VPR Assessment of a Novel Partitioning Algorithm

David Munro

September 24, 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

I should probably thank 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.

1.2 Triple Modular Redundancy

•Make three copies of a circuit, and feed the outputs to a voter. •Once an error is detected we can fix it by e.g. selectively reconfiguring the incorrect module. •Clock frequency, pipeline length and circuit area all affect recovery time. •Need to detect, reconfigure and resynchronise within error rate. TMR Module 1 Dout1 TMR Module 2 Din Dout2 TMR Module 3 Dout3 Voter Dout Source Cetin & Diessel (2012) Bad TMR Module 3 ———————————————————————————————- 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 to0 determine the majority value. For any Single Event Upset (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 occuring 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,

acro:SEU

!

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. 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. ——————————————————————————— 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 Field Programmable Gate Arrays (FPGAs). There are a number of existing **TMR!** (**TMR!**) solutions, however none quite meet our requirements. [2] requires existing code to be rewritten. [1] and [3] 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 ([5]), require custom hardware ([4]), or otherwise don't meet our criteria.

1.4 CAD Design Flow

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

Chapter 2

Benchmarking

2.1 Overview

Benchmarking •Two things we're looking at. •1. Comparing circuits with and without TMR, as per thesis title. •2. Trying to find a relationship, or suitable guesses, for our partitioner. •For now we are calculating minimum channels and area to look at general trends. Later we look at the case where we are close to channel/area capacity. •Circuits used are MCNC (Microelectronics Centre of North Carolina) benchmarks, provided by the VTR Project. ——————————————————————— We have a number of benchmark circuits (detailed in benchmarkList 2.1) we will be using to evaluate the performance of our partitioner and our **TMR!** scheme in general using the Microelectronics Centre of North Carolina (MCNC) benchmarks provided with Verilog To Routing (VTR). Additionally, we're looking for patterns, or ways of estimating area usage and timing information from a **BLIF!** (**BLIF!**) file or Directed Flow Graph (DFG), without needing to actually place and route the partial circuit after each iteration, as it would take too long.

acro:MCNC

acro:VTR

acro:BLIF

To start with we use a python script to just triplicate some provided circuit, and insert 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 $elements_{circuit} >> elements_{voter}$.

2.2 Why VPR

How and why we use VPR to collect data, what settings/options we used, etc.

Open source. •Open file formats. •Want to partition as late as possible. •Adding elements means routing and placing again =¿ partition just before or after packing. •Before packing is easier, but might be less effective. ———————————————————————

acro:VPR

——————— Versatile Place and Route (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

	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

benchmar

literature TODO: why is this good?

!

Architecture File

Use imaginary standard architecture, somewhat similar to Virtex 5. •FPGA are and routing channels grow to accommodate design. CLB and BLE layout •Customisable, so can make more accurate to target a specific platform. ——————————————————————— TODO: Insert image of architecture The architecture file format is well described by TODO: Insert reference to manual so we shall content ourselves with merely detailing the reference architecture we're using. It's relatively similar to the Virtex 5, consisting of a grid of Complex Logic Blocks (CLBs) each consisting of ten fully interconnected Basic Logic Elements (BLEs), and each BLE having a latch and 6-Look Up Table (LUT). Each BLE has TODO: number of inputs/outputs. Each CLB has TODO: number of inputs/outputs. Routing resources consist of a number of length 4 tracks with **Wilton!** (**Wilton!**) switches.

!
!

acro:BLE

acro:LUT

o:Wilton

!
!

2.3 Methodology

Our procedure for collecting data. ——————————————————————- 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 simply Python script to

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 2.2: Median Scale Factors for specified channel widths

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 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, maximum path length, and other statistics we analysed. Additionally, our partitioner reported the critical path length, a factor in error recovery time.

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 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

Results •Quite close to initial guess. •Area requirement slightly greater than tripled. •Mean increase in critical path time is 6.9E-10s. Max is 1.87E-09s increase to 8.28E-9s (156MHz-¿121MHz max frequency) The results listed in this section highlight the information of interest. More complete results, including the data used to calculate the medians in this section, can be found in TODO: reference to appendix.

