StitchUp: Automatic Control Flow Protection for High Level Synthesis Circuits

Shane T. Fleming and David B. Thomas
Electrical and Electronic Engineering Dept, Imperial College London, U.K.
{sf306 | dt10}@ic.ac.uk

ABSTRACT

Soft-error detection in FPGAs typically requires replication, doubling the required area. We propose an approach which distinguishes between tolerable errors in data-flow, such as arithmetic, and intolerable errors in control-flow, such as branches and their data-dependencies. This approach is demonstrated in a new high-level synthesis compiler pass called StitchUp, which precisely identifies the control critical parts of the design, then automatically replicates only that part. We applied StitchUp to the CHStone benchmark suite and performed exhaustive hardware fault injection in each case, finding that all control-flow errors were detected while only requiring 1% circuit area overhead in the best case.

1. INTRODUCTION

High-level Synthesis (HLS) tools, such as LegUp [2] and Vivado HLS [5], can automatically generate FPGA circuits from C code. Listing 1 gives a simple example of a synthesisable dot-product, which combines data-dependent control-flow in the for loop (line 3), and pure data-flow in the body of the for loop (line 4). HLS tools will generate a circuit which executes correctly in a reliable FPGA, but what if this circuit is required to execute in the presence of soft errors? The naive approach would be to instantiate two or more copies of the circuit and run them in lock-step on the same input data, with any deviations between their states or outputs indicating an error. Naive replication is expensive in both area and more importantly power, especially in this example as expensive floating point arithmetic units are required for the multiply accumulate computation (line 4).

When tight area and power constraints prevent full replication, then reliability can still be improved through selectively replicating "important" portions of the design. Studies have shown that some applications, such as media processing [9] and machine learning [15], contain critical regions of code that are sensitive to soft errors, and non critical regions that can tolerate soft errors.

Control-flow regions, (i.e. instructions effecting branch decisions), are arguably more critical than data-flow, since faults within control-flow can effect real time guarantees or prevent program termination. Previous work has demonstrated this, with [4, 3] generating reliable HLS circuits for control-flow intensive (CFI) ASIC designs, and [12, 10, 15] performing fault injection experiments demonstrating that control-flow is vulnerable to errors.

DAC '16, June 05-09, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2897937.2898097

```
1 float DotProduct(float A[20],float B[20])
2 { float c = 0.0;
3   for(int i=0; i<20; i++)
4     c += A[i]*B[i];
5   return c;
6 }
```

Listing 1: Dot Product Example

For example, faults within the control structure of our dot product example (Listing 1) could cause elements of `A` and `B` to be skipped in the calculation of `c`, or even worse, cause the circuit to enter an infinite loop and never terminate. This paper presents an approach for automatically extracting the control flow structure for a HLS application so that it can be selectively replicated to protect control decisions and ensure all branches are taken correctly.

Figure 1(a) provides a diagram of the default HLS generated circuit for our dot product example. Two clear partitions can be seen, a data path where functional units reside, and a control FSM responsible for scheduling instructions onto the functional units. In order to replicate only the control-flow structure, any data-path functional units which may influence a state transition need to be duplicated along with the control FSM. We shall refer to the collection of the control FSM and control dependent data-path elements as the *control replicant*, Figure 1(b) shows the control replicant for our dot product example.

Analysing the code in Listing 1 we can determine that the instructions which influence control-flow decisions are the ones used in the `for` loop conditions, which require an integer comparator **icmp** and adder **iadd**. In comparison, the expensive floating point units, **fpmul** and **fpadd**, required for calculating the multiply add accumulate have no influence over any branch decision and will not be replicated.

Manually inspecting code and removing elements that don't influence control flow is simple in the case of our dot product example. However as the complexity of the input program increases, the engineering effort required to both analyse the input code and generate the replicated control FSM is significant. StitchUp fully automates this process through the use of static analysis technique, known as program slicing, to extract and duplicate any instructions that may influence control. The contribution of this paper are:

- StitchUp, an approach for decoupling control-flow and data-flow to increase circuit relability. Along with an open source LegUp HLS pass that can automatically extract and protect the control flow structure of a circuit.

