25
apex2	25
seq	25
bigkey	25
des	25
alu4	25
diffeq	25
misex3	25
dsip	25
apex4	25
ex5p	25
tseng	25

Table 4.5: Target Recovery Time 7.5×10^{-5}s

Name	NumPartitions	Clock Period Original (ns)	Clock Period TMR (ns)	Slowdown Factor
s38584.1	1	3.22	4.61	1.43
s38584.1	1	2.06	4.94	2.40

Table 4.6: Comparison of slowdown factors between runs with same input parameters

4.3 Stochastic Nature of Placement

As VPR's placer uses simulated annealing which contains a random factor, there was variation between different runs, potentially extremely large such as the example in table 4.6 where one run had a 40% slowdown, while another run with exactly the same set of parameters had a 140% slowdown. All results are the mean across a minimum of fifteen runs unless otherwise noted, and where time permitted a larger number of runs were performed. The appendix contains the number of successful runs for each circuit and target recovery time. Note that for some circuits the number of successful runs was actually below 15. As the partitioner's estimate for final clock period depends on the quality of VPR's place and route on the original circuit, if VPR finds a poor placement then the estimated clock period may be so high that the partitioner is unable to find a valid partitioning. See Table 4.5 for examples.

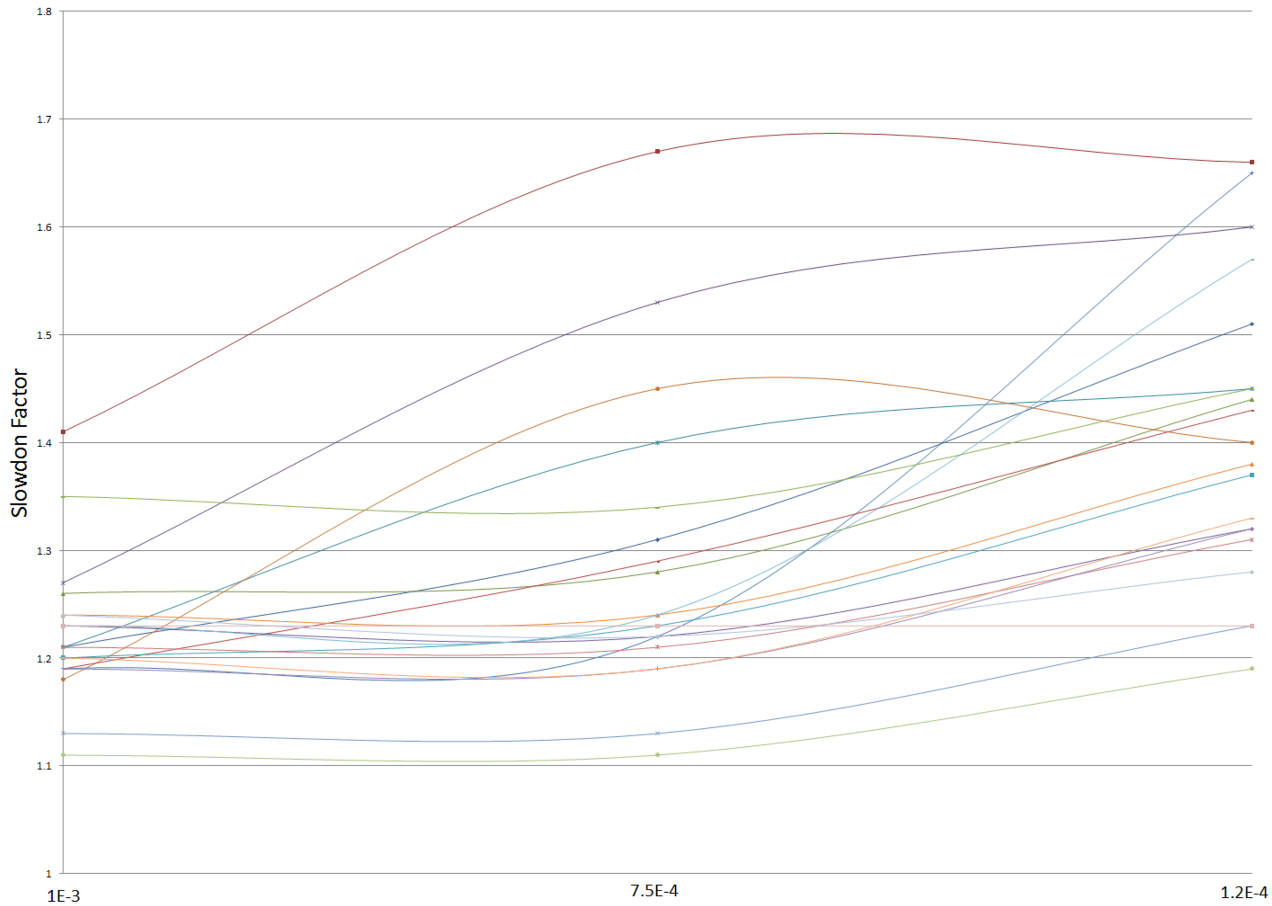

Figure 4.2: Slowdown Factors for each Benchmark at Different Recovery Times

4.4 Area

As expected, area usage is slightly greater than tripled, which corresponds to results in literature [5]. The number of BLEs used is equal to three times the original, plus the total voter area. The larger the number of partitions, the greater the area usage due to the additional voters required. Area increase depends on the circuit and number of partitions, but typical overheads for our approach are around a $3.1\times$–$3.5\times$ with a mean across our benchmark circuits of $3.13\times$ increase while running TMR on the circuit as a whole had a typical overhead of around $3\times$–$3.3\times$ with a worst case mean (out of measured result sets) of $3.26\times$.

4.5 Operating Frequency

