VPR Assessment of a Novel Partitioning Algorithm

David Munro

October 1, 2012

Abstract

ro:FPGA

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.

cro:SEU

Acknowledgements

Thanks go to...

Contents

Chapter 1

Introduction

1.1 Overview

Space plays an increasingly important role in the functioning of modern societies, being vital for fields
ro:FPGA including navigation, meteorology, and communications [OECD, 2011]. Field Programmable Gate Ar-
cro:NRE rays (FPGAs) have many beneficial features, such as their flexibility and low Non Recurring Engineer-
ing (NRE) costs which would make them highly desired for space based applications except for their
greater susceptibility to space radiation. Hardened FPGAs have dropped far behind main commercial
offerings prompting a search for a solution to the radiation susceptibility of FPGAs using mainstream
cro:TMR hardware [Marty, 2004], one of the most popular of which is Triple Modular Redundancy (TMR), in
which vulnerable components are triplicated allowing for errors to be detected and mitigated. This the-
sis is based on the work of [Cetin and Diessel, 2012] 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
cro:CAD 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

ro:FPGA

Field Programmable Gate Arrays (FPGAs) are popular devices capable of implementing a wide variety of
ro:ASIC 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.

!

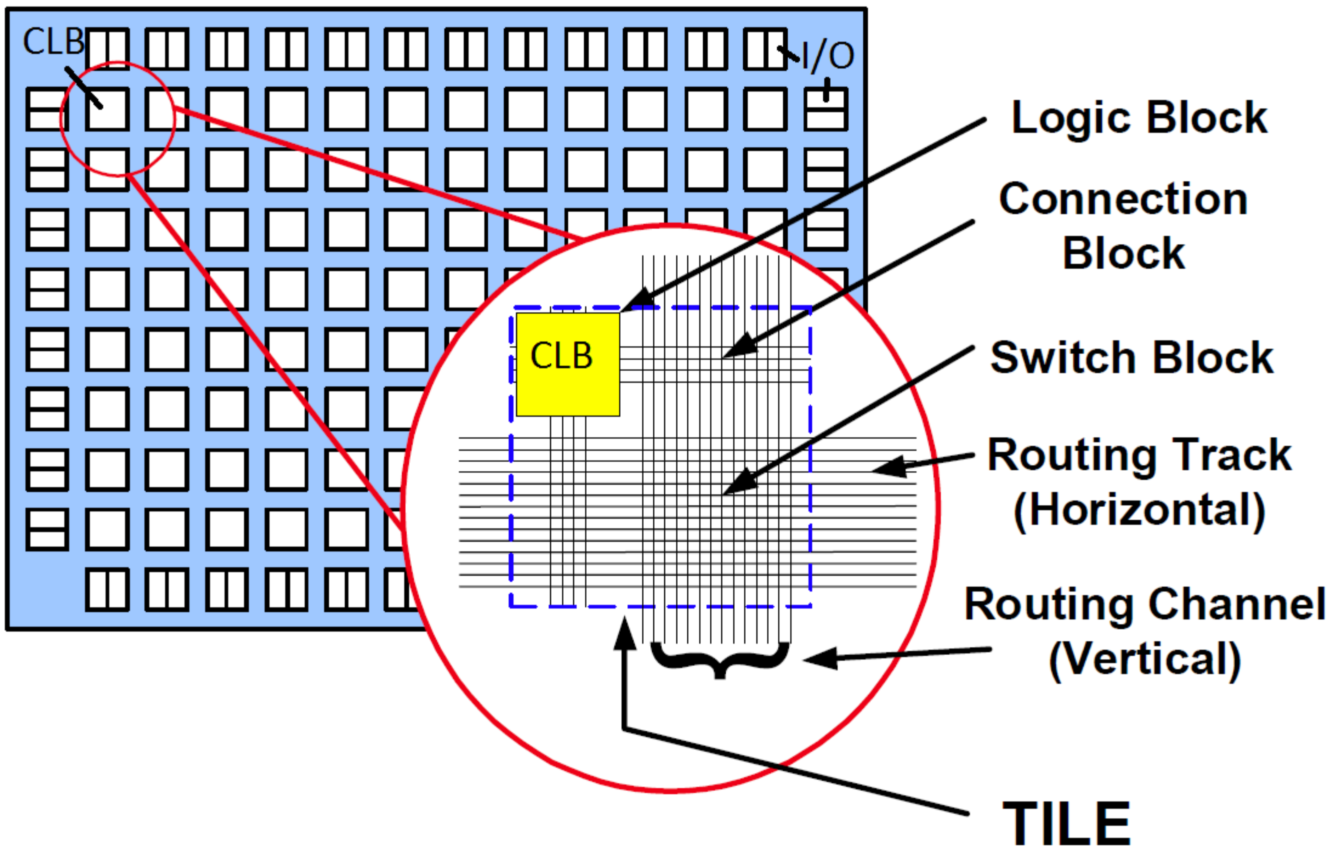


Figure 1.1: Island Style FPGA [Wilton,]

PGAArch

TODO: Use own image. Wilton lecture notes have no license/copyright notice/etc attached, so don't know if this usage is actually allowed. Wouldn't come under Fair Use

acro:IO
cro:CLB
cro:LUT
cro:BLE
cro:mux

There are three main components to an FPGA: Input/Output (I/O) blocks—usually around the edge—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. Programmable switches connect 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 [Farooq et al., 2012]. 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 bitsream 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, which involves loading the bitstream for just section of interest while the rest of the FPGA keeps running.