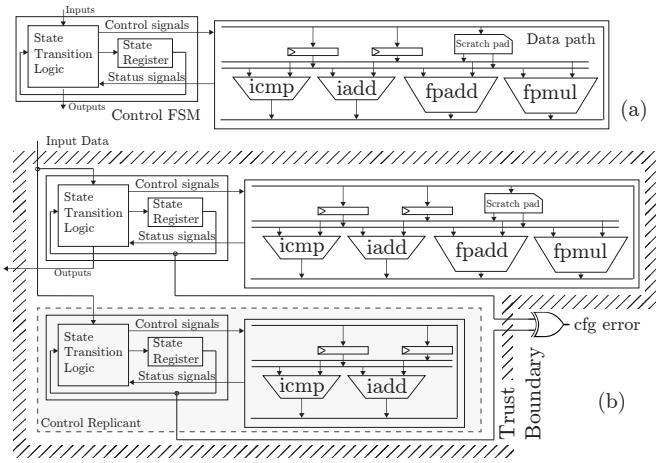

Figure 1: Default circuit (a) and Control Replicant (b) for our Dot Product example in Listing 1, with Trust Boundary

- Detailed resource and power results for StitchUp using the popular CHStone HLS benchmark.
- Exhaustive single bit hardware fault injection, where every essential configuration bit is flipped to assess how well StitchUp can catch errors.

2. FPGA DEVICE FAULTS DUE TO SOFT-ERRORS

A Soft Error or Single Event Upset is a random, non-catastrophic error, where radiation causes a perturbation to a circuit by temporarily altering a single signal or datum. Detecting and mitigating such errors is important in some fields, such as satellite design, where high design costs, harsh environments, and the difficulties in repairing once deployed make it necessary. Additionally, due to the continuing increase in semiconductor density, the current soft error challenges experienced in space will also occur in future terrestrial devices.[11][8].

FPGA devices work by mapping circuit descriptions into a collection of Logic Elements, which are described in programmable Look-Up-Tables (LUTs) and routed together via programmable switchboxes. Configuration data for both the look-up-tables and switchboxes are stored in an SRAM based configuration memory, which means soft errors can either change the functionality of Logic Elements or can change the routing between them. Other memories called block memory and flip-flops are also present in FPGA devices, and soft errors in these regions can cause transient errors in the circuit.

For ASIC designs, registers are the most vulnerable to soft-errors with combinational logic contributing very little to overall circuit relability [1]. This makes ASIC approaches which analyses and protect vulnerable registers, such as in [4], promising approaches. However in FPGA devices the most vulnerable region is the SRAM based configuration memory, so such approaches would have little impact requiring methods that perform combinatorial logic replication, such as StitchUp or DMR/TMR.

Circuits described within configuration memory are usually protected in one of two ways: (1)the circuit is replicated with comparison; (2)ECC/CRC signals associated with frames of the configuration memory are regularly scanned

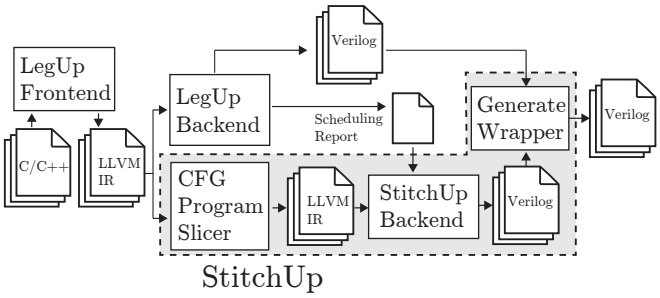

Figure 2: Tool Flow Overview diagram

for errors. Replication uses extra area and power but has low error detection latency and can protect both configuration memory and registers, while ECC/CRC checks consume less area and power in exchange for higher detection latency and an inability to protect registers.

For this paper our *fault model* assumes that a single soft error can occur at most every clock cycle, and that this error can effect block memory, SRAM configuration memory, or flip-flops. Since errors are not solely within the configuration memory, replication must be used, as configuration ECC/CRC cannot protect the registers in our control structure. Figure 1 shows a StitchUp generated circuit for the dot product example in Listing 1, with the original circuit on top and the control-flow structure shadow beneath it. StitchUp aims to protect all configuration, block, and Flip-flop bits within the trust boundary (grey box) as everything outside, such as routing to external I/O or DDR, is not protected.

3. STITCHUP OVERVIEW

Finding and extracting the high-level control-flow structure from an RTL description of a large circuit is difficult, whether it is attempted automatically or manually. Finding the high-level control-flow in a high-level input source is easy, but manually extracting an equivalent circuit is still challenging for large designs. StitchUp can automatically perform this extraction as a compiler pass, which can be easily invoked by setting a compiler flag in the LegUp HLS tool. This section discusses how our analysis is implemented in StitchUp enabling the automatic extraction of the control-flow structure along with the generation of the control replicant. An open source version of our tool described in this paper can be found at *https://github.com/STFleming/StitchUp*.

