VPR Assessment of a Novel Partitioning Algorithm

David Munro

October 2, 2012

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 literature. This thesis aims to implement a key part of a theoretical technique to protect against radiation induced Single Event Upsets (SEUs) and assess the overheads of said technique.

Acknowledgements

Thanks go to...

Contents

1 Introduction **5**
1.1 Overview . . . 5
FPGAs . . . 5
Partial Reconfiguration . . . 7
Space Based Applications . . . 7
How We Deal With FPGA Downsides . . . 8
1.2 Triple Modular Redundancy . . . 9
Triple Modular Redundancy Implementions . . . 10
Our Algorithm . . . 11
1.3 CAD Flow . . . 11
How VPR Works . . . 12
Packer . . . 13
Placer . . . 13
Router . . . 13

2 Benchmarking **15**
2.1 Overview . . . 15
Expected Results . . . 15
2.2 Architecture . . . 16
2.3 Methodology . . . 16
2.4 Results . . . 18

2.5 Discussion 19

3 Partitioning Algorithm **20**
3.1 Overview 20
3.2 Design 20
Design Choices 20
Choice of Language 21
3.3 Input file format 21
3.4 Implementation 22
3.5 Estimating restrictions 23
Results for Time and Area Estimation 24
Discussion of Time and Area Estimation 24

4 What next **26**
4.1 Progress 26
4.2 Schedule 27

5 Conclusion **28**

A Results **29**

Bibliography **33**

VPR	Versatile Place and Route
MCNC	Microelectronics Centre of North Carolina
BLE	Basic Logic Element
CLB	Configurable Logic Block
DFG	Directed Flow Graph
SEU	Single Event Upset
LUT	Look Up Table
VTR	Verilog To Routing
STL	Standard Template Library
FPGA	Field Programmable Gate Array
TMR	Triple Modular Redundancy
BLIF	Berkeley Logic Interchange Format
ASIC	Application Specific Integrated Circuit
LAB	Logic Array Block
I/O	Input/Output
ICAP	Internal Configuration Access Port
SRAM	Static RAM
mux	Multiplexer
CAD	Computer Assisted Design
MBU	Multi Bit Upset
NRE	Non Recurring Engineering

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 [17]. Field Programmable Gate Array systems (FPGAs) have many beneficial features, such as their flexibility and low Non Recurring Engineering (NRE) costs which would make them highly desired for space based applications except for their greater susceptibility to space radiation. Hardened Field Programmable Gate Arrays (FPGAs) have dropped far behind main commercial offerings prompting a search for a solution to the radiation susceptibility of FPGAs using mainstream hardware [16], one of the most popular of which is Triple Modular Redundancy (TMR). In TMR vulnerable components are triplicated allowing for errors to be detected and mitigated. This thesis is based on the work of [9] who introduce an approach to TMR, and aims to both implement a key part of their approach then assess the implementation with the aid of an open source Computer Assisted Design (CAD) toolchain for FPGAs. The remainder of this chapter provides an overview of these technologies, discusses alternative approaches to our approach, and details why we have chosen the technique we have. The following chapter introduces our approach to benchmarking circuits, and presents our initial results along with a brief discussion. The next chapter then describes our implementation and design choices made in the implementation. The chapter after that outlines our schedule and current progress, and our final chapter presents our closing remarks.

FPGAs

Field Programmable Gate Arrays (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 device which can be mass produced by manufacturers and then adapted for an individual user's needs. Their flexibility, low cost, and faster development process make them popular for a number of applications.

There are three main components to an FPGA: Input/Output (I/O) blocks—usually around the edge—

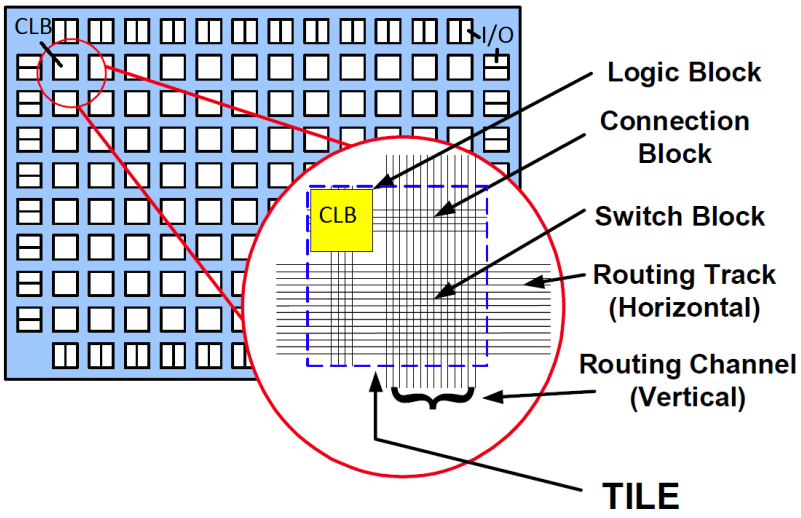

Figure 1.1: Island Style FPGA [23]

allowing for input and output from the FPGA, Configurable Logic Blocks (CLBs) containing all the logic elements or *primitives*, and the routing between everything. 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 latches and Look Up Tables (LUTs), embedding them as discrete components allows for denser designs. The routing between everything 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 buffer storing the state, and a connection can be made or unmade by writing a new value to the buffer 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 for combinational logic, a latch for register operations and allowing sequential logic, and a Multiplexer (mux) to switch between the two. As with the switches in routing, buffers store the values for the LUT, whether the mux is selecting the latch or LUT output, and other component states.

Programming an FPGA involves loading in a bitstream which describes all the component values (i.e. contents of each buffer) 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 just section of interest while the rest of the FPGA keeps running.

There are four main technologies used to implement the buffers in FPGAs: Static RAM (SRAM), which gives the highest density devices and includes the Virtex-5 family this thesis focuses on, however they are volatile and must be reprogrammed every power up from an external configuration memory; (anti)fuse, which are only one time programmable; and flash, which is non-volatile, thus not requiring an external

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

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

configuration memory, and reprogrammable, however has a lower density than SRAM based FPGAs [11].