! Check grammar in this next section. There are four main technologies used to implement the buffers ro:SRAM 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 :EEPROM an external configuration memory; (anti)fuse, which are only one time programmable; and **EEPROM!** (**EEPROM!**), which is reprogrammable and non-volatile thus not requiring an external configuration memory, however has a lower density than SRAM based FPGAs [Farooq et al., 2012].

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 ro:ICAP the internal Internal Configuration Access Port (ICAP), each with a bandwidth of 400MB/s in all Virtex devices [Westfeldt, 1999, Cetin and Diessel, 2012]

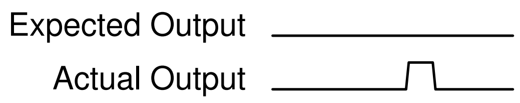
Space Based Applications

Space is quite different from a terrestrial environment, and FPGAs have a number of advantages due to cro:NRE their lower Non Recurring Engineering (NRE) costs and flexibility. As FPGAs can be reconfigured during a mission, faulty/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 [Sturesson, 2009]. As outlined in Table SEURate 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, cro:SEU 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. This cro:MBU work does not look at detecting Multi Bit Upsets (MBUs) in which multiple components are affected at the same time.

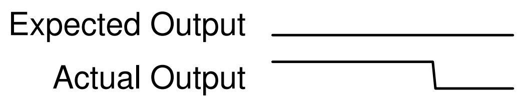
Orbit	SEUs per device/day	Mean time to upset (s)
LEO (560 km)	4.09	2.11×10^4
Polar (833 km)	1.49×10^4	5.81
GPS (20,200 km)	5.46×10^4	1.58
Geosynchronous (36,000 km)	6.20×10^4	1.39

Table 1.1: SEU Rate Predictions for Virtex-4 devices at various orbits [Cetin and Diessel, 2012]

SEURate



(a) ASIC



ICVFPGA

(b) FPGA

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 **EEPROM!** based FPGAs are vulnerable [Bobrek et al., 2007].

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 [Baze et al., 2002]:

- 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 techniques 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 [Baze et al., 2002]. These techniques often slow down operation and are ineffective against

	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 [Baze et al., 2002]

configuration errors.

- Spatial Redundancy, which uses multiple redundant circuits to detect errors and be able to continue operating. Spatial redundancy techniques are split into those which provide hardware level redundancy such as Dual Interlock Storage Cell (DICE) [Calin et al., 1996], with slightly increased area usage, increased power, and often increased component time, and those which provide design level redundancy such as Triple Modular Redundancy (TMR), which increase area and power usage significantly (see Table 1.2) however require no custom hardware and greatly improve the circuit's hardness. As the area is increased a common side effect is a decrease in maximum operating frequency, however unlike some temporal filtering options the actual component speed remains the same.

While hardened FPGAs are available, they typically lag well behind mainstream commercial offerings [Marty, 2004], 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 [Farooq et al., 2012] 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, 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 [Dutton and Stroud, 2009].

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. 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. 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 [Cetin and Diessel, 2012], where the resynchronising circuit is run on the same input for a number of steps equal to the critical path length so no potentially incorrect data is left in the pipeline, however our implementation will work for any method where Equation ERT 1.2 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 critical path and clock frequency, so it is required that our area and critical path length are small enough, and frequency large enough, that our error recovery time is within a user specified limit.

$$\begin{aligned}
\text{Error Recovery Time} &= \text{Error Detection Time} + \text{Reconfiguration Time} + \text{Resynchronisation Time} \\
\text{Error Detection Time} &<= \frac{1}{\text{Clock Frequency}} \times \text{Pipeline Steps} \\
\text{Reconfiguration Time} &= \frac{1}{\text{Reconfiguration Speed}} \times \text{Bitstream Size} + Constant \\
&\propto \text{Partition Size in Slices} \\
\text{Resynchronisation Time} &<= \frac{1}{\text{Clock Frequency}} \times \text{Pipeline Steps} \qquad (1.1) \\
\therefore \text{Error Recovery Time} &<= \frac{2 \times \text{Pipeline Steps}}{\text{Clock Frequency}} + \frac{\text{Bitstream Size}}{\text{Reconfiguration Speed}} + Constant \qquad (1.2)
\end{aligned}$$

ERT

[Cetin and Diessel, 2012] 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

cro:DFG

This thesis builds on the work of [Cetin and Diessel, 2012] 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

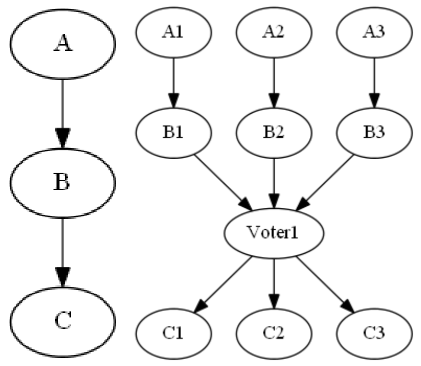


Figure 1.2: DFG before and after partitioning

RFigure

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 [Marty, 2004], 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 latchupexplain? immune or only include inbuilt TMR on registers, leaving them still vulnerable to SEUs [Habinc, 2002]. Non hardware solutions are typically implemented pre-synthesis, such as [ftm, 2002], and require existing code to be rewritten, or during synthesis such as [syn,] and [tmr, 2012] which support neither specifying an error recovery limit, nor for adding reconfiguration logic. Other options look at using partial TMR (e.g. [Pratt et al., 2008]) 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 [VTM,]. !

