VPR Assessment of a Novel Partitioning Algorithm

David Munro

September 29, 2012

Abstract

Space based systems, etc, so we want ”To implement the partitioning algorithm and assess the results using Versatile Place and Route (VPR) and MCNC benchmarks”.

Acknowledgements

Thanking various people

Contents

Chapter 1

Introduction

1.1 Overview

•To implement the partitioning algorithm and assess the results using Versatile Place and Route (VPR) and MCNC (Microelectronics Centre of North Carolina) benchmarks. •VPR used as it's open source.

•What are we partitioning and why? •FPGA's are useful for space based applications due to low cost, wide availability, etc. [1]. •Downsides include increased susceptibility to radiation induced errors[1]. For Virtex 4 in geosynchronous orbit, predicted mean time between errors is only 1.4s[2]. •Use Triple Modular Redundancy (TMR) to detect errors.

FPGA

ro:FPGA
ro:ASIC

What is an FPGA? ——————————————————————— Field Programmable Gate Arrays (FPGAs) are popular devices capable of implementing a wide variety of circuits. Unlike Application Specific Integrated Circuit (ASIC) circuits 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 individual user's needs. Their flexibility, low cost, and faster development process make them popular for a number of applications.

cro:CAB
acroLWC
cro:BLE

An FPGA consists of a grid of Configurable Logic Blocks (CLBs) (sometimes called Logic Array Blocks (LABs)), generally containing flip flops and Look Up Tables (LUTs); a number of Input/Output (I/O) blocks, allowing for inputs and outputs from the FPGA; and routing channels containing the wires between blocks. 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 ! An n-input Look Up Table (written as n-LUT) can be programmed to represent any n-input boolean function, while latches allow for register transfer operations. A CLB is made up of a number of smaller blocks, called Basic Logic Elements (BLEs), with each block containing the flips flops and LUTs. There is then programmable routing both within each CLB and in between CLBs.