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 to reconfigure. The configuration memory of recent Virtex devices is split into frames, and one can only reconfigure entire frames. As more frames are being reconfigured the larger the bitstream, and consequently the longer the time to reconfigure. The main configuration ports used are the external SelectMAP interface, or the internal Internal Configuration Access Port (ICAP), each with a bandwidth of 400MB/s in all Virtex devices [9, 22]

Space Based Applications

Space is quite different from a terrestrial environment, and FPGAs have a number of advantages due to their lower Non Recurring Engineering (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 intersecting with a component causes charge build up, potentially triggering incorrect operation [20]. 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 hardware still operates correctly. 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.

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 a buffer it can flip the state of that buffer changing the actual 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 will be loaded.

(Anti)fuse devices, being non reprogrammable, are immune to configuration errors, however both SRAM and flash based FPGAs are vulnerable and all three are susceptible to transient SEU [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 [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 Triple Modular Redundancy (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.

While hardened FPGAs are available, they typically lag well behind mainstream commercial offerings [16], 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.

One additional technique specific to SRAM based FPGAs relates to the protection of the off-chip configuration memory. As SRAM is volatile and loads the state from off chip at power up, this external configuration memory must also be protected from SEUs. 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].

	POWER	SPEED	HARDNESS (e/b-d)	AREA (mm^2)
Std Low Power	Rise – 0.7 μ W Fall – 0.2 μ W	Rise – 0.21 ns Fall – 0.27 ns	$10E-8$ 1 node	360
Increased IDRIVE	Rise – 1.0 μ W Fall – 0.2 μ W	Rise – 0.16 ns Fall – 0.15 ns	$2 \times 10E-9$ 1 node	460
TMR	Rise – 1.72 μ W Fall – 1.27 μ W	Rise – 0.21 ns Fall – 0.27 ns	$10E-11$ 2 node	1200
DICE	Rise - 1.4μ W Fall - 1.1μ W	Rise - 0.96 ns Fall - 0.97ns	$1.6 \times 10E-10$ 2 node	520

Table 1.2: Comparison of hardening techniques [5]

1.2 Triple Modular Redundancy

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. Any SEU will affect the output value of at most one version, so the majority vote is still correct and one can then incorporate partial reconfiguration in order to recover from detected errors. However, this 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 are negligible. Once the error has been detected and the circuit reconfigured it must then be resynchronised with the other partitions. For this thesis we will assume a method similar to that described in [9], where the resynchronising circuit is run on the same input for a number of steps equal to the number of register stages so no potentially incorrect data is left in the pipeline, however our implementation will work for any method where Equation 1.1 holds. The error recovery time consists of the time to reconfigure the circuit, which is a function of the circuit area, and the resynchronisation time, which is a function of the number of register stages and clock frequency, so it is required that our area and number of register stages are small enough, and frequency large enough,

that our error recovery time is within a user specified limit.

$$\begin{aligned}
\text{Error Recovery Time} &= \text{Error Detection Time} + \text{Reconfiguration Time} + \text{Resynchronisation Time} \\
\text{Error Detection Time} &<= \frac{1}{\text{Clock Frequency}} \times \text{Register Stages} \\
\text{Reconfiguration Time} &= \frac{1}{\text{Reconfiguration Speed}} \times \text{Bitstream Size} + Constant \\
&\propto \text{Partition Size in Frames} \\
\text{Resynchronisation Time} &<= \frac{1}{\text{Clock Frequency}} \times \text{Register Stages} \\
\therefore \text{Error Recovery Time} &<= \frac{2 \times \text{Register Stages}}{\text{Clock Frequency}} + \frac{\text{Bitstream Size}}{\text{Reconfiguration Speed}} + Constant \qquad (1.1)
\end{aligned}$$

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

Triple Modular Redundancy Implementions

This thesis builds on the work of [9] which details a partitioning algorithm that traverses a circuit represented as a Directed 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. There are a number of existing TMR solutions, however none quite meet our requirements. Our first requirement is that standard FPGA hardware can be used, with our implementation specifically targeting Virtex 5 chips. Options with custom hardware such as [16], are often prohibitively expensive, and prevent us from using our existing boards. Many FPGAs marketed specifically at space based applications are, in addition to requiring custom hardware, only latchup immune or only include inbuilt TMR on registers, leaving them still vulnerable to SEUs [12]. Non hardware solutions are typically implemented pre-synthesis, such as [3], and require existing code to be rewritten, or during synthesis such as [2] and [4] which support neither specifying an error recovery limit, nor for adding reconfiguration logic. Other options look at using partial TMR (e.g. [18]) 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 [1]. One approach similar to ours is presented by [13] who also partition a post-synthesis netlist (represented by a DFG), however they focus on cutting feedback loops rather than partitioning to utilise partial reconfiguration, although we do cut feedback loops as part of our partitioning process.

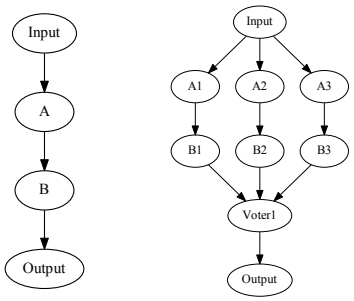

Figure 1.2: DFG before and after partitioning

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 breadth first manner, keeping track of the number of register stages, area, and maximum frequency, extending our partition area as we do so, until our recovery time constraint would be violated. At that point we triplicate our partition, insert our additional voting logic, and then repeat for a new partition, until all nodes have been partitioned. While doing so we must make sure that no loops exist within a partition and that all values are voted on before being reused, as otherwise the circuit may not resynchronise. This is accomplished by making sure that each node is only added once, and when inserting the voting logic that all outputs are voted on before being used as inputs.

1.3 CAD Flow

FPGAs are typically programmed in a higher level description language such as VHDL or Verilog, and then a number of programs—collectively making up the Computer Assisted Design (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 sub processes as illustrated in Figure 1.3 [6, 11, 15].