Our Algorithm

gorithm

Given a netlist description of a circuit, it is possible to represent the circuit as a DFG [Farooq et al., 2012]. 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 critical path length, 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 (loop $\implies$ pipeline length $= \infty$). 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 cro:CAD 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 is typically split into a number of sub processes as illustrated in Figure CADFlow 1.3 [Betz et al., 1999, Luu et al., 2012, Farooq et al., 2012].

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 in between CLBs (global routing).

How VPR Works

cro:VPR 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's open source allowing modifications to be made if necessary, and it's 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. [Luu et al., 2012] has a more detailed list of all the options VPR takes.

Packer

VPR uses the AAPack algorithm described by [Luu, 2010]. 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

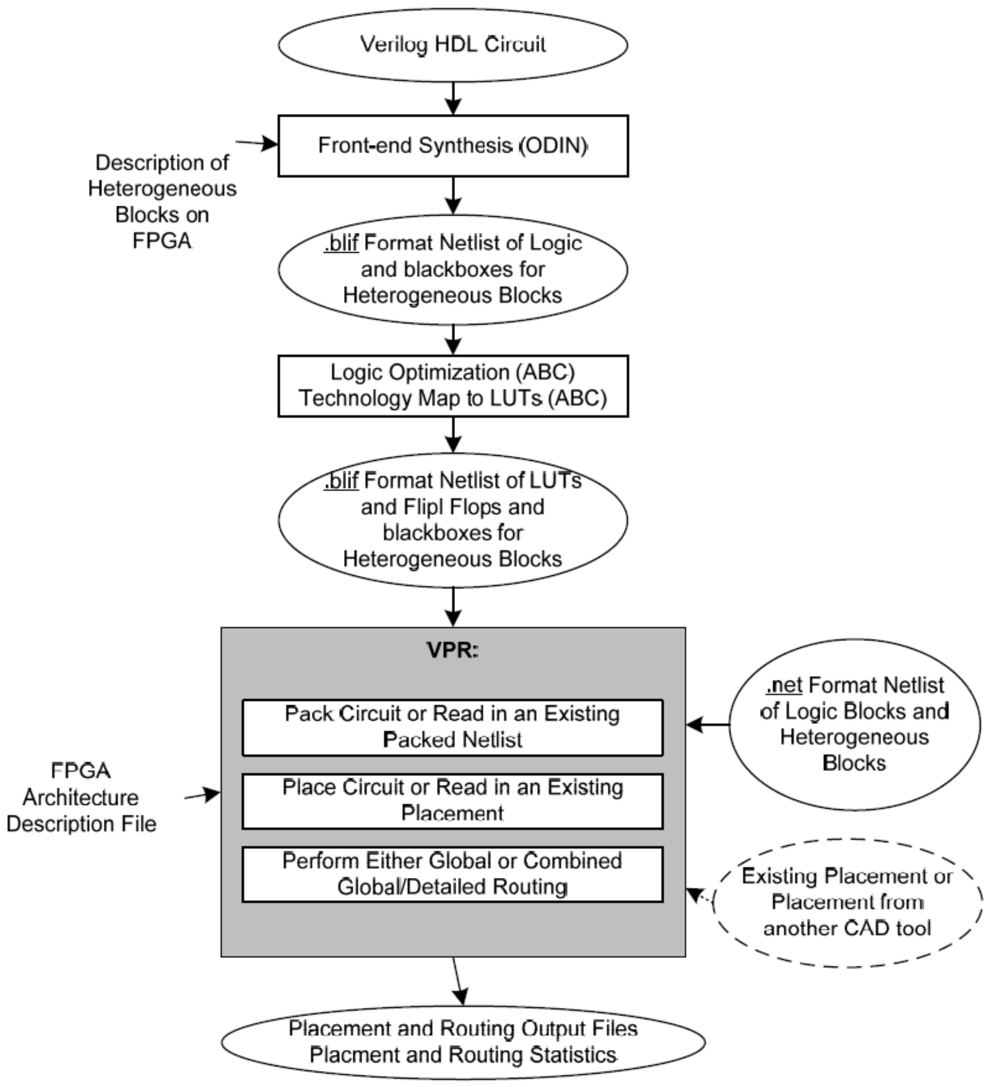

Verilog HDL Circuit
Front-end Synthesis (ODIN)
Description of Heterogeneous Blocks on FPGA
.blif Format Netlist of Logic and blackboxes for Heterogeneous Blocks
Logic Optimization (ABC) Technology Map to LUTs (ABC)
.blif Format Netlist of LUTs and Flipl Flops and blackboxes for Heterogeneous Blocks
VPR:
Pack Circuit or Read in an Existing Packed Netlist
Place Circuit or Read in an Existing Placement
Perform Either Global or Combined Global/Detailed Routing
FPGA Architecture Description File
.net Format Netlist of Logic Blocks and Heterogeneous Blocks
Existing Placement or Placement from another CAD tool
Placement and Routing Output Files Placment and Routing Statistics