In general, the more partitions the slower the resulting circuit, as per Figure 4.2 where each line represents a different circuit. This result is unsurprising, as increasing the number of voters increases the number of signals to be routed increasing congestion. Mean slowdown is around $1.2\times$–$1.65\times$ depending on target recovery time, though it varies considerably from circuit to circuit. This compares favourably with TMR'ing the entire circuit as a whole, which saw typical slowdowns between $1.1\times$–$1.4\times$ For our recovery time calculations, as they required an estimate of the final circuit clock period

Step	Time (s)
VPR Original	225.76
Partition	0.56
Triplicate	0.31
Join	0.07
Flatten	0.35
Test	0.92
VPR TMR	5237.50

Table 4.7: Running times for clma with a target recovery time of 2.5e-4s

we used an estimate of $1.8\times$ the original circuit's clock period. As we can see from the results, in the general case this factor is quite conservative, and we can likely get away with a lower value, say 1.5 for most cases.

4.6 Running Time

As shown in table 4.7, the largest contributor to the running time in our toolflow is VPR, taking several orders of magnitude longer than any other step. Of the time VPR took, routing is generally the largest contributor, followed by placement, followed by packing. Routing for standard FPGA architectures is NP-complete [26] with the specific routing algorithm used by VPR being $O(k^2 \log k)$ per net on average, where k is the number of terminals for the net [6].

4.7 Recovery Time

For the benchmark circuits typical recovery times ranged from 1.2×10^{-4}s to 10^{-3}s. Anything larger is redundant, as the entire circuit fits within one partition, and anything smaller has the circuits unable to be partitioned. The number of partitions and the size of each partition are the two main contributing factors to the recovery time of a partition, therefore the smaller the circuit, the smaller a recovery time its partitions are able to achieve. As the circuit size increases (as measured by the number of BLEs) either the size of each partition, or the total number of partitions must increase, driving up the recovery time. Table 4.8 details the experimentally determined minimum achievable recovery time for each of the twenty largest MCNC benchmark circuits. The estimated final clock period was taken as the mean original clock period for that circuit $\times1.8$. This value was passed to the partitioner with progressively smaller target recovery times until the partitioner was unable to partition whilst meeting the target recovery time. Figure 4.3 shows the same information as a scatter plot, making the correlation between circuit size and minimum recovery time more visible.