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.

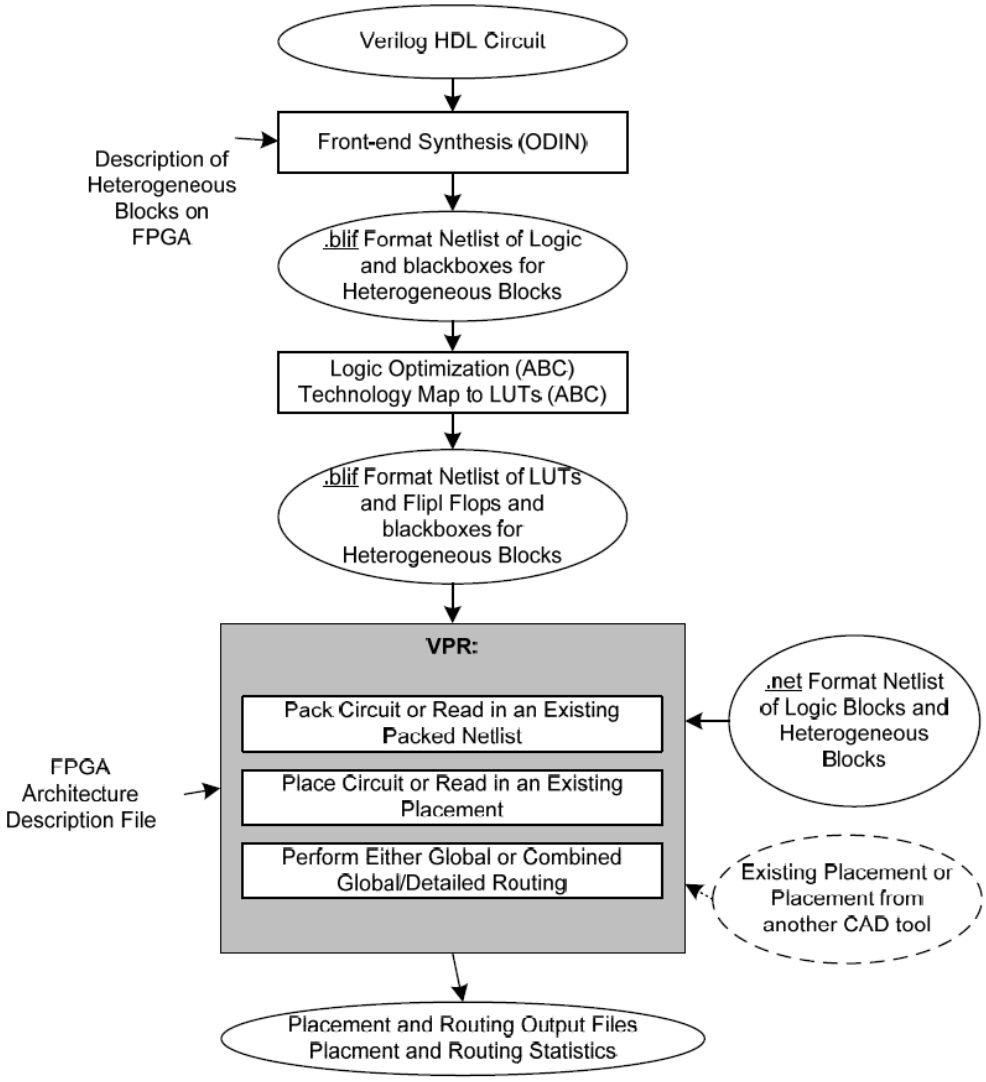

Figure 1.3: Cad Design Flow. [15]

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 in between CLBs (global routing).

How VPR Works

For this thesis we will be assessing the results of our algorithm implementation after processing by Versatile Place and Route (VPR), an open source packer, placer and router. VPR was chosen as 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. A brief understanding of the algorithms used in VPR and the effects of different settings is useful, though not critical, for understanding the results. [15] has a more detailed list of all the options VPR takes.

Packer

VPR uses the AAPack algorithm described by [14]. 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 based on a configurable cost function until no more primitives can be added. It then repeats for the next cluster, and the next after that, until every primitive is 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 [15]:

- 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. Even when optimising timing exclusively, it will still attempt to maximally pack each cluster.

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, are likely superior to custom options we may choose to use, and affect the quality of the result rather than materially affecting the behaviour [6, 15]. For these reasons we will be leaving them at their default.

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 much faster routes (2×–10×) with less CPU time; and directed_search, which like breadth_first is routability driven however uses A* to improve runtime. We will be using the default timing_driven algorithm There

are a number of options setting algorithm parameters, all of which we will leave at their defaults, however we will be changing the route_chan_width parameter as we collect results. route_chan_width specifies the width of the channels in the architecture. If omitted VPR will perform a binary search on channel capacity to determine the minimum channel width.

Chapter 2

Benchmarking

2.1 Overview

We have a number of benchmark circuits (detailed in Table 2.1 and obtained as part of the Verilog To Routing (VTR) project[1]) which we will be using to evaluate the performance of our partitioner and our TMR scheme in general. Additionally, we're looking for ways of estimating area usage and timing information from a Berkeley Logic Interchange Format (BLIF) file or DFG, without needing to actually place and route the partial circuit after each iteration, as doing so is computationally prohibitive.

To start with we made simple test circuits to compare to our benchmarks by triplicating each entire benchmark and adding in simple voter logic. As progress is made on the partitioner we can start collecting results from further partitioned circuits, however triplicating the entire circuit should be sufficient for rough approximations provided $elements_{circuit} >> elements_{voter}$.

To make the test circuits we created a small Python script to, given an input circuit and input voter circuit, triplicate the circuit and add voting logic. It creates a hierarchical BLIF file, that is, it contains nested subcircuits, which are then passed through an external program called SIS (SIS)[2] to flatten it into a format VPR can read.