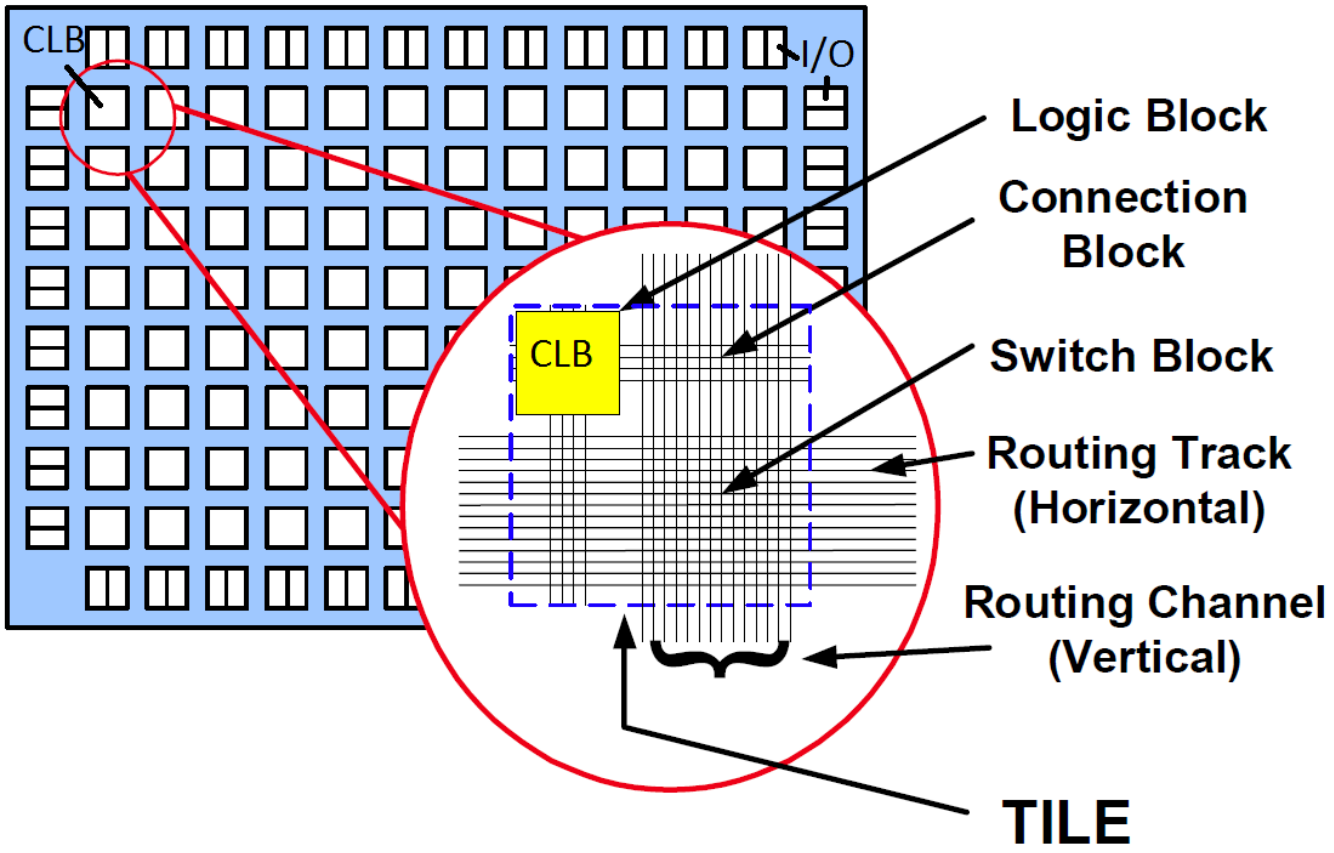

Figure 1.1: FPGA Architecture [Wilton,]

FPGAArc

There are three main components to an FPGA. I/O blocks, usually around the edge, allowing for input and output from the FPGA, CLBs containing all the logic elements (LUTs, latches, etc), and the routing between everything. 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. CLBs are a cluster of smaller blocks, called BLEs, with each BLE containing the basic logic elements, typically a programmable LUT for combinational logic, a latch for register operations and allowing sequential logic, and a **mux!** (**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 switch states.

cro:mux

Programming an FPGA involves loading in a bitstream which describes all the component values (i.e. contents of each buffer) for a circuit. 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.

There are four main technologies used to implement the reprogrammable buffers in FPGAs: **SRAM!** (**SRAM!**), which gives the highest density devices and includes the Virtex 5 family this thesis focuses

ro:SRAM

on, however are volatile and must be reprogrammed every power up from an external configuration memory; (anti)fuse, which are only one time programmable; EPROM, **EEPROM!** (**EEPROM!**), which are reprogrammable, non-volatile this not requiring an external configuration memory, however have a lower density than **SRAM!** FPGAs.

The reprogrammable buffers can be implemented using a storage technology of choice, however in practice there only three technologies which see common use:TODO: reference **SRAM!**, which gives the highest density devices and includes the Virtex 5 family this thesis focuses on, however are volatile and must be reprogrammed every power up from an external configuration memory; antifuse, which are only one time programmable; and**EEPROM!**, which are reprogrammable and non-volatile, thus not requiring an external configuration memory, however have a lower density than **SRAM!** FPGAs.

Partial Reconfiguration

Partial reconfiguration involves loading configuration information for part of a circuit during operation. Much like the complete configuration described above, which happens at power up, 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 **ICAP!** (**ICAP!**), each with a bandwidth of 400MB/s in all Virtex devices TODO: reference: tc-draft.