Name	Number of BLEs (original)	Clock Period (original) (ns)	Minimum Recovery Time ($\times 10^{-5}$s)
clma	8365	9.21	12.2
s38584.1	6177	4.97	7.80
s38417	6042	6.27	8.50
ex1010	4598	5.92	7.30
pdc	4575	6.47	7.70
spla	3690	6.01	6.50
elliptic	3602	7.72	7.60
frisc	3539	10.95	9.00
s298	1930	8.59	6.10
apex2	1878	5.07	4.50
seq	1750	4.55	4.00
bigkey	1699	2.28	2.80
des	1591	3.97	3.50
alu4	1522	4.54	3.80
diffeq	1494	6.58	4.80
misex3	1397	4.40	3.50
dsip	1362	2.24	2.60
apex4	1262	4.57	3.40
ex5p	1064	4.51	3.20
tseng	1046	5.94	3.70

Table 4.8: Minimum recovery time for circuits

	Channel Width		Network Delay (ns)		Logic Delay (ns)		Clock Period (ns)	
	Base	TMR	Base	TMR	Base	TMR	Base	TMR
Breadth-First	40	58	2.37	3.53	4.16	5.53	6.52	9.06
Depth-First	40	54	1.96	3.53	4.16	4.51	6.11	8.04

Table 4.9: Breadth- vs Depth-first traversals for s38417 with a target recovery time of 2.5e-4s

Recovery Time ($\times 10^{-4}$s)	Number of Outputs	Number of Inputs	Number of cut loops	Number of latches	Number of LUTs	Critical Path Length
2.5	723	859	430	714	2560	17
2.5	1029	921	345	565	2560	9
1.2	421	606	184	184	976	3

Table 4.10: Breadth-first traversal per partition values for s38417 with a target recovery time of 2.5e-4s

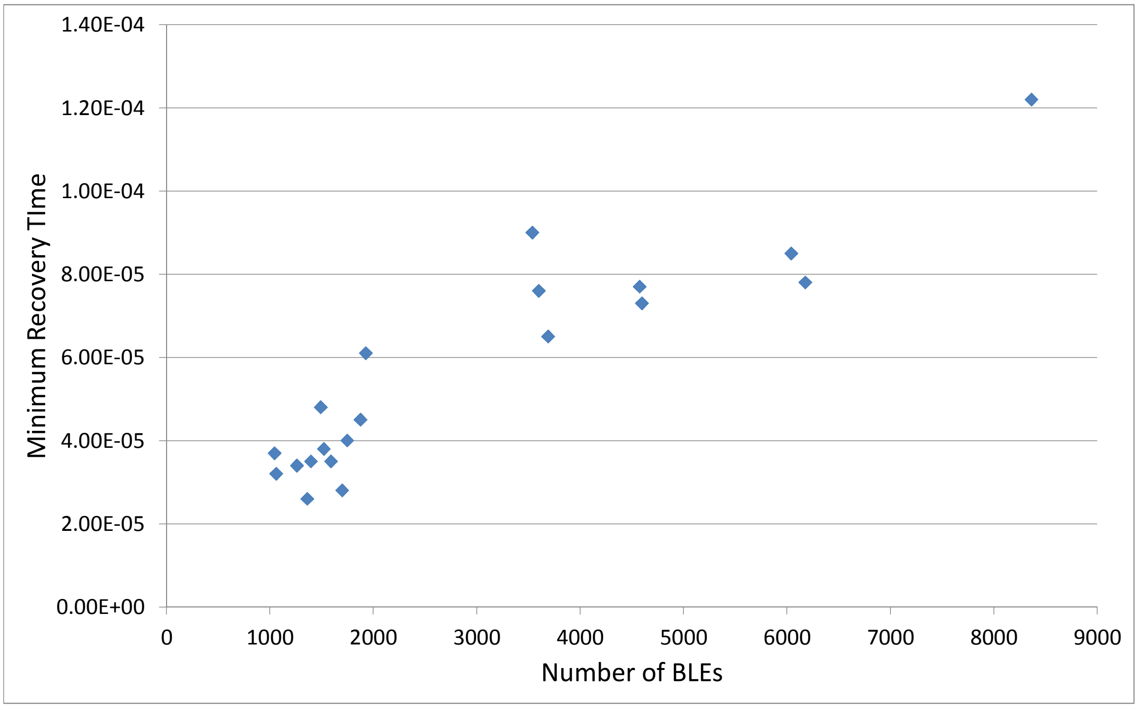

Figure 4.3: Minimum Achievable Recovery Time vs Circuit Size

Recovery Time ($\times 10^{-4}$s)	Number of Outputs	Number of Inputs	Number of cut loops	Number of latches	Number of LUTs	Critical Path Length
2.5	775	663	563	567	2560	30
2.5	596	683	446	627	2560	19
1.2	280	381	150	269	976	10