Expected Results

As well described in literature ([5]) and as is intuitive, the area usage should increase by a factor of slightly more than three. There are three copies of each component, plus additional components for the voting circuitry. Maximum frequency is expected to decrease slightly due to the additional components increasing wire length and crowding routing channels, however this is likely to vary depending on circuit.

	Number of:			
Name	Inputs	Outputs	Latches	LUTs
alu4	14	8	0	4574
apex2	38	3	0	5637
apex4	9	19	0	3805
bigkey	229	197	672	5294
clma	62	82	99	25177
des	256	245	0	5018
diffeq	64	39	1131	4521
dsip	229	197	672	4283
elliptic	131	114	3366	10920
ex1010	10	10	0	13804
ex5p	8	63	0	3255
frisc	20	116	2658	10733
misex3	14	14	0	4205
pdc	16	40	0	13765
s298	4	6	24	5796
s38417	29	106	4389	18232
s38584.1	38	304	3780	18835
seq	41	35	0	5285
spla	16	46	0	11116
tseng	52	122	1155	3260

Table 2.1: Benchmark circuits used

2.2 Architecture

VPR allows us to specify a custom architecture for it to run against in an XML format. Initially we are keeping the default architecture detailed by [15] consisting of a grid of CLBs each consisting of ten fully interconnected BLEs, and each BLE having a latch and 6-LUT. Each BLE has 6 inputs and 1 output and each CLB has 33 inputs and 10 outputs.

2.3 Methodology

To start with, we wanted to collect rough estimates on the impact of partitioning a circuit to provide a baseline with which to compare our partitioning algorithm, and to develop the rough estimates needed for our partitioning algorithm. To that end we first created a simple Python script to take an arbitrary input circuit, triplicate it, and insert arbitrary voter logic. These triplicated circuits were then placed and routed by VPR, as were the original benchmarks, and the results compared.

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

[2] Available from `http://www1.cs.columbia.edu/~cs4861/sis/`

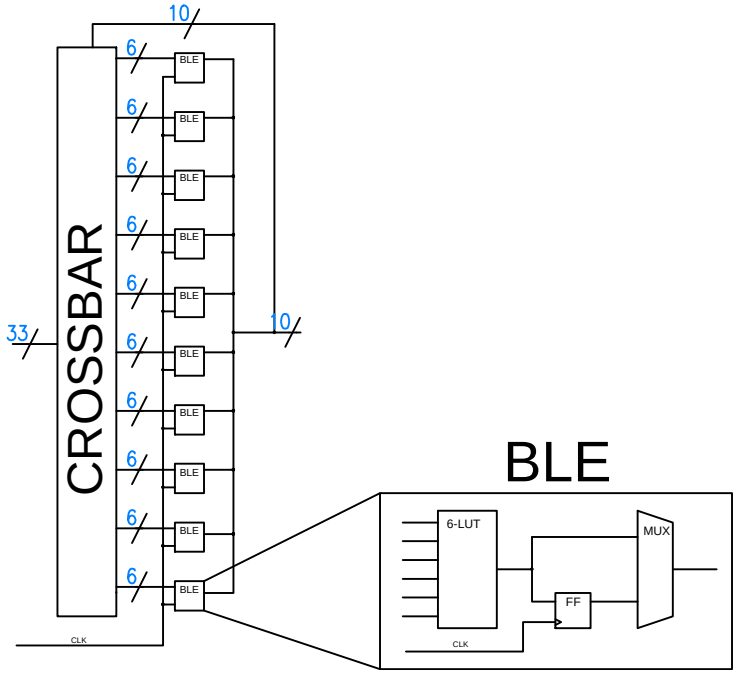

Figure 2.1: CLB Architecture

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 2.2: Architecture Elements

Width	Num. Latches	Num. LUTs	FPGA Width	Channel Width	Num. Wire Segments	Used Area	Frequency	CPU Time
Auto Width	300%	301%	169%	119%	106%	301%	93%	405%
200 Width	300%	301%	169%	N/A	110%	301%	85%	385%
60 Width	300%	301%	169%	N/A	113%	302%	86%	444%

Table 2.3: Median increase for specified channel widths

Name		Num. Latches	Num. LUTs	FPGA Width	Num. Wire Segments	Used Area	Frequency	CPU Time
pdc width	200	N/A	300%	176%	104%	301%	75%	425%
tseng width	200	300%	312%	169%	126%	308%	102%	388%

Table 2.4: Increase in circuits with maximum and minimum frequency slowdown

VPR is used with our architecture file (described in Section 2.2) and the command line options

```
1 VPR architecture.xml circuit.blif --full_stats[ --route_chan_width x]
```

where x is the width of the routing channels and - -full_stats tells VPR to be more verbose in its output. As mentioned earlier, if - -route_chan_width is excluded then VPR determines the minimum channel width needed to successfully route the circuit [15]. We then place and route our benchmark circuits and our partitioned circuits. VPR itself then reports the area usage, critical path time (inverse of the frequency), and other statistics we analysed. VPR does not unfortunately report the number of register stages, however our partitioner will as it needs to calculate the number of steps for its time estimation function.

2.4 Results

The results listed in this section highlight information of interest in a few key circuits, rather than including page after page of tables. Any aggregate statistics (e.g. median) are calculated on the entire result set, not just the results included. No collected results were considered to be outliers and the only exclusions from the median calculation are those where the operation was not able to be completed, so there is no data. These tables list scale factors, that is, $\frac{TMR}{Non-TMR}$.

2.5 Discussion

Our simple voter circuit consists of one 3-LUT per output. Therefore we expect the number of logic elements (latches and combinational logic) to be exactly three times larger, with an additional 3-LUT per original circuit output. As shown in table 2.3 our triplicated circuits are just slightly over three times as large. Circuit area should be roughly tripled as well, which again, matches, with the width increasing by $1.69 \approx \sqrt{3}$ and the used area increasing by just over triple. The partitioned circuits require slightly larger channels, in order to route the extra wires needed, and the additional elements and wires lead to slightly more segments per wire, and a slightly lower maximum frequency. Of note is that the time to place and route the partitioned circuits was much higher, taking around four times longer.