Space Based Applications

A common application for FPGAs is in space based applications (satellites, etc). The lower flat costs, increased flexibility, and other advantages make them quite suitable for space based applications. However, there is a significant downside. As systems go further into space, and are no longer protected by the earth's atmosphereTODO: Check this, they become increasingly likely to suffer from radiation induced Single Event Upsets (SEUs), where a charged particle TODO: Need to clarify particles vs em radiation (waves)? induces an incorrect signal. TODO: Add in table and diagram. In an ASIC the effect of a charged particle affecting a component is transient. While the erroneous pulse may cause run on effects, be latched, or otherwise affect the circuit in future, the component itself continues operating normally. FPGAs are vulnerable to an additional error type. When the charged particle impacts a buffer it can flip the state of the buffer changing the actual circuit. Unlike transient errors, these errors persist indefinitely. TODO: images of this. Picture of transient, picture of permanent. Need to be a slideshow? 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), and incorrect circuit will be loaded.

(Anti)fuse devices, being non reprogrammable, are immune to these errors, however both **SRAM!** and **EEPROM!** based FPGAs are vulnerable TODO: reference.

	POWER	SPEED	HARDNESS (e/b-d)	AREA (mm^2)
Std Low Power	Rise – 0.7 mW Fall – 0.2 mW	Rise – 0.21 ns Fall – 0.27 ns	10E-8 1 node	360
Increased IDRIVE	Rise – 1.0 mW Fall – 0.2 mW	Rise – 0.16 ns Fall – 0.15 ns	2x10E-9 1 node	460
Low Power triplicate-and-vote	Rise – 1.72 mW Fall – 1.27 mW	Rise – 0.21 ns Fall – 0.27 ns	10E-11 2 node	1200
DICE	Rise – 1.4 mW Fall – 1.1 mW	Rise – 0.96 ns Fall – 0.97 ns	1.6 x10E-10 2 node	520

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. 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 and usually increase power usage.
- Temporal Filtering, which aims to filter out transient SEUs, such as delay-and-vote [Baze et al., 2002]. 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, including the common technique of Triple Modular Redundancy (TMR) which this thesis is mainly concerned with. These techniques increase area and power usage significantly, however require no custom hardware and greatly improve the circuit's hardness.

cro:TMR

Harden systems, use ASIC, partial TMR, TMR without partial reconfiguration, Virtual TMR [Habinc, 2002], [VTM,], [Baze et al., 2002]

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. For any SEU it will affect the output value of at most one circuit, so the majority vote is still correct. One can then incorporate partial reconfiguration in order to recover from detected errors, as described in TODO: reference. 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 can be 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. 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 system availability threshold.

$$\begin{aligned} \text{Error Recovery Time} &= \text{Error Detection Time} + \text{Reconfiguration Time} + \text{Resynchronisation Time} \\ \frac{1}{\text{Error Recovery Time}} &= \text{System Availability} \end{aligned}$$

$$\text{Error Detection Time} <= \frac{1}{\text{Clock Frequency}} \times \text{Pipeline Steps}$$

$$\begin{aligned} \text{Reconfiguration Time} &= \frac{1}{\text{Reconfiguration Speed}} \times \text{Bitstream Size} \\ &= TODO : ActualValues \times \text{Partition Size in Slices} \end{aligned}$$

$$\text{Resynchronisation Time} <= \frac{1}{\text{Clock Frequency}} \times \text{Pipeline Steps}$$

Additionally, as each voter circuit adds some constant overhead, in terms of area, power usage and clock frequency, so it is desirable to have each partition as large as possible. To this end, this thesis looks at implementing the partitioning algorithm described in TODO: reference, and assessing its performance. !

1.3 Prior Work, lit review, etc?

Some lit review and prior work stuff. ———————————————————————————— This thesis builds on the work of TODO: reference which details an overview of the partitioning ! algorithm we are implementing Our goal is to create an algorithm which gives good performance, stays within a user specified system availability limit, doesn't require existing code to be rewritten, allows for both voting logic and reconfiguration logic to be added, and can use industry standard FPGAs. There are a number of existing TMR solutions, however none quite meet our requirements. [ftm, 2002] requires existing code to be rewritten. [syn,] and [tmr, 2012] don't seem to support staying within a system availability limit, nor for adding reconfiguration logic. Other options look at using partial TMR to reduce the associated overhead ([Pratt et al., 2008]), require custom hardware ([Marty and Lyke, 2004]), or otherwise don't meet our criteria.