Figure 1.3: Cad Design Flow. [Luu et al., 2012]

CADFlow

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 [Luu et al., 2012]:

- 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 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 [Luu et al., 2012, Betz et al., 1999]. 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 TODO: Reason? 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

cro:VTR We have a number of benchmark circuits (detailed in benchmarkList 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 ro:BLIF 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 cro:SIS nested subcircuits, which are then passed through an external program called **SIS!** (**SIS!**)[2] to flatten it into a format VPR can read.

benchmarkList

Expected Results

As well described in literature ([Baze et al., 2002]) 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. Power usage is also expected to increase by approximately a factor of three, as there are three times as many components in use concurrently. Critical path length is expected to decrease slightly due to the additional components increasing wire length and crowding routing channels, however

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

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

	Number of:			
Name	Inputs	Outputs	Latches	Combinational Logic Elements
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

this is likely to vary depending on circuit. In some cases inserting voter logic could split up a long pipeline into multiple pipeline steps, actually decreasing critical path latency.

2.2 Architecture

rchFile

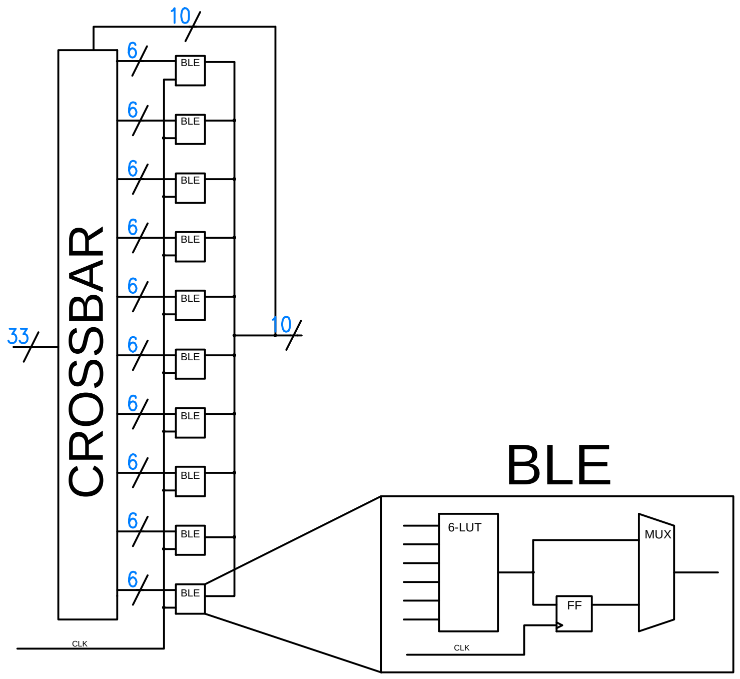
VPR allows us to specify a custom architecture for it to run against in an XML format. Initially we are keeping the default architecture detailed by [Luu et al., 2012] consisting of a grid of CLBs each consisting of ten fully interconnected BLEs, and each BLE having a latch and 6-LUT. Each BLE has 6 inputs and 1 output and each CLB has 33 inputs and 10 outputs. Routing resources consist of a configurable number of unidirectional length 4 tracks with **Wilton!** (**Wilton!**) (see Figure WiltonSwitch 2.1b) switches.

:Wilton

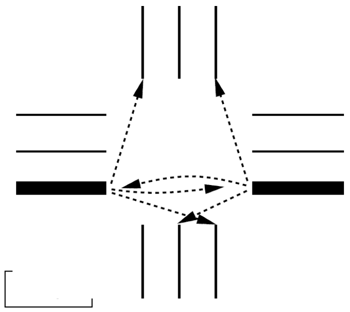
2.3 Methodology

kMethod

To start with, we wanted to collect rough estimates on the impact of partitioning a circuit to allow us to evaluate whether it's worthwhile proceeding, 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,



(a) CLB Architecture


(b) Wilton Switch Block [Masud and Wilton, 1999]

WiltonSwit

Component	Number	Delay (s)	Notes
Flip Flop	1 per BLE	3.221e-11	Shown as FF on Diagram
6-LUT	1 per BLE	2.690e-10	
MUX	1 per BLE	0	
BLE	10 per CLB	0	
Crossbar	1 per CLB	8.044e-11[a]	
CLB	Autosized by VPR	0	

[a] Maximum time.