Circuits were 15% slower on average, with a worst case of 25% overall and a best case (in 200 width circuits) of a 2% speedup. For minimum channel routing the smallest median slowdown was observed as on average the minimum channel width needed for partitioned versions was higher, giving the router more flexibility. For 60 width circuits some were unable to be routed, therefore the results don't represent all circuits. For those reasons the reported statistics are taken from the 200 width circuits.

The speedup, while small, is unusual. It is likely due to the packer having more options, and due to the larger available area (as the packer will only increase the number of CLBs, and hence FPGA area used, when it can no longer fit new primitives in the current block).

Chapter 3

Partitioning Algorithm

3.1 Overview

In this section we discuss the partitioning algorithm, including how we're implementing it, progress, and the reasoning behind design choices made.

3.2 Design

See Section 1.2 We will be implementing the algorithm introduced in Section 1.2, in which we traverse a DFG in a depth first manner splitting the DFG into subgraphs which we triplicate and insert voter logic into, while ensuring the error recovery time for each subgraph is below a threshold.

Design Choices

As much as possible, we'd like our partitioner 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. We already have python scripts written to create our benchmark circuits which are able to manipulate BLIF files, so we opted for a toolchain incorporating them to reduce development time before we have a working implementation. Specifically, our partitioner operates on BLIF files, then just generates separate BLIF files for each partition, leaving our Python scripts to perform the actual triplication, insertion of additional elements, and stitching them together. Given time we would like to combine the functionality into one program, however this is a lower priority than developing a working implementation.

Other design choices include deciding on VPR due to its open nature as discussed earlier in Section **??**, and how we traverse our DFG. A depth first traversal would tend to generate long narrow pipelines within each partition, thus increasing the number of register stages, whereas a breadth first traversal would lend itself to fewer register stages for the same number of nodes. 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 were faced with a choice of 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, however 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're using a combination of languages, mainly Python and C++. Language choice primarily came down to preference regarding familiarity and personal taste, however 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 development time while still being readable and maintainable (although Python's performance on text is still quite reasonable) [19]. 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 [19]. 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).

3.3 Input file format

The BLIF file format is a textual format which describes an arbitrary sequential or combinational network of logic functions [21]. Of the full BLIF specification, VPR only supports a subset of it, and hence our partitioner is also designed to only support that same subset.

```
1  .model voter
2  .inputs in1 in2 in3
3  .outputs out1 out 2
4  .clock clock
5  .names in1 in2 in3 out1
6  11- 1
7  1-1 1
8  -11 1
9  .latch in1 out2 re clock 1
10 ...
11 commands
```

```
12 ...
13 .end
```

Listing 3.1: BLIF file layout

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

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 signals (named with Signal above) as it builds and traverses the DFG. All other element information is stored and written back out when the node is written. Likewise for *Field*s.

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

3.4 Implementation

Our implementation is still incomplete and so is both liable to change and doesn't currently match our intended design. Most notably, we partition first, then triplicate, then join into one file, rather than triplicating as we partition. A simple pseudocode description is included in 3.2 (omitting code to read and write BLIF files) and is discussed below.

Reading in the BLIF file is a relatively simple process as subcircuits aren't supported. We make one pass through the input file, first reading in the list of inputs, outputs and clocks, and then creating a node for each primitive. We then iterate through the set of nodes building a list of signals, with each signal storing its sources and sinks.

```
1 Model = new BlifModel(file)
2 Queue = new Queue()
3 for each(Signal in Model->Inputs)
4     Queue.Push(Signal->Sinks)
5 Partition = new Partition()
6
7 while(Queue.Size > 0)
```

```
8       Node = Queue.Pop()
9       if(AlreadyUsed(Node))
10          continue
11      if(EstimatedRecoveryTime > MAXIMUM_FAULT_RECOVERY_TIME)
12          WritePartitionToFile(FileName, Partition)
13          Partition = new Partition()
14      Partition.Add(Node)
15      for each(Signal in Node->Signals)
16          Queue.Push(Signal->Sinks)
17
18  CombinePartitions()
```

Listing 3.2: Simplified Pseudocode

As mentioned in Section 3.3, our input file format is a text file listing all the nodes. We read the file into memory, store it as a DFG, with all nodes and signals additionally stored in a hashmap to allow for quick random access. Each node contains a list of all connected signals, and each signal contains a list of all its sources and sinks making the DFG quite easy to traverse. Additionally we store a status for each node indicating whether it's part of the current partition, a previous partition, or new, allowing us to detect feedback loops and avoid adding nodes multiple times. We then traverse the DFG in a breadth first matter while keeping running track of an estimate of the current partition's area and timing information. Once adding a new node would exceed our constraints we write the set of contained nodes to an output BLIF file, and proceed with partitioning the rest. Eventually we have one BLIF file for each partition. We then pass these to a set of existing Python scripts, written for initial benchmarking purposes and described in section 2.3 which triplicates each partition and inserts voting logic, then connects each partition back up in a hierarchical BLIF file. This file is then passed to SIS to be flattened, at which point the partitioned circuit is ready for VPR.

3.5 Estimating restrictions

As mentioned earlier, in order to partition our circuit we need a method of calculating the partition's (including voter logic) recovery time, which is based on circuit area (affects time to reconfigure), number of register stages (affects time to resynchronise), and frequency (affects time to detect error and resynchronise). Calculating the number of register stages is accomplished while traversing the DFG. Area and timing information are more difficult to determine as they rely on the placement and routing of the circuit. Placing and routing each partial partition every step as we traverse is not computationally feasible in a reasonable amount of time, as placement and routing are relatively slow processes and one of our goals is for our partitioning stage to be approximately as fast as the other stages. Therefore, we need a way of estimating them. To do so we've collected preliminary benchmark information for a number of