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 2.3: Scale factors for circuit with maximum critical path slowdown

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 2.2 this matches exactly. 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 a 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 2.3 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 !

3.3 Input file format

Input format description and what we do with it. ——— 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. TODO: Insert 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!** (**SIS!**) or **ABC!** (**ABC!**) TODO: Check that ABC can flatten? can be used to flatten **BLIF!** files for use by VPR.

3.4 Implementation

Read blif file into in-memory graph. •Partition graph. •Insert voter logic into blif file as appropriate. •File format is text input of inputs, outputs and logic elements. •Only logic elements supported or Look Up Tables (LUT) and latches. ———

———

The file format is a simple text file format listing each node, and each node's inputs and outputs. We read the file into memory, stored as a DFG, with all nodes and signals additionally stored in a hashmap. Each node contains a list of all connected signals, and each signal contains a list of all its sources and sinks. This makes it 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. As we traverse the DFG we keep running track of an estimate of the current partition's area and timing information. Once the constraints are exceeded 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.

The reason for the somewhat Goldberg machine process involving multiple programs and scripts is to reuse existing and known working code as much as possible, rather than reimplementing functionality already found elsewhere. A possible improvement in future is to incorporate all functionality in a single program, which should be relatively easy as all code is open source, and the necessary functionality is relatively simple.

Choice of Language

Using a combination of languages, depending on what specifically it's being used for, mainly Python and C++, with a bit of Perl and C. For blif joining and inserting the voting logic Python was used. Blif files are plain text so parsing and inserting additional text is quite simple with Python. Perl was originally considered however TODO: make up a reason why Python is better. For the actual partitioner C++ was chosen. VPR is written in C which allows for easy code reuse between the two, or for the partitioner to be actually added into VPR. C++ was chosen over C as it was felt the object oriented nature of C++ allowed for easier to maintain code, and the addition of the Standard Template Library (STL) simplified the implementation and reduced the dependency on external libraries for basic tasks such as string parsing (a large part of dealing with blif files) and creating a DFG.

acro:STL

3.5 Estimating restrictions

•We also care about estimating area and latency from a DFG. •Clear relationship between nodes and area.

How we estimate our values for area and time. 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 is more difficult, as they rely on the placement and routing of the circuit, however 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 around 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

Results – Estimating Time •Much harder to estimate time before placement. •Options: •Guessing. Guided by number of blocks or pre-TMR time, may not be entirely inaccurate. •Partial placement. Placer uses simulated annealing, so run very rough placement to get estimate [3]. •Partition after placement. Better knowledge, much harder to do. 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 relatively easy to accurately estimate, as there is a clear relationship between the number of nodes and the area usage (in CLBs). 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 is simply $max(num_{latch}, num_{names})$. Each BLE is completely interconnected within each CLB allowing for each CLB to be fully packed, giving an area usage, in CLBs of $\frac{max\ num\ elements}{num\ \text{BLE}s\ per\ \text{CLB}}$. This is shown nicely in graph TODO: reference with the linear relationship between number of elements and area usage. Note that we're looking at the case where we try to minimise area used. Some optimisations may use more area in exchange for a higher max frequency, which would skew our results.

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 a pessimistic 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.

3.6 Progress

Still in progress, hopefully nearly finished with first cut •TMR arbitrary user specified subcircuit preserving surrounding circuit. •Just need to automate subcircuit selection, instead of user defined. •To do that need way of estimating area/timing from set of nodes.

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

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.

Detailed Results Lots of tables listing the results collected.

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

Bibliography

synplify [1] Using synplify to design in microsemi radiation-hardened fpgas. http://www.actel.com/documents/SynplifyRH_AN.pdf.

ftmr [2] Functional triple modular redundancy. Technical report, Gaisler Research, December 2002.

tmrtool [3] Xilinx tmrtool product brief. Technical report, Xilinx, 2012.

milVTMR [4] James C. Lyke Barbara Marty. Virtual field programmable gate array triple modular redundant cell design. Technical report, Schafer, AIR FORCE RESEARCH LABORATORY/VSSE, mar. 2004.

rtialTMR [5] 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, aug. 2008.