1.4 CAD Design Flow

Talking about the overall CAD design flow. ODIN -¿ ABC -¿ Partitioner -¿ VPR, etc.

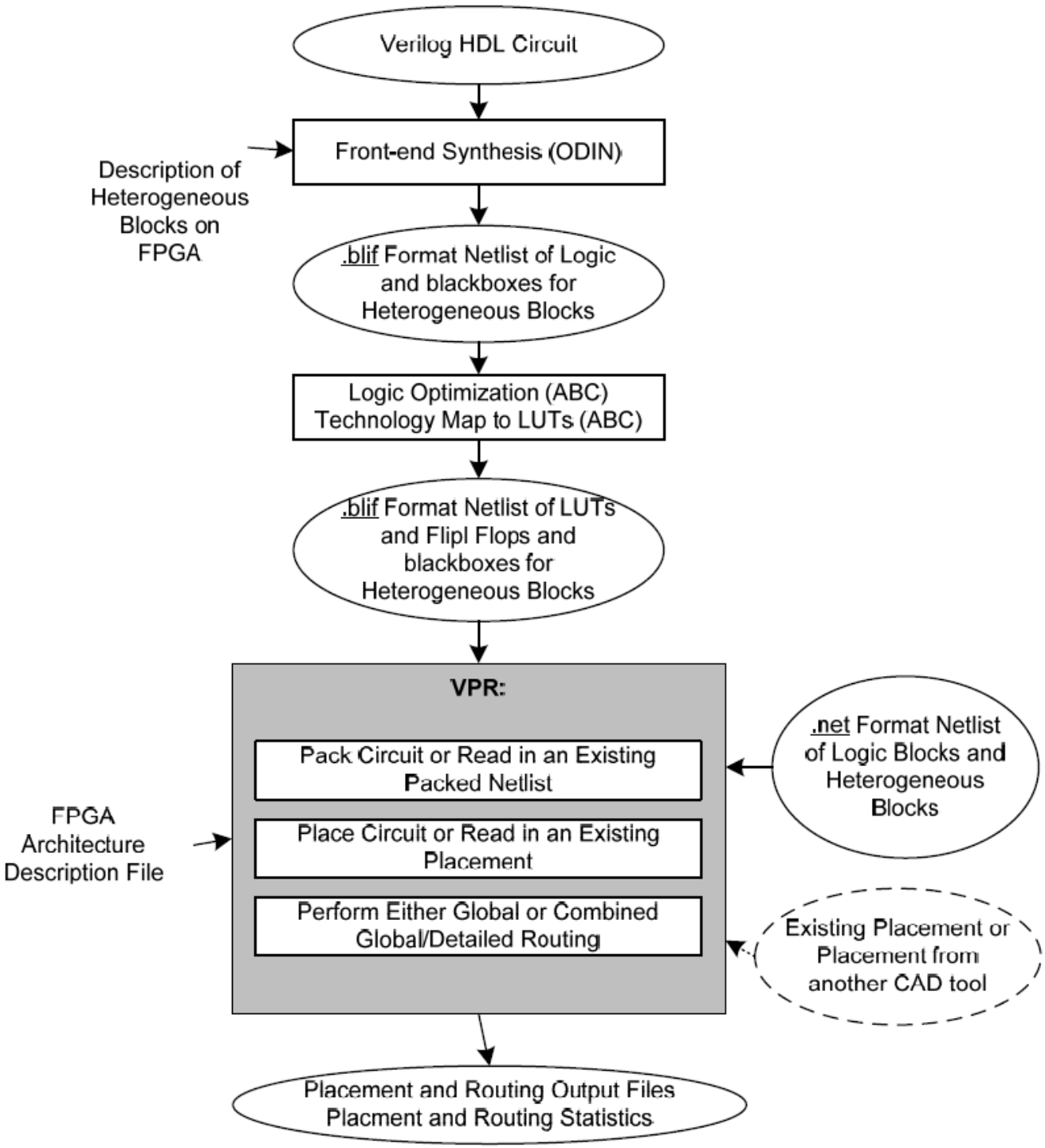

Figure 1.2: Cad Design Flow. Sourced from TODO: `CADFlow`

How VPR Works

Default settings for VPR and what they all change, algorithms used, etc.