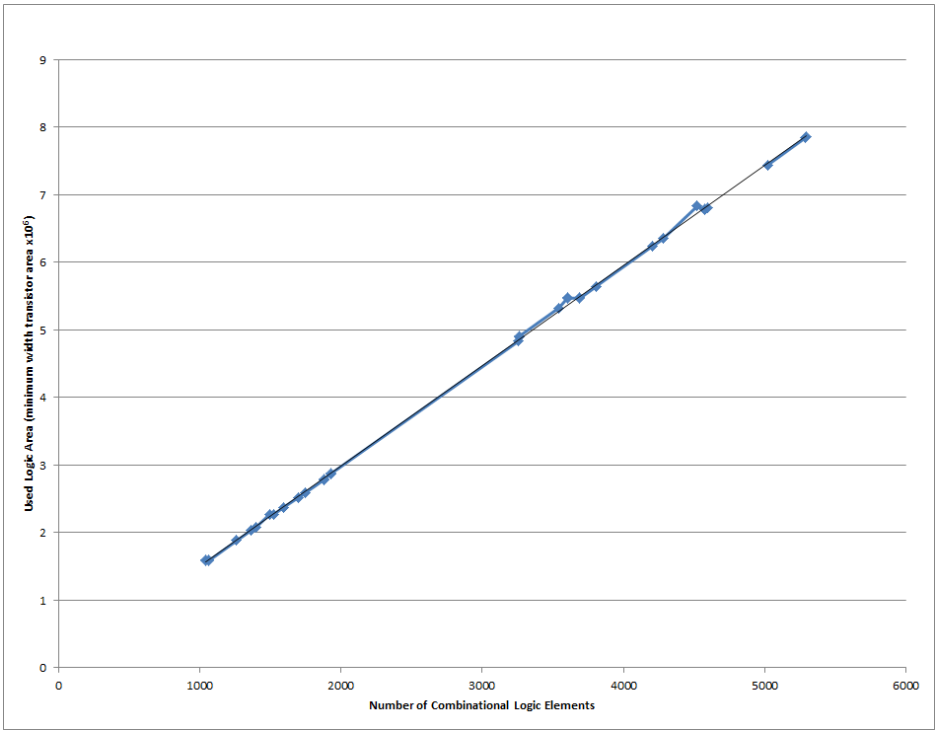

Figure 3.1: Circuit Area compared to number of Logic Elements

Name	Num. Latches	Num. LUTs	FPGA Width	Num. Wire Segments	Used Area	Frequency	CPU Time
pdc 200 width	N/A	300%	176%	104%	301%	75%	425%

Table 3.1: Percent increase in circuit with maximum path slowdown

test circuits and analysed them for patterns allowing us to accurately guess area and timing from a given circuit without placing and routing.

Results for Time and Area Estimation

Discussion of Time and Area Estimation

As is shown in Figure 4.1 the area usage is can be accurately estimated, as there is a clear relationship between the number of nodes and the area usage. The architecture we're using has one latch and one LUT per BLE, so for our supported logic elements the number of BLEs used close to linear in $max(num_{latch}, num_{names})$. VPR's packer can be either timing or area driven, currently we are using default settings (mostly area driven) giving us the linear relationship shown in Figure 4.1, however even when completely timing driven the packer still tries to fill every CLB [14].

After we have a basic partitioning algorithm an area of further investigation is the impact of changing VPR's settings on the benchmark results, and the accuracy of our estimation functions.

Timing information, on the other hand, is harder to estimate with no obvious pattern, with maximum frequency appearing independent of the number of nodes. In Section 2.4 we saw that the median slowdown was 15% with a 25% worst result. Conversely, for a few rare cases the partitioned version is actually faster. Initially we will do a rough place and (optionally) route of the original circuit to determine a base time, then multiply it by an experimentally determined slowdown factor to obtain an estimate for the frequency. Initially we're using a slowdown factor of 2 (so half speed after partitioning) which easily encompasses all test circuits we've tried. We can then modify this factor by hand to examine if the impact of it on the final partition's performance warrants improving our estimation function.

Chapter 4

What next

4.1 Progress

The initial partitioner implementation is still in progress. We anticipate a basic working version by November 2012, and then spending the next months collecting results and improving the partitioner. Done:

- Can read a BLIF file into a DFG.
- Can traverse a circuit represented as a DFG
- Have basic area and timing estimation functions.
- Can triplicate an arbitrary circuit (in a single BLIF file) and insert arbitrary voter logic (stored in another BLIF file).
- Initial benchmarks.

To Do:

- Write DFG to BLIF.
- Incorporate Python scripts into partitioning toolchain.
- Benchmark initial partitioning algorithm.
- Improve partitioner benchmarks.
- Investigate the effect of changing VPR's default parameters upon our results.
- Combine functionality of Python scripts and C++ partitioner into one program.
- Incorporate that single program into VTR's design flow, likely as part of VPR.