Arch

Table 2.2: Architecture Elements

Name	Input	Output	.latch	.names	FPGA Width	Channel Width	Av. Wire Segments	Used Area	Critical Path	VPR Time
Auto Width	1	1	3	3.01	1.69	1.19	1.06	3.01	1.08	4.05
200 Width	1	1	3	3.01	1.69	1	1.10	3.01	1.17	3.85
60 Width	1	1	3	3.01	1.69	1	1.13	3.02	1.16	4.44

Table 2.3: Median Scale Factors for specified channel widths

dianRes

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.

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

```
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 [Luu et al., 2012]. We then place and route our benchmark circuits and our partitioned circuits. VPR itself then reports the area usage, critical path time, and other statistics we analysed. VPR does not unfortunately report the number of pipeline steps, however our partitioner will as it needs to calculate the number of steps for its time estimation function.

The place and route process has a random factor to it due to the algorithms used, however they should still be quite close to an optimal solution every run, a hypothesis borne out through packing, placing and routing the same circuit multiple times and observing no significant variation in final circuit quality between repetitions.Fix this up after talking to Diessel

!

2.4 Results

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 or excluded. These tables list scale factors, that is, $\frac{TMR}{Non-TMR}$.

Name	Input	Output	.latch	.names	FPGA Area (width in CLBs)	Av. Wire Segments	Used Logic Block Area	Critical Path	VPR Time
pdc 200 width	1	1	0	3.00	1.76	1.04	3.01	1.33	4.25
tseng 200 width	1	1	3	3.12	1.69	1.26	3.08	0.98	3.88

Table 2.4: Scale factors for circuit with maximum critical path slowdown

timing

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 medianRes 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 long critical path. Of note is that the time to place and route the partitioned circuits was much higher, taking around four times longer.

Circuits were 17% slower on average, with a worst case of 33% 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 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 Algorithm 1.2Recap?

!

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 VPRSection **??**, and how we traverse our DFG. A depth first traversal would tend to generate long narrow pipelines within each partition, thus increasing critical path length, whereas a breadth first traversal would lend itself to shorter critical paths 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) [Prechelt, 2000]. 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 [Prechelt, 2000]. 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 (e.g. Standard Template Library (STL)) making our implementation much easier).

cro:STL

3.3 Input file format

Section

The BLIF file format is a textual format which describes an arbitrary sequential or combinational network of logic functions [University of California, 2005]. 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.

pleBlif

```
.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 {Signal}	The model inputs.
Output List:	.outputs {Signal}	The model outputs.
Clock List:	.clock {Signal}	The model clocks.

	Commands
Combinational Logic Element:	.names {InputSignals} ⟨OutputSignal⟩
	{Line}
Latch:	.latch ⟨InputSignal⟩ ⟨OutputSignal⟩ [Field ClockSignal] [Field]
Optional End Marker:	.end

{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 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 Pseudocode 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.

udocode

```
Model = new BlifModel(file)
Queue = new Queue()
for each(Signal in Model->Inputs)
    Queue.Push(Signal->Sinks)
Partition = new Partition()

while(Queue.Size > 0)
    Node = Queue.Pop()
    if(AlreadyUsed(Node))
        continue
    if(EstimatedRecoveryTime > MAXIMUM_FAULT_RECOVERY_TIME)
        WritePartitionToFile(FileName, Partition)
        Partition = new Partition()
    Partition.Add(Node)
    for each(Signal in Node->Signals)
        Queue.Push(Signal->Sinks)

CombinePartitions()
```

Listing 3.2: Simplified Pseudocode

As mentioned in Section BLIFSection 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 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 BenchmarkMethod 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!**Explain again? 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), critical path length (affects time to resynchronise), and frequency (affects time to detect error and resynchronise). Calculating critical path length is relatively easy for our supported input format, as it's just the number of latches on the critical path, which is easily calculated while we traverse the DFG. Area and timing information are more difficult 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 test circuits and