Figure 2 shows the transformation process from a C program to a control-flow protected Verilog hardware description, with StitchUp specific sections highlighted in grey. Initially the C input is passed into the standard LegUp frontend, which is a series of passes that perform various tasks, such as annotating instructions with pipelining information. This outputs an LLVM intermediate representation (LLVM-IR) which is passed both into the backend of LegUp, to generate the original unprotected circuit and also into the frontend of StitchUp to generate the control replicant circuit. Finally an integration stage connects the original circuit to the duplicate control-flow circuit and generates comparison logic to ensure that the state registers match.

Our motivation for integrating our tool with LegUp is primarily due to it being open source, as the StitchUp backend requires modification to the generation of the schedul-

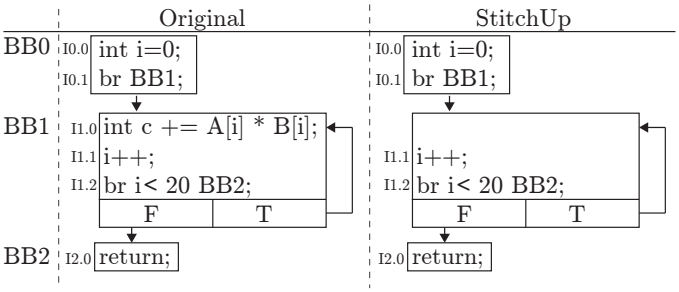

Figure 3: Control-Data-Flow Diagram for Matrix Multiplication example

ing control FSMs. However, the technique itself could be incorporated in VivadoHLS and all other HLS tools through modifying the generated control FSM in a different fashion, (such as modifying the output HDL directly). It is common for modern HLS tool to be constructed on top of LLVM-IR which has a similar level of abstraction to assembly code. It also has two important features for writing compiler optimisations: first, all instructions are in single static assignment form (SSA) where every variable is only assigned once; and second, instructions are grouped into straight-line sequences known as basic blocks (BB) where there is only one entry branch at the very start of the block, and one exit branch at the very end.

3.1 Extracting the Control Instruction Set

LLVM-IR, which is passed into the frontend of StitchUp, is naturally arranged into a CDFG where each node is a basic block and edges are branch between them. An example of the CDFG from the dot product in Listing 1 is shown on the left side of Figure 3. The StitchUp frontend performs a backwards analysis walking the CDFG from bottom to top, collecting any control flow related instructions that may influence whether an edge transition is taken.

All instructions deemed to be control related are collected into what we refer to as a Control Structure Instruction Set (*CSIS*), which can be used to generate a control replicant. Each basic block, i, in the input CDFG will have it's own associated $CSIS_i$ containing all instructions that may effect control-flow after this point in the program. The overall *CSIS* for the input is $CSIS_s$ where s is the initial basic block of the input program.

Algorithm 1 is used to construct the *CSIS* for all nodes in the CDFG. It requires a fixed point iteration until the set of all *CSIS*, C_{CSIS} has reached a stable state which can be seen in line 1-2. Each iteration walks backwards from the final basic block of the CDFG to the start basic block, as seen in line 3, and for each encountered the corresponding *CSIS* is updated using the following rules:

Rule 1 $Sucessor(B)$ returns the set of all basic blocks in the CDFG that follow on from the basic block B. For all successor j of basic block i this rule adds every element of $CSIS_j$ to $CSIS_i$.

Rule 2 Adds all branch instructions for the current basic block, i, to $CSIS_i$ using $TerminatingInstruction(i)$ a function which returns the final instruction from the basic block i.