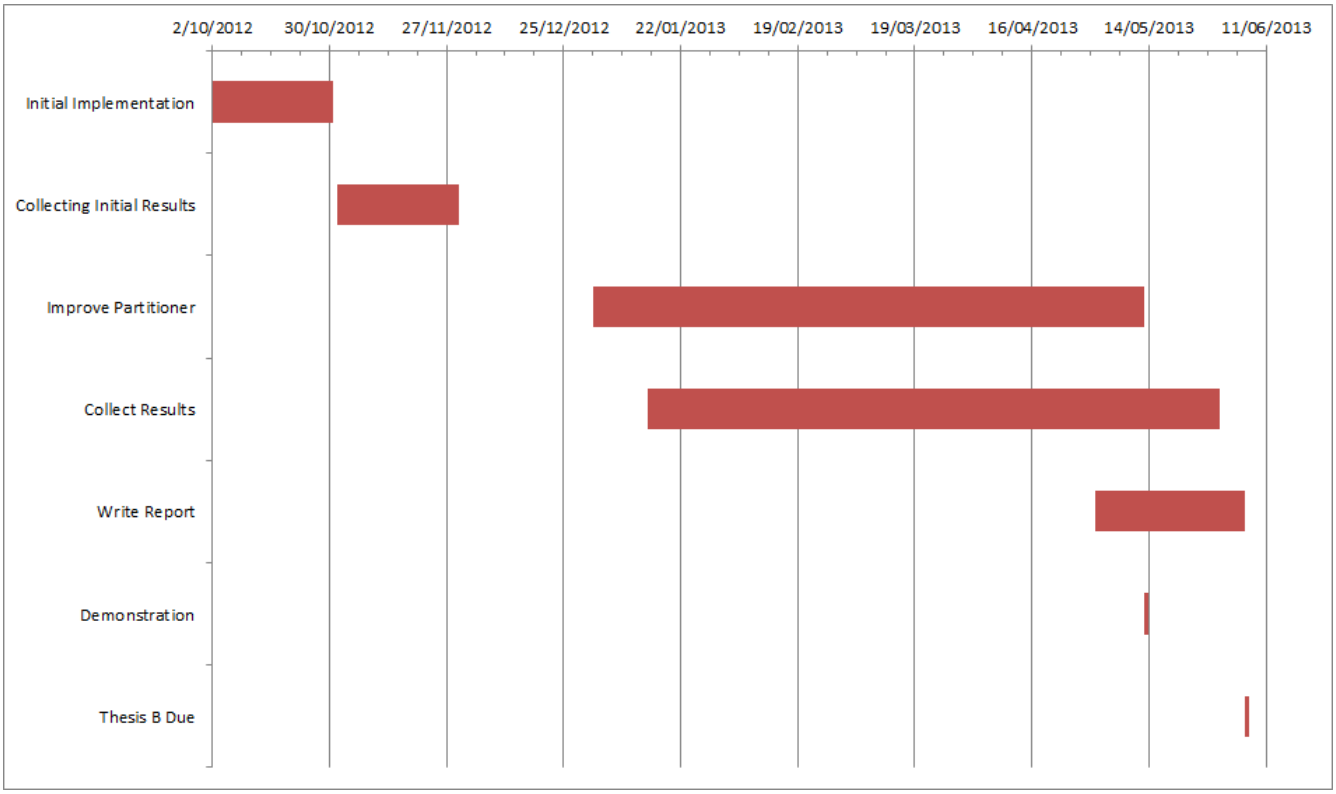

Figure 4.1: Schedule

4.2 Schedule

Chapter 5

Conclusion

We are currently on track with our initial schedule, and expect to begin collecting results on our partitioner implementation, rather than the effects of generic TMR, at the end of this year. During the Christmas holidays progress will likely be quite limited, however as we are very close to a working implementation it is believed to be achievable. Our results collected so far reflect only TMR in general, rather than our specific implementation, and they match those of literature and our expectations. We expect slightly worse performance from our partitioner due to additional logic being inserted, but not significantly so.

Appendix A

Results

Name	Num. Latches	Num. LUTs	FPGA Width	Channel Width	Av. No. Wire Segments	Maximum Frequency (MHz)	CPU Time (s)
alu4	0	1522	15	200	4.76391	225	34.566
alu4TMR	0	4574	25	200	5.29148	188	132.41
apex2	0	1878	16	200	5.5124	216	47.45
apex2TMR	0	5637	29	200	6.21459	172	197.825
apex4	0	1262	14	200	5.0436	249	31.272
apex4TMR	0	3805	23	200	5.33318	204	119.002
bigkey	224	1699	16	200	3.70506	427	56.61
bigkeyTMR	672	5294	27	200	4.93838	377	193.388
clma	33	8365	34	200	5.62286	115	379.801
clmaTMR	99	25177	59	200	6.2762	103	2146.4
des	0	1591	16	200	4.07107	262	98.709
desTMR	0	5018	27	200	3.9862	229	263.883
diffeq	377	1494	15	200	3.0611	157	60.103
diffeqTMR	1131	4521	25	200	3.44041	156	204.691
dsip	224	1362	14	200	3.823	433	60.372
dsipTMR	672	4283	24	200	4.74885	377	177.405
elliptic	1122	3602	23	200	4.2743	134	123.967
ellipticTMR	3366	10920	39	200	5.15534	130	513.637
ex1010	0	4598	25	200	5.31938	176	146.81
ex1010TMR	0	13804	44	200	5.4995	134	655.814
ex5p	0	1064	13	200	4.95796	240	30.659
ex5pTMR	0	3255	22	200	5.2736	187	112.023
frisc	886	3539	23	200	5.52549	94.3	144.196
friscTMR	2658	10733	39	200	5.82234	90.1	525.145
misex3	0	1397	14	200	4.93	244	35.739
misex3TMR	0	4205	24	200	5.46202	198	136.961
pdc	0	4575	25	200	7.21792	175	167.232
pdcTMR	0	13765	44	200	7.51339	131	711.155
s298	8	1930	17	200	4.76794	124	47.789
s298TMR	24	5796	29	200	4.83777	116	185.169
s38417	1463	6042	30	200	3.53974	158	241.471
s38417TMR	4389	18232	51	200	3.78557	149	942.7
s38584.1	1260	6177	30	200	3.67662	206	263.313
s38584.1TMR	3780	18835	52	200	4.36338	157	1199.39
seq	0	1750	16	200	5.31333	253	50.624
seqTMR	0	5285	27	200	6.23725	193	184.467
spla	0	3690	23	200	6.44786	190	122.311
splaTMR	0	11116	39	200	6.75377	152	496.416
tseng	385	1046	13	200	3.04212	161	30.561
tsengTMR	1155	3260	22	200	3.82465	164	118.583