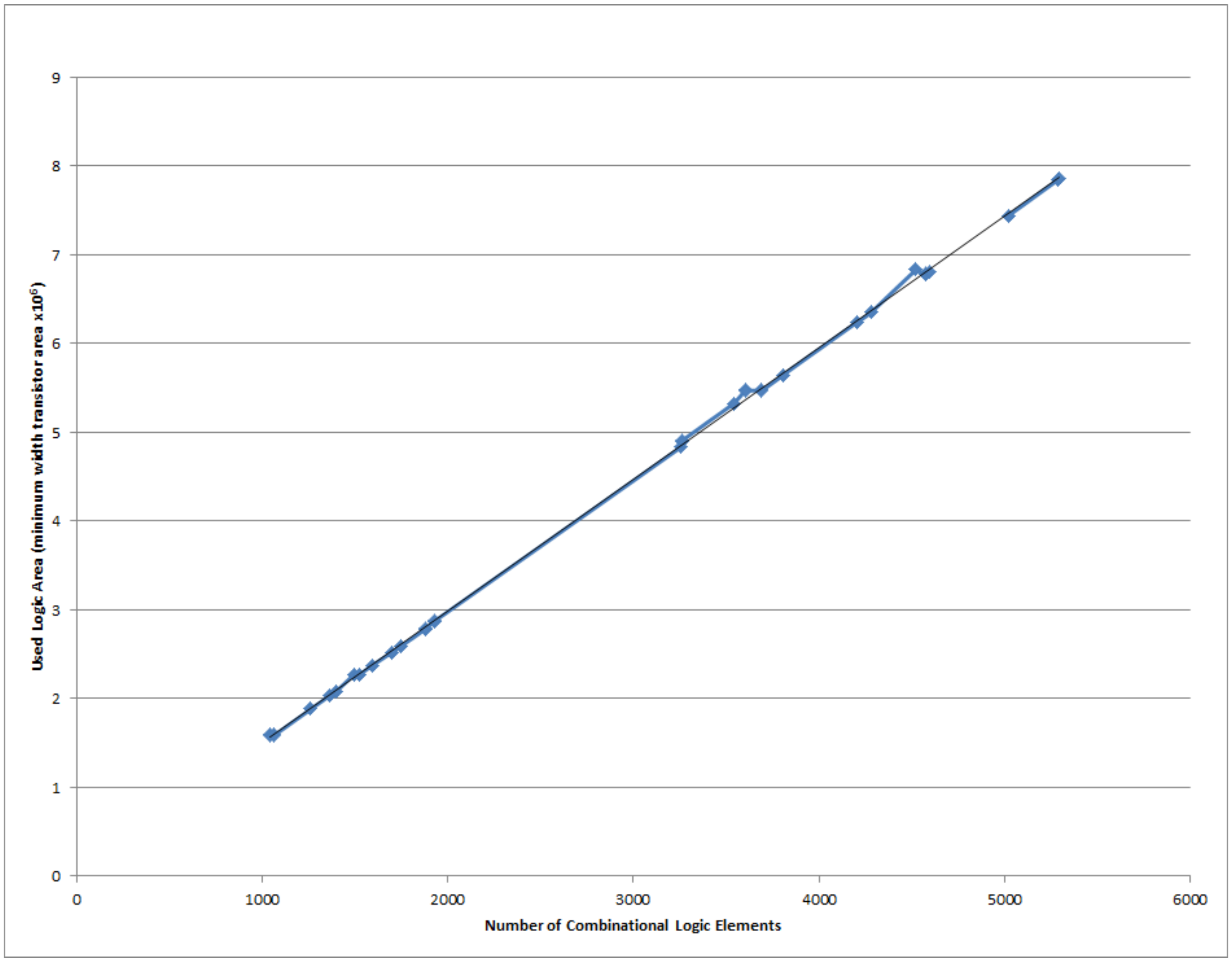


Figure 3.1: Circuit Area compared to number of Logic Elements

lements

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

!