Rule 3 Any instruction within the current basic block (i) that is used as an operand (op_c) by an element of $CSIS_i$ is added to $CSIS_i$ [1]

Algorithm 1 *CSIS* Extraction Static Analysis Algorithm

INPUT: a program, P, set of Basic Blocks, $\{BB_0 ... BB_N\}$
OUTPUT: P' a program consisting of I_c the set of control structure instructions.

```
1:  while C_CSIS != P_CSIS do
2:      P_CSIS ← {CSIS_BB0, ..., CSIS_BBN}
3:      for each B ∈ {BB_N, ..., BB_0} do
4:  ─────────────────────────────────────────────────
5:          for each S ∈ Successor(B) do              ▷ Rule 1
6:              for each I_c ∈ CSIS_S do
7:                  CSIS_B ∪ {I_c}
8:  ─────────────────────────────────────────────────
9:          T ∈ TerminatingInstruction(B)             ▷ Rule 2
10:         CSIS_B ∪ {T}
11:         for each op ∈ T do
12:             CSIS_B ∪ {op}
13: ─────────────────────────────────────────────────
14:         for each I ∈ B do                         ▷ Rule 3
15:             for each I_c ∈ CSIS_B do
16:                 for each op_c ∈ I_c do
17:                     if I = op_c then
18:                         CSIS_B ∪ {I}
19:     P' ← CSIS_BB0
```

To demonstrate this we will apply Algorithm 1 to the dot product example in Listing 1 to extract the control-flow structure, labelled StitchUp in Figure 3. For each update to a *CSIS*, the label for the instruction added from 3 is given along with the rule from Algorithm 1 used to add it.

Initially $CSIS_i = \emptyset$ for all basic blocks i

Iteration 1:

$CSIS_{BB2}$: add I2.0:**return** using rule 2
$CSIS_{BB2}$ **= {I2.0}**

$CSIS_{BB1}$: add every element in $CSIS_{BB2}$ using rule 1
add I1.2:**br i<20 BB2** using rule 2
add I1.1:**i++** using rule 3
$CSIS_{BB1}$ **= {I2.0,I1.2,I1.1}**

$CSIS_{BB0}$: add every element in $CSIS_{BB1}$ using rule 1
add I0.1:**br BB1** using rule 2
add I0.0:**i=0** using rule 3
$CSIS_{BB0}$ **= {I2.0,I1.2,I1.1,I0.1,I0.0}**

Iteration 2: No Change, fixed point reached

Once the analysis is completed the final value of $CSIS_{BB0}$ is turned into LLVM-IR and given to the StitchUp backend. Here scheduling information from the original unmodified LegUp flow is used to generate an identical control FSM to the original circuit, often requirng the insertion of blank states used to schedule instructions that have not been included in the control replicant. Finally, an integration stage takes the original LegUp and StitchUp circuits, adds comparison logic to compare the state registers of the two circuits in each cycle, and generates a top level wrapper to connect them

[1] For the current version of the tools the worst case is assumed for memory operations.

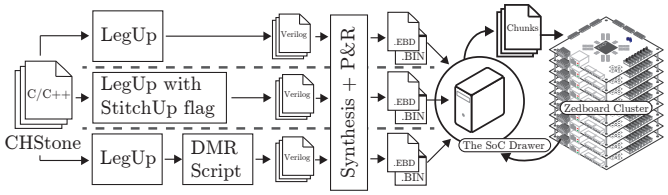

Figure 4: The experiment flow

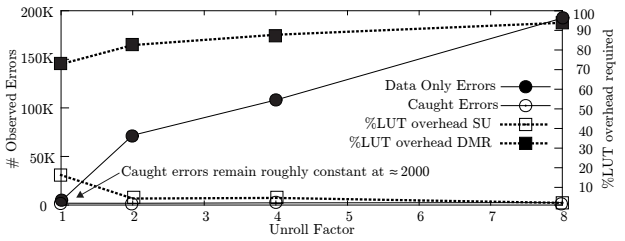

Figure 5: Data Only Errors (DOEs) and Caught Errors (CEs) for a dot product with increasing loop unroll factors with StitchUp along with %LUT overhead against DMR

4. EXPERIMENT SETUP

To test the error mitigation abilities of StitchUp generated circuits we have adopted the FPGA gold standard of hardware fault injection, where configuration bits on the FPGA device are flipped from their original state. Typical FPGA designs will require a large number of configuration bits often making full exhaustive testing infeasible, usually resulting in the adoption of random testing. However in our case we have developed an experiment platform allowing us to achieve full exhaustive fault injection in reasonable time, for example reducing the time to exhaustively test the smallest CHStone circuit dfmul from 4 days to 8 hours.

We make use of many low cost Xilinx FPGA SoC devices known as Zedboards all injecting faults on the same circuit in parallel. Each Zedboard runs a full Linux OS on its hardened ARM processor, and is arranged into a cluster managed by a single desktop.