Table 4.11: Depth-first traversal per partition values for s38417 with a target recovery time of 2.5e-4s

Recovery Time ($\times 10^{-5}$s)	Number of Outputs	Number of Inputs	Number of cut loops	Number of latches	Number of LUTs	Critical Path Length
7.45	20	1318	0	0	480	1
7.45	480	1314	0	0	480	1
7.45	480	986	0	0	480	1
7.45	480	949	0	0	480	1
7.45	480	852	0	0	480	1
7.45	480	633	0	0	480	1
7.45	480	432	0	0	480	1
7.45	480	464	0	0	480	1
7.45	480	592	0	0	480	1
5.97	265	135	0	0	278	1

Table 4.12: Depth-first traversal per partition values for ex1010 with a target recovery time of 7.5e-5s

4.8 Depth- vs Breadth-First Traversal

The values in Tables 4.9, 4.10 and 4.11 are representative samples from a single run. As the partitions generated depend on the estimated clock period which varies from run to run it makes little sense to average per-partition values across runs, especially when the actual number of partitions may vary. As such, while absolute values may change the general trends hold across all runs.

Using a breadth-first traversal algorithm tends to lead to broader partitions, with correspondingly more inputs and outputs for each partition but shorter critical path lengths. Using a depth-first traversal on the other hand leads to deeper but narrower partitions, with fewer inputs and outputs for each partition, but a longer critical path length. As each value which is exported from a partition to be used in another needs to be voted on, this means every feedforward or feedback edge requires another LUT in the final design. As the pipeline depth increases the likelihood of having an internal cycle which needs to be cut increases, so we see more feedback edges, shown by the Cut Loops column in Tables 4.10 and 4.11 also leading to more voters. In general however, breadth-first will have more total LUTs required. As outlined in Table 4.9, breadth-first traversal had an extra 1.02ns of logic delay on the critical path from the extra LUTs, and required slightly wider routing channels to route the design.

The difference is minor for larger partition sizes, however as the number of partitions increases the effect becomes more pronounced, until with some circuits every almost every single circuit element is voted upon when using a breadth-first traversal, as in Table 4.12. Tables B.3 and B.5 in the Appendix list measured slowdowns and area increases for depth- and breadth- first circuits at 1.2×10^{-4}.

Chapter 5

Limitations and Future Work

Our algorithm implementation is still just a first pass at the problem to evaluate its feasibility. There is still much work to be done.

Some notable limitations are that our implementation operates on BLIF files and targets a theoretical simplified architecture. In practice, it would be ideal to use and target industry standard tools, formats and architectures. There are a number of assumptions and approximations made as part of the implementation, especially in the calculation of recovery time; improving the accuracy of approximations allows the partitioner to find better solutions. Additionally, the partitioner itself makes no attempt to find an optimal solution; partitions may be closed off before they're full, or partitions may be unbalanced with some having many more voters or a much longer path length than others. There are some straightforward techniques which can be implemented to improve the partitioner, but the limiting factor tends to be the partition size, and when (as in the benchmark circuits), there are many more LUTs than latches, the ability to reduce the number of partitions through clever partitioning is extremely limited with most partitions being maximally packed already.

In addition to improving the quality of results, the performance of the algorithm can be easily improved in a few areas. The algorithm has a theoretical $O(n^3)$ worst case due to a naive brute force estimation function for the target number of partitions. Improving this to not require repartitioning over and over by using more intelligent estimates can reduce the number of passes needed. There are a few other sections in the implementation where less than optimal approaches were used though they are likely to provide less dramatic gains. Regardless, VPR remains the limiting factor, so optimising the partitioner for speed is not especially helpful as it is already significantly faster than VPR.

An additional limitation is restrictions on input files. The implementation makes some assumptions about the format of input files which, while they hold for all MCNC we used, do not necessarily hold for all valid circuits. For example, that all latches have the same clock. These assumptions will need to be replaced with more robust implementations.