TODO: All the graphs and tables of our estimation stuff

Discussion of Time and Area Estimation

AreaVElements

As is shown in Figure 3.4 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

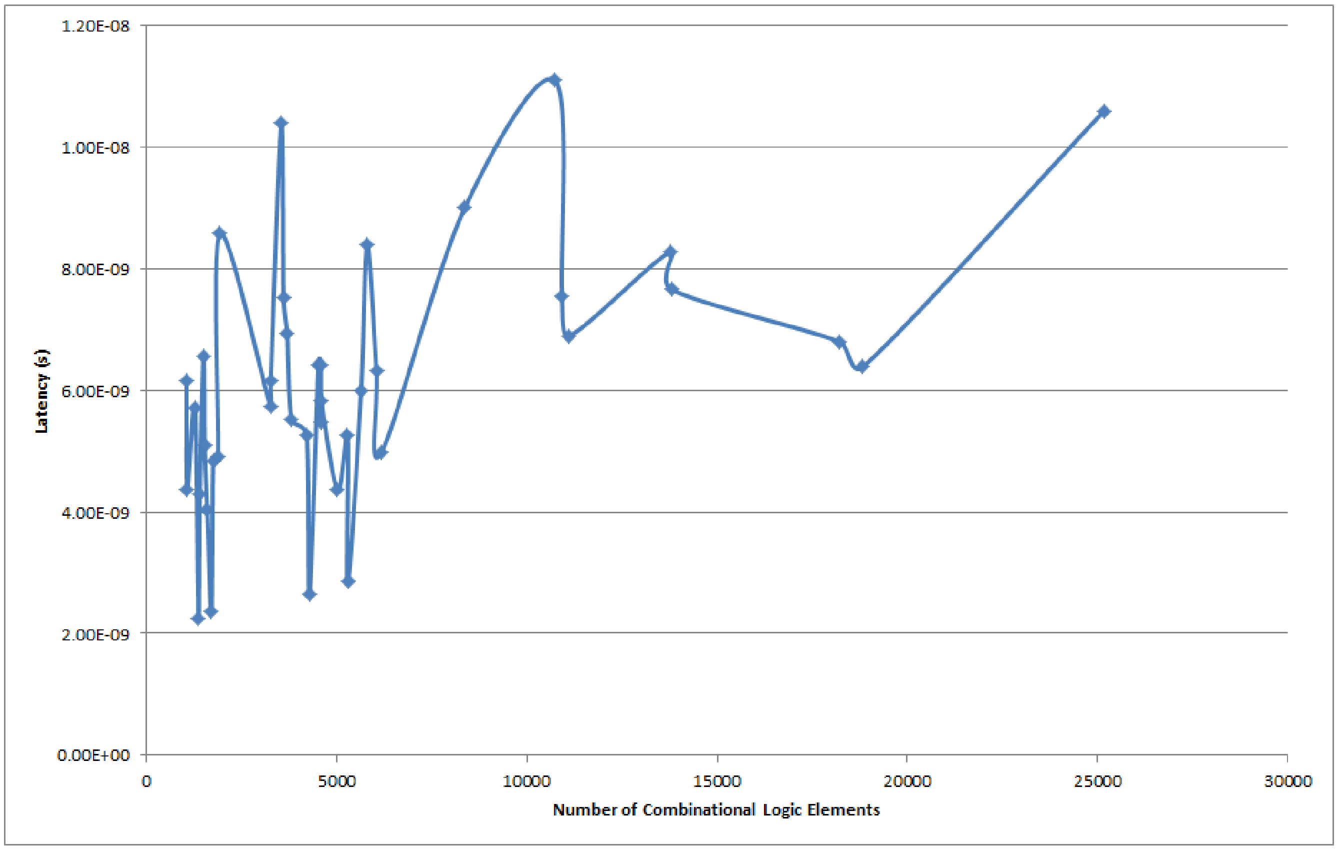


Figure 3.2: Critical Path Latency compared to number of Logic Elements for 200 width channels

lements

Name	Input	Output	.latch	.names	FPGA Area (width in CLBs)	Av. Wire Seg-ments	Used Logic Block Area	Critical Path	VPR Time
pdc 200 width	1	1	0	3.00	1.76	1.04	3.01	1.33	4.25
tseng 200 width	1	1	3	3.12	1.69	1.26	3.08	0.98	3.88

Table 3.1: Scale factors for circuit with maximum critical path slowdown

timing2

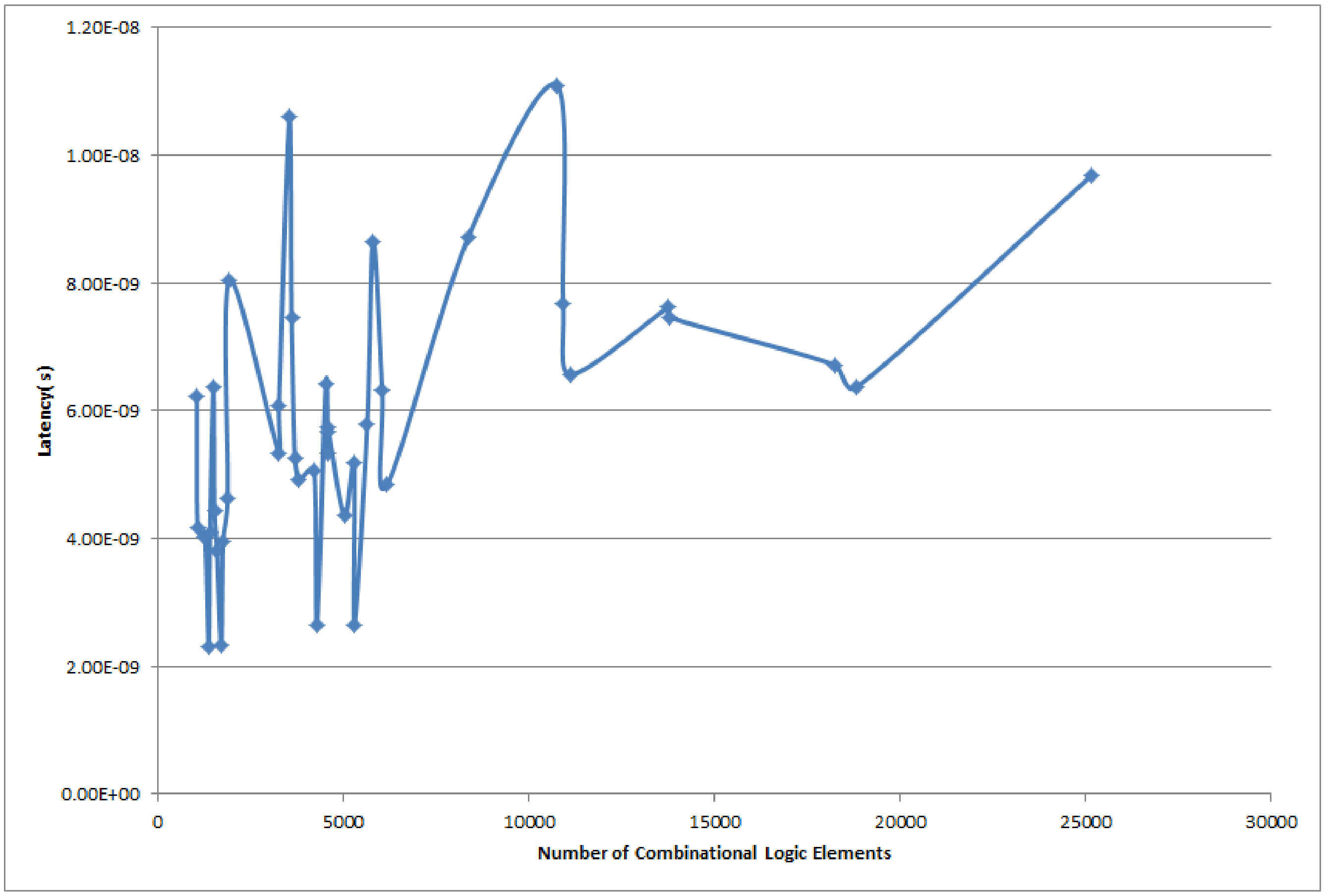


Figure 3.3: Critical Path Latency compared to number of Logic Elements for autosized channels