To flip individual configuration bits a Xilinx Soft Error Mitigation (SEM) IP Core connected to the FPGA internal configuration port (ICAP) was used – targeting Xilinx devices was possible due to the latest release of LegUp (v4.0) which has the ability to generate generic Verilog circuits. For each experiment the SEM core and the test circuit were instantiated and connected to the ARM processing system via AXI interconnects.

Figure 4 shows an overview of the experiment setup. The top flow generates a standard LegUp circuit with no protection; the middle uses StitchUp to generate a control-flow structure protected circuit; and the bottom generates a full DMR protected circuit duplicating the original LegUp circuit and generating comparison logic on the control FSM state register;

Each different version is then passed through the Xilinx FPGA circuit tool flow to produce two files: an essential bits file (.EBD), and a configuration binary (.BIN). The essential bits file is a list of configuration memory bits that have any influence on our generated circuit and is used to reduce the number of injections required to fully test our design.[2] For each experiment the EBD files are divided up into smaller chunks, and each chunk is processed in parallel on a different Zedboard device.

Software running on the ARM of each Zedboard marshals each test by, injecting a fault, running the circuit, and storing the result. The injected fault is then repaired by re-injecting an error into the same configuration memory location (i.e. flipping the bit back to it's original state). Some faults may cause the circuit to never terminate, so if no response is seen within three orders of magnitude of its expected time a, timeout halts its execution.

[2]Essential bits do not contain BRAM data, since these are considered transient data not essential

The results are then analysed and each test outcome is classified into three distinct categories:

1. Execution Time Error **ETE** - This is where the execution of the circuit either took an incorrect number of cycles to complete or timed out. Errors of these type are control flow related, since any deviation from the correct execution path will cause an incorrect number of cycles.
2. Data-Flow Only Errors **DOE** - These are errors where the circuit has returned an incorrect data result but has executed in the correct number of cycles, such as error caused by faults in a non-control structure functional unit.
3. Caught Errors **CE** - These are errors in the state register that were detected by the protection method, which is either DMR or StitchUp.

For each test the proportion of ETE, DOE, and CE error results should sum to 100%. ETEs are the most critical since they are control related and may affect runtime guarantees or cause timeouts; DOEs are the next critical category since there is an error in data, but no damaging effect on control; and CEs are the least critical, where the circuit has failed but at least it has been detected.

5. EXPERIMENT RESULTS

To test the strengths and weaknesses of the StitchUp approach compared to DMR both techniques were evaluated on the CHStone benchmark suite. In order to explore the effects of varying control-to-data ratio, we also evaluate the example code from Listing 1 while varying the degree of loop unrolling to increase data parallelism. In all cases the resources in terms of LUTs, registers (REG), and DSPs were compared; and where the circuits could be implemented on Zedboard/ZC702 devices, power results were obtained, and exhaustive error injection was performed.

5.1 Unrolling Results

Figure 5 shows how the DOEs and CEs vary as the loop in Listing 1 is unrolled. Increasing the unroll factor reduces the control-to-data ratio since the number of basic blocks remains constant but the number of (data-flow) only functional units instantiated in parallel increases. This effect is observable in the results since the data only errors increase with the unroll factor but the number of caught errors remains constant. A similar trend can be observed for the percentage LUT

Table 1: Resource Usage Results - missing power results mean that the circuit could not be implemented on our power measurement platform (Xilinx ZC702)

	Original				StitchUp				DMR			
Bench	*LUT*	*REG*	*DSP*	*Pow mW*	*LUT*	*REG*	*DSP*	*Pow mW*	*LUT*	*REG*	*DSP*	*Pow mW*
aes	47230	28152	0	299.346	47539	28800	0	316.186	90467	53944	0	-
adpcm	21050	16752	168	133.962	29599	29484	178	156.903	38664	31077	348	-
blowfish	97002	45320	0	-	97440	46039	0	-	194598	88268	0	-
dfadd	4639	4310	0	50.0916	5562	5095	0	54.224	6394	5754	0	58.114
dfdiv	12144	13157	30	100.435	22254	23449	60	145.577	22811	23904	60	147.027
dfmul	3348	3912	16	47.024	4553	4845	32	54.128	5105	5397	32	57.521
dfsin	21343	20347	71	137.802	40928	38116	136	222.217	41136	38321	142	-
gsm	11953	9879	72	147.006	22049	17228	144	260.730	22399	17331	144	265.949
mips	8278	6367	4	89.326	15639	10231	8	132.409	14304	10351	8	134.316
motion	52665	26743	1	-	104840	50719	1	-	106986	51199	2	-
sha	9117	9287	3	-	9491	10188	6	-	16788	16189	6	-