Chapter 2

Benchmarking

2.1 Overview

We have a number of benchmark circuits (detailed in benchmarkList 2.1) 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 Directed Flow Graph (DFG), without needing to actually place and route the partial circuit after each iteration, as we desire a partitioner with speed of the same order as the other steps in the **CAD!** (**CAD!**) flow.

To start with we make 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 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 **SIS!** (**SIS!**) to flatten it into a format Versatile Place and Route (VPR) can read.

2.2 Why VPR

VPR is an open source packer, placer, and router commonly used for research. We chose VPR as, largely due to it being open source and popular, it is well documented, uses easy to read file formats, and is easy to modify if necessary. Additionally, all of the algorithms used to place and route are available and well described in literature.

	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

benchma

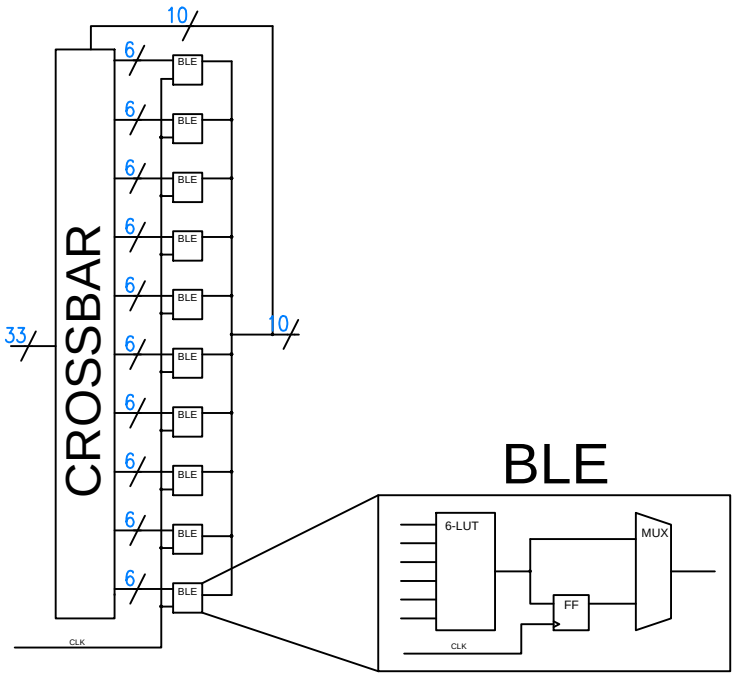

Figure 2.1: CLB Architecture

Arch

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.

Table 2.2: Architecture Elements

Arch

Architecture File

VPR allows us to specify a custom architecture for it to run against in an XML format, which for those interested is well documented at TODO: reference. As knowledge of the format is rather tangential to the goal of this thesis we will only be discussing the architecture we used, and not the specifics of the format.

The format is fairly similar to the Virtex 5, 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!**) switches.

:Wilton

2.3 Methodology

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, 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 TODO: reference 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. If –route_chan_width is excluded then VPR determines the minimum channel width needed to successfully route the circuit TODO: reference. 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. TODO: List of defaults for VPR parameters

The place and route process has a random factor to it, due to the methods used (simulated annealing for placement TODO: reference and TODO: what does it use to route?), however generated results are

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

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

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

usually within TODO: How close and reference of each other, so to save time the placement and routing process is only run once per circuit (depending on circuit and parameters, a circuit may take in excess of several hours to place and route for some extreme cases.