Chapter 6

Conclusion

This thesis was focused on developing a new TMR partitioning algorithm and assessing the effect of the new TMR technique on the performance of the twenty largest MCNC benchmark circuits. From a performance standpoint the algorithm shows promise. While there is still much work to be done the initial results collected in this thesis indicate that the partitioning method described in Chapter 3 and implemented by this thesis is capable of providing more effective fault tolerance with overhead not too much greater than typical TMR solutions as commonly implemented today. Additionally, integrating the algorithm into an existing CAD tool flow should be achievable with negligible design time cost, as it requires no to modification to existing code and the running time is insignificant compared to that of other steps such as routing.

Appendix A

Programs

The following programs were used in this thesis:

Name	Version	Source
ABC	1473	`http://www.eecs.berkeley.edu/~alanmi/abc/`
VTR	v1.0	`http://code.google.com/p/vtr-verilog-to-routing/`
VPR	6.0	Included with VTR

Appendix B

Data

This appendix tabulates the data used to calculate the relationships discussed in this thesis.

Circuit	Number of Partitions	Number of BLEs (original)	Increase in BLE Number	Clock Period (original) (ns)	Clock Slowdown Factor	Number of Successful Runs
clma	1	8365	3.01	9.21	1.21	25
s38584.1	1	6177	3.21	4.97	1.41	25
s38417	1	6042	3.21	6.27	1.26	25
ex1010	1	4598	3.00	5.92	1.27	25
pdc	1	4575	3.01	6.47	1.21	24
spla	1	3690	3.01	6.01	1.18	24
elliptic	1	3602	3.35	7.72	1.19	24
frisc	1	3539	3.27	10.95	1.19	24
s298	1	1930	3.01	8.59	1.35	24
apex2	1	1878	3.00	5.07	1.23	24
seq	1	1750	3.02	4.55	1.20	24
bigkey	1	1699	3.21	2.28	1.24	24
des	1	1591	3.15	3.97	1.13	24
alu4	1	1522	3.01	4.54	1.21	24
diffeq	1	1494	3.28	6.58	1.11	24
misex3	1	1397	3.01	4.40	1.19	24
dsip	1	1362	3.17	2.24	1.23	24
apex4	1	1262	3.02	4.57	1.20	24
ex5p	1	1064	3.06	4.51	1.24	24
tseng	1	1046	3.49	5.94	1.23	24
Mean			3.13		1.22	

Table B.1: Results for target recovery time 1×10^{-3}s

Circuit	Number of Partitions	Number of BLEs (original)	Increase in BLE Number	Clock Period (original) (ns)	Clock Slowdown Factor	Number of Successful Runs
clma	4	8365	3.08	9.35	1.31	25
s38584.1	3	6177	3.27	5.00	1.67	25
s38417	3	6042	3.27	6.29	1.28	25
ex1010	2	4598	3.27	5.94	1.53	25
pdc	2	4575	3.12	6.43	1.40	25
spla	2	3690	3.12	5.89	1.45	25
elliptic	2	3602	3.38	7.70	1.22	25
frisc	2	3539	3.35	10.93	1.29	25
s298	1	1930	3.01	8.55	1.34	25
apex2	1	1878	3.00	5.10	1.22	25
seq	1	1750	3.02	4.38	1.23	25
bigkey	1	1699	3.21	2.27	1.24	25
des	1	1591	3.15	4.02	1.13	25
alu4	1	1522	3.01	4.49	1.21	25
diffeq	1	1494	3.28	6.59	1.11	25
misex3	1	1397	3.01	4.50	1.19	25
dsip	1	1362	3.17	2.23	1.24	25
apex4	1	1262	3.02	4.63	1.19	25
ex5p	1	1064	3.06	4.66	1.22	25
tseng	1	1046	3.49	5.94	1.23	25
Mean			3.16		1.29	