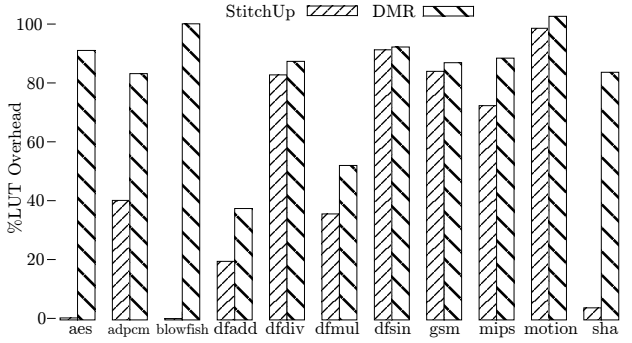

Figure 6: Comparison of % LUT overheads required for both StitchUp and DMR

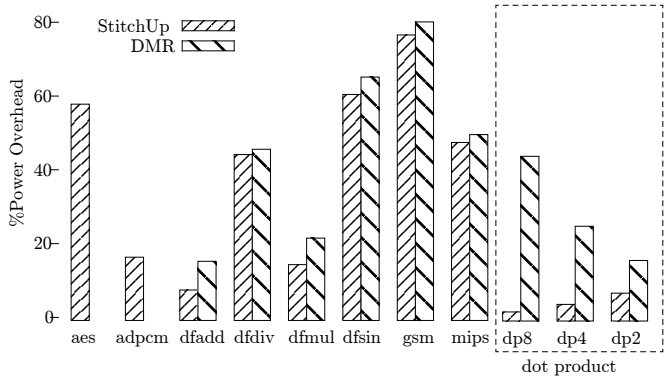

Figure 7: Comparison of % power overheads required for both StitchUp and DMR, where DMR power results for aes and adpcm could not be obtained because they are too large to implement on our power measurement platform.

overhead where the DMR overhead tends towards 100% as the unroll factor increases, while the StitchUp overhead tends towards 0%. This tells us that as more non-control-structure instructions are executed in parallel, the overhead required for StitchUp protection remains unaffected.

5.2 Resource Overheads

Table 1 shows the required resources and power to implement each of the CHStone benchmark circuits[3] with, no protection (original), StitchUp protection, and DMR protection. Figure 6 shows the relative percentage LUT overhead for the protection strategies on each circuit. In cases where the control-flow and data-flow are decoupled, little replication is required - this effect can be seen in the encryption benchmarks (aes, blowfish, sha) where replication can be as little as 1% of the original circuit. However there are other cases where protecting just the control flow structure results in DMR, such as dfsin or dfdiv. The amount of replication required by StitchUp is application dependant, however, from these results we can see that for a significant number of applications a low proportion of the original circuit needs to be replicated to protect the control-flow structure. [4]

5.3 Power Results

Figure 7 shows the percentage power overheads for all circuits which can be implemented on the ZC702. When compared to the resource overhead results there is a close correlation between resources required and power consumed. Along with some CHStone circuits the results for Listing 1 can be seen for an unroll factor 2 in dp2, 4 in dp4, and 8 in dp8. As the unroll factor increases and the ratio between control and data decreases the difference between the amount of power required by DMR and StitchUp also increases, with StitchUp consuming less power as more work is performed in parallel. This shows that not only does StitchUp have the potential to save area but that this also translates into a saving in power.

[3]with the exception of JPEG, which could not be routed onto our largest device because it is too large

[4]For the smaller circuits the overhead for DMR is often less than 100%, this is due to the overhead of instantiating the AXI infrastructure and SEM core.