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 or excluded. 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 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.

`maxRes` 2.4 is our worst case slowdown, with a critical path of 33% longer.

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

Partitioning in action •Start with inputs. Partitioning wavefront •TMR-ify node-set. •Repeat process until all nodes are TMR'd. •Add nodes in a breadth first manner. •Continue until area, frequency or critical path exceeds threshold. Max recovery time Estimated recovery time Area Critical Path Frequency 1.00E-08 1.00E-09 20 1 100MHz 5.00E-30 2 64MHz 2.00E-08 40 50MHz

To do this, need a way of efficiently calculating area, frequency and pipeline length for a set of nodes. •Pipeline length is trivial, the other two not so much. •No way to tell until design is routed, which takes too long, therefore we need some way of estimating. •Also, to effectively traverse, need circuit as a graph. VHDL/Verilog too high level, needs to happen post-synthesis, somewhere in CAD flow.

Given a DFG our goal is to traverse the DFG and partition it, such that every node is in exactly one partition, and the recovery time for each partition is less than our limit. 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. 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. A possible improvement in future is improving the traversal algorithm to be more intelligent than just breadth first, to try and maximise each partition's size, though further work will be needed to

determine if such optimisations are effective. TODO: pictures and worked example !

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 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, discussed earlier in TODO: reference, namely that ! its free and open source nature made it amenable to modification, and possible to examine how every step operates, rather than being a black box using some proprietary file format and algorithm, 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 a higher element density per partition.

Additionally, we were faced with a choice of when in the **CAD!** design process (outlined in TODO: ! reference) 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 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. 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 Reference?. 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

A detailed description of the BLIF file format can be found at TODO: reference, however an overview of the format and features used is included below.

pleBlif

```
1 .model voter
2 .inputs a b c
3 .outputs out
4 .names a b c out
5 11- 1
6 1-1 1
7 -11 1
8 .end
```

Listing 3.1: Sample BLIF file

A BLIF file is a plain text file which simply lists all the elements of a circuit, and their inputs and outputs. VPR (and hence our partitioner) only supports a subset of the BLIF file format, detailed in table TODO: reference, and insert table.

A BLIF file consists of a module declaration (.module), followed by a list of all input elements (.inputs in1 in2 ...), then a list of all outputs (.outputs out1 out2 ...), then a list of clocks (.clocks clk1 clk2 ...), then a list of all the circuit elements (.latch (latch) and .names (combinational logic)), and finally an optional .end. VPR only supports flat BLIF files, so only one module declaration is allowed per BLIF file. **SIS!** or **ABC!** (**ABC!**) TODO: Check that ABC can flatten? can be used to flatten BLIF files for use by VPR.

cro:ABC

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.