Table B.2: Results for target recovery time 2.5×10^{-4}s

Circuit	Number of Partitions	Number of BLEs (original)	Increase in BLE Number	Clock Period (original) (ns)	Clock Slowdown Factor	Number of Successful Runs
clma	14	8365	3.18	8.85	1.51	3
s38584.1	6.15	6177	3.29	5.00	1.66	26
s38417	7	6042	3.3	6.35	1.44	26
ex1010	5	4598	3.3	5.83	1.60	26
pdc	5	4575	3.19	6.54	1.45	25
spla	4	3690	3.17	6.01	1.40	25
elliptic	4	3602	3.4	7.78	1.65	25
frisc	4	3539	3.36	10.99	1.43	25
s298	2	1930	3.04	8.56	1.45	25
apex2	2	1878	3.13	5.12	1.32	25
seq	2	1750	3.18	4.42	1.37	25
bigkey	2	1699	3.21	2.29	1.38	25
des	2	1591	3.23	4.00	1.23	25
alu4	2	1522	3.11	4.61	1.31	25
diffeq	2	1494	3.36	6.57	1.19	25
misex3	2	1397	3.12	4.47	1.32	25
dsip	2	1362	3.31	2.25	1.57	25
apex4	2	1262	3.17	4.65	1.33	25
ex5p	1	1064	3.06	4.45	1.28	25
tseng	1	1046	3.49	5.96	1.23	25
Mean			3.23		1.41	

Table B.3: Results for target recovery time 1.2×10^{-4}s

Circuit	Number of Partitions	Number of BLEs (original)	Increase in BLE Number	Clock Period (original) (ns)	Clock Slowdown Factor	Number of Successful Runs
clma		Could not partition for such a small recovery time				0
s38584.1		Could not partition for such a small recovery time				0
s38417		Could not partition for such a small recovery time				0
ex1010	10.23	4598	3.31	5.84	1.57	22
pdc	12.14	4575	3.22	6.21	1.57	7
spla	8	3690	3.19	5.97	1.49	25
elliptic	10.33	3602	3.42	7.54	1.66	12
frisc		Could not partition for such a small recovery time				0
s298	5	1930	3.05	8.43	1.50	25
apex2	3	1878	3.13	5.10	1.34	25
seq	3	1750	3.21	4.44	1.40	25
bigkey	3	1699	3.21	2.29	1.58	25
des	3	1591	3.30	3.95	1.41	25
alu4	3	1522	3.14	4.53	1.32	25
diffeq	3	1494	3.38	6.57	1.44	25
misex3	3	1397	3.16	4.48	1.32	25
dsip	3	1362	3.24	2.22	1.42	25
apex4	2	1262	3.26	4.63	1.40	25
ex5p	2	1064	3.40	4.44	1.44	25
tseng	2	1046	3.57	5.97	1.49	25
Mean			3.26		1.46	

Table B.4: Results for target recovery time 7.5×10^{-5}s

Circuit	Number of Partitions	Number of BLEs (original)	Increase in BLE Number	Clock Period (original) (ns)	Clock Slowdown Factor	Number of Successful Runs
clma	14	8365	3.85	8.87	1.81	1
s38584.1	6	6177	3.67	4.97	1.61	15
s38417	7	6042	3.59	6.29	1.62	15
ex1010	5	4598	3.78	5.90	1.65	15
pdc	5	4575	3.82	6.56	1.64	15
spla	4	3690	3.71	5.89	1.60	15
elliptic	4	3602	3.61	7.73	1.33	15
frisc	4	3539	3.68	10.93	1.55	15
s298	2	1930	3.08	8.60	1.39	15
apex2	2	1878	3.39	5.09	1.43	15
seq	2	1750	3.39	4.42	1.45	15
bigkey	2	1699	3.35	2.28	1.63	15
des	2	1591	3.42	4.06	1.32	15
alu4	2	1522	3.31	4.58	1.41	15
diffeq	2	1494	3.38	6.50	1.39	15
misex3	2	1397	3.25	4.41	1.41	15
dsip	2	1362	3.20	2.26	1.76	15
apex4	2	1262	3.14	4.55	1.33	15
ex5p	1	1064	3.06	4.61	1.21	15
tseng	1	1046	3.49	5.92	1.22	15
Mean			3.46		1.49	

Table B.5: Results for target recovery time 1.2×10^{-4}s using Breadth- instead of Depth-First Traversal

References

[1] Using synplify to design in microsemi radiation-hardened FPGAs. Application Note AC139, Microsemi, May 2012.

[2] Virtex-5 FPGA configuration user guide. User Guide UG191, Xilinx, October 2012.