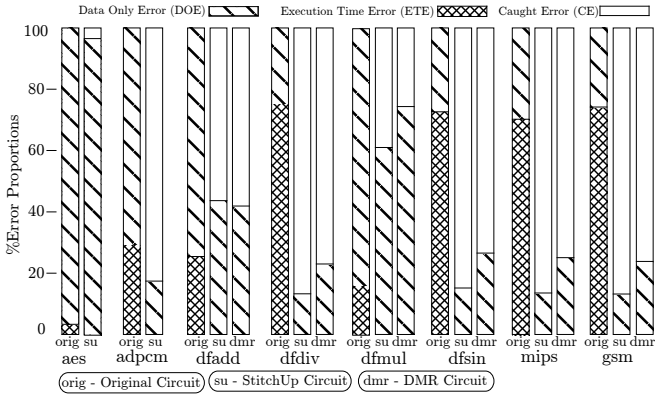

Figure 8: Error results for the CHStone benchmarks

5.4 Fault Injection Results

For every benchmarks possible to implement on the Zedboard device exhaustive error injection was performed with each error categorised into either ETE, DOE, or CE. Figure 8 shows the different proportions of these errors for each circuit, and in all cases we can see that the ETE have been removed for both StitchUp and DMR. Generally, the proportion of the circuit that is replicated corresponds to the proportion of detected faults: for example *aes* requires little replication but detects only ETEs; or *dfsin*, which requires a high proportion of replication but also detects a high proportion of errors. However there are some circuits between these extremes where the amount of replication returns a greater proportion than expected of errors caught, such as *adpcm* where a 40% replication results in 80% of errors caught. From these results not only can we see that StitchUp has the ability to remove all ETEs, but also can catch a high proportion of errors relative to the amount of the original circuit that was replicated.

6. RELATED WORK

The majority of reliable HLS approaches to date have come from the development of ASICs, and generally revolve around the idea of modifying the allocation, scheduling, and binding stages to include a library of components with different reliability criteria [14][6][7]. This library is then searched to determine the maximum possible reliability while meeting other constraints. We believe that our work is complimentary to these approaches, as we can protect the control-flow structure for a more reliable configuration of components.

More recent approaches have used static analysis techniques to analyse the vulnerabilities of registers based on their execution lifetimes[4]. While this approach is effective, it mainly targets ASIC designs, where register vulnerability is the largest concern. In FPGA devices this is not the case since the SRAM configuration memory is the most vulnerable region with register soft errors contributing very little.

Shastri et al. in [13] examined how HLS for FPGA reliability could be achieved by providing an allocation and scheduling algorithm which took into account the reuse and redundancy of certain shared resources. Again our approach is complimentary to this one, since it could be used to improve the reliability of the data path while we protect the control-flow structure.

7. CONCLUSION

Control-flow errors can be catastrophic, causing real-time deadlines to be missed, or worse non-termination. This work presents StitchUp a compiler flag for the Open Source HLS tool LegUp, which can automatically extract and protect the control-flow of an input program. To demonstrate this we applied it to the popular HLS benchmark suite CHStone and performed exhaustive hardware error injection on a selection of circuits. Our results show that StitchUp is capable of detecting all execution time errors while requiring a smaller fraction of the resources and power than DMR.

8. REFERENCES

[1] R. Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3):258–266, 2005.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[3] L. Chen, M. Ebrahimi, and M. B. Tahoori. Reliability-aware operation chaining in high level synthesis. In *Proceedings of European Test Symposium*, 2015.

[4] L. Chen and M. Tahoori. Reliability-aware register binding for control-flow intensive designs. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, pages 1–6. ACM, 2014.

[5] T. Feist. Vivado design suite. *White Paper*, 2012.

[6] M. Glaß, M. Lukasiewycz, T. Streichert, C. Haubelt, and J. Teich. Interactive presentation: Reliability-aware system synthesis. In *Proceedings of the conference on Design, automation and test in Europe*, pages 409–414. EDA Consortium, 2007.

[7] Y. Hara-Azumi and H. Tomiyama. Cost-efficient scheduling in high-level synthesis for soft-error vulnerability mitigation. In *Quality Electronic Design (ISQED), 2013 14th International Symposium on*, pages 502–507. IEEE, 2013.

[8] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: lessons learnt and future trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 99. ACM, 2013.

[9] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. *ACM SIGPLAN Notices*, 47(4):213–224, 2012.

[10] N. Nakka, K. Pattabiraman, and R. Iyer. Processor-level selective replication. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 544–553. IEEE, 2007.

[11] E. Normand. Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[12] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer. Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 760–769. IEEE, 2005.

[13] A. Shastri, G. Stitt, and E. Riccio. A scheduling and binding heuristic for high-level synthesis of fault-tolerant fpga applications. In *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE*, pages 202–209. IEEE, 2015.

[14] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, and Y. Xie. Reliability-centric high-level synthesis. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1258–1263. IEEE, 2005.

[15] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *In Proceedings of the Workshop on System Effects of Logic Soft Errors*. Citeseer, 2006.