lements

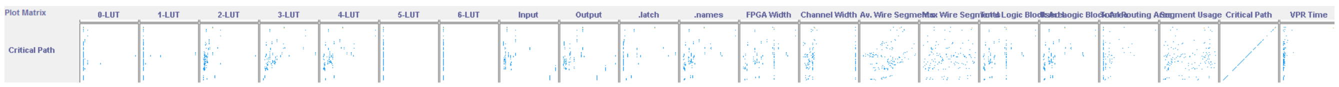


Figure 3.4: Relation between Critical Path Latency and other parameters

lements

AreaVElements
default settings (mostly area driven) giving us the linear relationship shown in Figure 3.4, however even when completely timing driven the packer still tries to fill every CLB [Luu, 2010].

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. Weka[1] was used to examine the data for relations between the critical path length and other measured variables In Section BenchmarkResults 2.4 we saw that the median slowdown was 17% with a 33% 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)

[1]A machine learning and data mining toolkit available from `http://www.cs.waikato.ac.nz/ml/weka/`

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 (above the line is minimum, below the line are nice extras we will be aiming for):

- 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.

4.2 Schedule

Tuesday Week 12: This report due.

Rest of Week 12 to early Week 13: Catching up on other work/subjects.

Stuvac/Exam period: First cut of partitioning algorithm, start collecting benchmarks.

Holidays end 2012: Collect more benchmarks, start analyzing results, play around with VPR settings.

Holidays start 2013: Data mine results for relationships to improve estimation functions.

Next Year: Keep tweaking estimation functions to try and improve performance, up until Demo and Thesis B due. Extract good/interesting results to discuss further.

Demo.

Thesis B.

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.

Glossary

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

Bibliography

VTMR [VTM,] 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.

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

ftmr [ftm, 2002] (2002). Functional triple modular redundancy. Technical report, Gaisler Research.

tmrtool [tmr, 2012] (2012). Xilinx tmrtool product brief. Technical report, Xilinx.

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

VPRBook [Betz et al., 1999] Betz, V., Rose, J., and Marquardt, A. (1999). *Architecture and CAD for Deep-submicron FPGAs*, volume SECS 497 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Bston, 1st edition.

HFPP [Bobrek et al., 2007] Bobrek, M., Woord, R. T., Ward, C. D., Killough, S. M., Bouldin, D., and Waterman, M. E. (2007). 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.

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

lChange [Cetin and Diessel, 2012] Cetin, E. and Diessel, O. (2012). 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. CHANGE.

ttonSEU [Dutton and Stroud, 2009] Dutton, B. F. and Stroud, C. E. (2009). Single event upset detection and correction in virtex-4 and virtex-5 FPGAs. ISCA International Conference on Computers and Their Applications.

PGAArch [Farooq et al., 2012] Farooq, U., Marrakchi, Z., and Mehrez, H. (2012). *Tree-based Heterogeneous FPGA Architectures*, chapter 2. Springer, 2012 edition.

AReview [Habinc, 2002] Habinc, S. (2002). Suitability of reprogrammable fpgas in space applications. Technical Report 0.4, Gaisler Research.

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

RManual [Luu et al., 2012] Luu, J., Betz, V., Campbell, T., Fang, W. M., Jamieson, P., Kuon, I., Marquardt, A., Ye, A., and Rose, J. (2012). *VPR User's Manual*. VPR.

FPGATMR [Marty, 2004] Marty, B. (2004). Virtual field programmable gate array triple modular redundant cell design. Technical report, Schafer, AIR FORCE RESEARCH LABORATORY/VSSE.

nSwitch [Masud and Wilton, 1999] Masud, M. and Wilton, S. (1999). A new switch block for segmented fpgas. In Lysaght, P., Irvine, J., and Hartenstein, R., editors, *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 274–281. Springer Berlin / Heidelberg.

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

tialTMR [Pratt et al., 2008] Pratt, B., Caffrey, M., Carroll, J., Graham, P., Morgan, K., and Wirthlin, M. (2008). Fine-grain seu mitigation for fpgas using partial tmr. *Nuclear Science, IEEE Transactions on*, 55(4):2274 –2280.

nchmark [Prechelt, 2000] Prechelt, L. (2000). 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.

chanism [Sturesson, 2009] Sturesson, F. (2009). Single event effects (SEE) mechanism and effects.

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

XCell33 [Westfeldt, 1999] Westfeldt, W. (1999). Who's using virtex and spartan fpgas in xilinx online applications? In Collins, C., editor, *XCell*, number 33 in XCell, page 10. Xilinx.

Lecture [Wilton,] Wilton, S. Fpga architectures. Course Lecture Notes.