[3] Xilinx TMRTool product brief. `http://www.xilinx.com/publications/prod_mktg/CS11XX_TRMTool_Product_Brief_FINAL.pdf`, 2012.

[4] Alternative System Concepts, Inc. Single event upset (SEU) mitigation by virtual triple modular redundancy (TMR) in design reduces manufacturing cost and lowers power.

[5] M. P. Baze, J. C. Killens, R. A. Paup, and W. P. Snapp. SEU hardening techniques for retargetable, scalable, sub-micron digital circuits and libraries. In *21st SEE Symposium*. Manhattan Beach, CA, US, 2002.

[6] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-submicron FPGAs*. Number 497 in The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Bston, 1st edition, January 1999.

[7] Miljko Bobrek, Richard T. Woord, Christina D. Ward, Stephen M. Killough, Don Bouldin, and Michael E. Waterman. Safe FPGA design practices for instrumentation and control in nuclear plants. In *8th Annual IEEE Conference on Human Factors and Power Plants (HFPP)*, Monterey, California, August 2007.

[8] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, 43(6):2874 –2878, dec 1996.

[9] Ediz Cetin and Oliver Diessel. Guaranteed fault recovery time for FPGA-based TMR circuits employing partial reconfiguration. In *2nd International Workshop on Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environments (CHANGE)*, CHANGE, Moscone Center, San Francisco, California, June 2012. CHANGE.

[10] Bradley F. Dutton and Charles E. Stroud. Single event upset detection and correction in virtex-4 and virtex-5 FPGAs. In *ISCA International Conference on Computers and Their Applications*, June 2009.

[11] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based Heterogeneous FPGA Architectures*, chapter 2. Springer, 2012 edition, 2012.

[12] Sandi Habinc. Functional triple modular redundancy (FTMR). Technical Report FPGA-003-01, Gaisler Research, December 2002.

[13] Sandi Habinc. Suitability of reprogrammable FPGAs in space applications. Technical Report FPGA-002-01, Gaisler Research, September 2002.

[14] Jonathan M. Johnson and Michael J. Wirthlin. Voter insertion algorithms for FPGA designs using triple modular redundancy. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 249–258, New York, NY, USA, 2010. ACM.

[15] Jason Luu. A hierarchical description language and packing algorithm for heterogenous FPGAs. Master's thesis, Electrical and Computer Engineering, University of Toronto, 2010.

[16] Jason Luu, Vaughn Betz, Ted Campbell, Wei Mark Fang, Peter Jamieson, Ian Kuon, Alexander Marquardt, Andy Ye, and Jonathon Rose. *VPR User's Manual*, January 2012.

[17] Barbara Marty. Virtual field programmable gate array triple modular redundant cell design. Technical Report AFRL-VS-PSTR-TR-2004-1093, Schafer, AIR FORCE RESEARCH LABORATORY/VSSE, March 2004.

[18] A. Mishchenko, M. Case, R. Brayton, and S. Jang. Scalable and scalably-verifiable sequential synthesis. In *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 234–241, 2008.

[19] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 836–843, New York, NY, USA, 2006. ACM.

[20] OECD. The space economy at a glance 2011. Online, 2011.

[21] B. Pratt, M. Caffrey, J.F. Carroll, P. Graham, K. Morgan, and M. Wirthlin. Fine-grain SEU mitigation for FPGAs using partial TMR. *Nuclear Science, IEEE Transactions on*, 55(4):2274 –2280, August 2008.

[22] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, Fakultät für Informatik Universität Karlsruhe, D-76128 Karlsruhe, Germany, March 2000.

[23] F Sturesson. Single event effects (SEE) mechanism and effects. EPFL Short Course, June 2009. `http://space.epfl.ch/webdav/site/space/shared/industry_media/07%20SEE%20Effect%20F.Sturesson.pdf`.

[24] Berkeley University of California. *Berkeley Logic Interchange Format (BLIF)*. University of California, Berkeley, February 2005.

[25] Wallace Westfeldt. Who's using Virtex and Spartan FPGAs in Xilinx online applications? In Carlis Collins, editor, *XCell*, number 33 in XCell, page 10. Xilinx, 1999.

[26] Yu-Liang Wu and Douglas Chang. On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, ICCAD '94, pages 362–366, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.