Table A.1: Results for 200 width channels

Name	Num. Latches	Num. LUTs	FPGA Width	Channel Width	Av. No. Wire Segments	Maximum Frequency (MHz)	CPU Time (s)
alu4	0	1522	15	60	4.79098	241	22.625
alu4TMR	0	4574	25	60	5.38017	193	101.828
apex2	0	1878	16	60	5.53926	212	34.256
apex2TMR	0	5637	29	60	6.34397	164	153.485
apex4	0	1262	14	60	5.13079	241	20.928
apex4TMR	0	3805	23	60	5.66967	198	90.166
bigkey	224	1699	16	60	3.79333	421	43.024
bigkeyTMR	672	5294	27	60	5.12677	363	154.662
clma				Could Not Route			
clmaTMR				Could Not Route			
des	0	1591	16	60	4.15635	257	50.68
desTMR	0	5018	27	60	4.07778	230	141.366
diffeq	377	1494	15	60	3.20978	151	28.392
diffeqTMR	1131	4521	25	60	3.60614	148	115.085
dsip	224	1362	14	60	3.84115	434	36.32
dsipTMR	672	4283	24	60	4.87989	368	118.515
elliptic	1122	3602	23	60	4.76756	137	101.398
ellipticTMR	3366	10920	39	60	5.74891	133	475.005
ex1010				Could Not Route			
ex1010TMR				Could Not Route			
ex5p	0	1064	13	60	5.29429	241	18.899
ex5pTMR				Could Not Route			
frisc	886	3539	23	60	6.17131	94.7	107.33
friscTMR				Could Not Route			
misex3	0	1397	14	60	4.99571	241	21.286
misex3TMR	0	4205	24	60	5.64103	197	91.433
pdc				Could Not Route			
pdcTMR				Could Not Route			
s298	8	1930	17	60	4.62137	122	27.868
s298TMR	24	5796	29	60	4.75444	118	120.257
s38417	1463	6042	30	60	3.66998	160	172.413
s38417TMR	4389	18232	51	60	3.87309	153	758.585
s38584.1	1260	6177	30	60	3.83166	202	195.832
s38584.1TMR	3780	18835	52	60	4.67375	159	944.364
seq	0	1750	16	60	5.30778	245	32.426
seqTMR	0	5285	27	60	6.56298	188	149.187
spla	0	3690	23	60	6.46323	188	86.03
splaTM				Could Not Route			
tseng	385	1046	13	60	3.22621	161	20.384
tsengTMR	1155	3260	22	60	3.93092	164	79.889

Table A.2: Results for 60 width channels

Name	Num. Latches	Num. LUTs	FPGA Width	Channel Width	Av. No. Wire Segments	Maximum Frequency (MHz)	CPU Time (s)
alu4	0	1522	15	34	5.61203	196	88.236
alu4TMR	0	4574	25	48	5.6293	182	245.343
apex2	0	1878	16	48	5.78099	204	80.968
apex2 TMR	0	5637	29	60	6.3913	167	885.343
apex4	0	1262	14	48	5.74387	175	55.509
apex4 TMR	0	3805	23	58	5.59716	181	357.909
bigkey	224	1699	16	42	4.00538	422	84.037
bigkey TMR	672	5294	27	48	5.46703	351	314.02
clma	33	8365	34	64	5.9249	111	826.48
clma TMR	99	25177	59	74	6.81963	94.3	6225.16
des	0	1591	16	46	4.46904	248	85.209
des TMR	0	5018	27	44	4.43131	229	329.551
diffeq	377	1494	15	38	3.67617	152	53.137
diffeq TMR	1131	4521	25	44	3.94535	156	238.817
dsip	224	1362	14	40	4.14675	446	86.123
dsip TMR	672	4283	24	38	5.10172	377	207.014
elliptic	1122	3602	23	50	5.19297	133	289.63
elliptic TMR	3366	10920	39	60	5.74707	133	1562.75
ex1010	0	4598	25	64	5.33384	172	401.434
ex1010 TMR	0	13804	44	62	5.70191	131	1399.9
ex5p	0	1064	13	50	5.64865	229	53.45
ex5p TMR	0	3255	22	62	6.03046	174	452.114
frisc	886	3539	23	56	6.66421	96.2	429.658
frisc TMR	2658	10733	39	66	6.5103	90.1	1383.44
misex3	0	1397	14	42	5.74857	233	60.019
misex3 TMR	0	4205	24	50	5.86212	190	368.648
pdc	0	4575	25	64	7.8308	156	397.958
pdc TMR	0	13765	44	72	7.74884	121	1005.6
s298	8	1930	17	30	5.16031	117	2620.11
s298 TMR	24	5796	29	34	4.97929	119	425.303
s38417	1463	6042	30	42	4.02152	158	376.671
s38417 TMR	4389	18232	51	50	3.96346	147	7381.12
s38584.1	1260	6177	30	44	4.22808	200	459.989
s38584.1 TMR	3780	18835	52	56	4.69062	156	2790.9
seq	0	1750	16	46	5.86111	207	105.526
seq TMR	0	5285	27	58	6.68624	190	448.058
spla	0	3690	23	54	7.08891	144	357.396
spla TMR	0	11116	39	66	6.92343	145	1320.57
tseng	385	1046	13	34	3.63807	163	49.618
tseng TMR	1155	3260	22	42	4.36026	163	183.167

Table A.3: Results for autosized width channels

Bibliography

[1] Single event upset (seu) mitigation by virtual triple modular redundancy (tmr) in design reduces manufacturing cost and lowers power. Technical report, Alternative System Concepts, Inc.

[2] Using synplify to design in microsemi radiation-hardened fpgas. http://www.actel.com/documents/SynplifyRH_AN.pdf.

[3] Functional triple modular redundancy. Technical report, Gaisler Research, December 2002.

[4] Xilinx tmrtool product brief. Technical report, Xilinx, 2012.

[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. *Nuclear Science, IEEE Transactions on*, 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. ISCA International Conference on Computers and Their Applications, 2009.

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

[12] Sandi Habinc. Suitability of reprogrammable fpgas in space applications. Technical Report 0.4, Gaisler Research, September 2002.

[13] 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.

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

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

[16] Barbara Marty. Virtual field programmable gate array triple modular redundant cell design. Technical report, Schafer, AIR FORCE RESEARCH LABORATORY/VSSE, March 2004.

[17] OECD. http://www.oecd.org/futures/48301203.pdf. Online, 2011.

[18] 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.

[19] 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.

[20] F. Sturesson. Single event effects (SEE) mechanism and effects. June 2009.

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

[22] 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.

[23] Steve Wilton. Fpga architectures. Course Lecture Notes.