Pseudocode

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

udocode

```
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 Reference, 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 TODO: section 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!** TODO: reference/explain 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

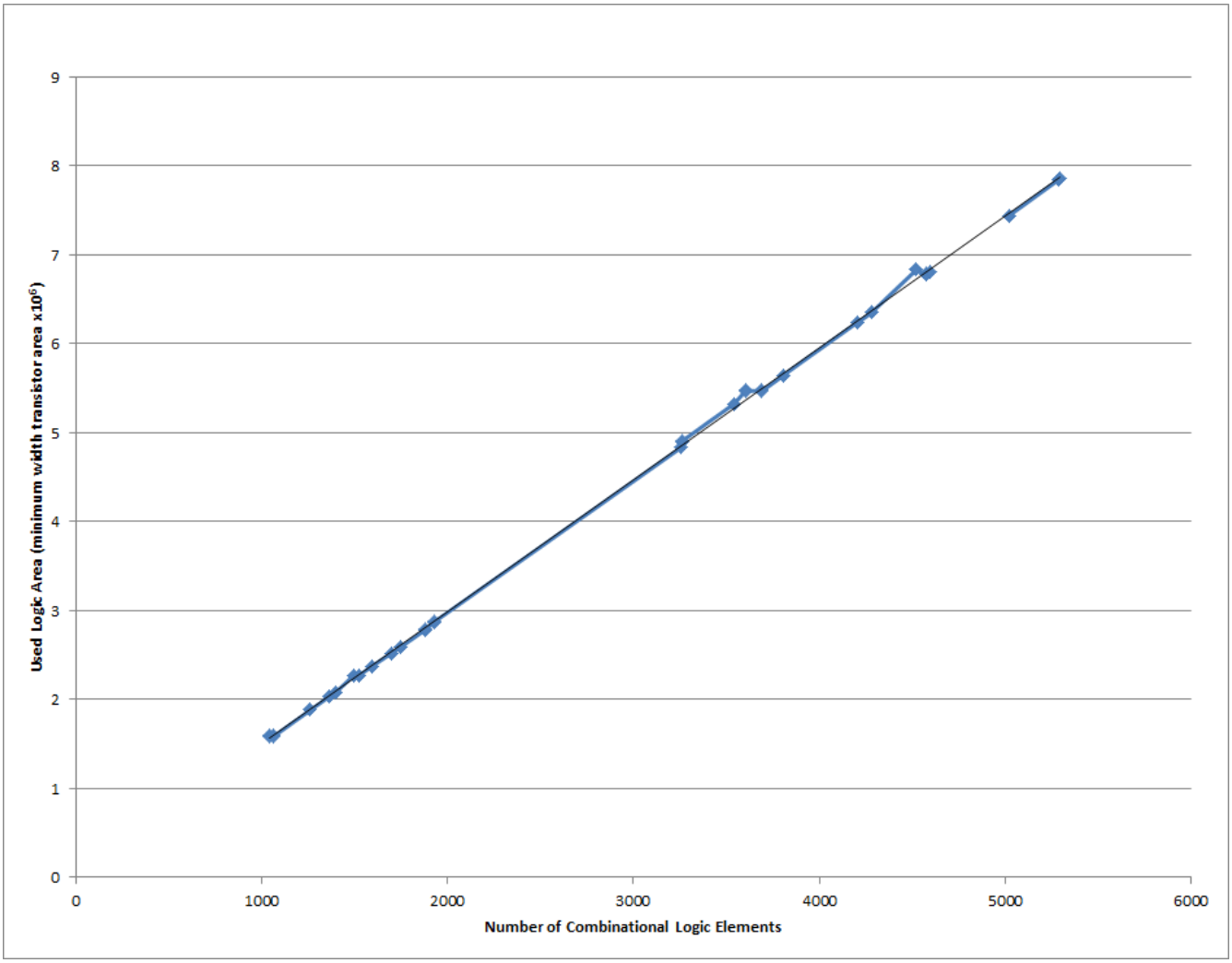

Figure 3.1: Circuit Area compared to number of Logic Elements AreaVEl

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 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

As is shown in the results TODO: table/graph reference 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 !

TODO: reference to where we describe it and what it has 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 3.1. 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.

AreaVElements

Timing information, on the other hand, is harder to estimate, with no obvious pattern. Generally the maximum frequency is within 10% slower, though for some cases it goes down to 30% slower. 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.

3.6 Progress

Still in progress, hopefully nearly finished with first cut. 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.

cro:VTR

- Incorporate that single program into Verilog To Routing (VTR)'s design flow, likely as part of VPR.

Chapter 4

What next

4.1 What is still to be done

•Using very basic and arbitrary estimation functions for area and timing, implement partitioning. •Improve estimates.

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

Holidays: 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

All on track, difficult parts will be timing estimates, algorithm may be somewhat restricted in real world applications to problems i.e. better suited for benchmarking the effects of TMR upon a circuit, then actually automatically partitioning a circuit.

Glossary Quick explanations of what things are (BLE, CLB, DFG, Wilton, etc)

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.

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

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

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.

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