PREPRINT: "S. Ananthanarayanan, S. Garg, and H. Patel, "Low Cost Permanent Fault Detection Using Ultra-Reduced Instruction Set Co-Processors ," in proceedings of IEEE Design Automation and Test in Europe (DATE), 2013, pp. 933-938."

re-encode any instruction of any ISA. At the same time, since the URISC core executes only one instruction, it consumes very little area even when compared to a standard in-order MIPS processor. The basic idea is illustrated in Figure 1.

II. Paper Contributions

In this paper, we propose a new online permanent fault *detection* technique based on the URISC co-processor architecture that has low area overhead, low detection latency and high fault coverage. This goal is accomplished via a combination of the following novel software and hardware techniques:

- Static insertion of URISC code to detect faults in both data flow and control flow instructions using redundant execution. Dedicated hardware registers are introduced in the URISC core to ensure that control flow faults that result in jumps to *arbitrary* locations in the code segment can be detected.
- To reduce the performance overhead of fault detection, only a *subset* of instructions in the program are checked for correctness on the URISC. Instructions are selected using a novel check window criterion to minimize the error detection latency.
- A polynomial time algorithm, based on a reduction to the weighted set cover problem, is proposed to pick the minimum number of instructions to check such that the check window criterion is met.
- URISC ISA extensions (referred to as URISC++) are proposed that further reduce the performance overhead of redundant execution on the checker core, while introducing minimal additional area overhead.

Our experimental results, based on a library of permanent faults injected directly in the RTL code of a TigerMIPS processor (main core) augmented with a URISC++ checker core illustrate the efficacy of the proposed techniques for practical, online, ISA level fault detection.

III. Related Work

Rajendiran et al. [10] first proposed using the URISC as a reliable co-processor for *fault recovery* assuming the list of faulty instructions is known a priori. However, they do not address the problem of online fault detection, which is the focus of this paper.

Gizopoulos et al. [11] provide a comprehensive survey of hardware techniques used to detect and recover from transient and permanent faults. Hardware techniques based on dual- and triple-modular redundancy [1], [2] introduce significant area overheads and cannot detect systematic faults that occur both in the main core and its redundant copy. The Diva processor [3] provides low overhead fault detection when used to detect faults in complex out-of-order processors, but has a significant area overhead if the main core is itself a simple in-order processor. Similar to [7] and [12], we focus on online permanent fault detection for small in-order processors.

Software redundancy techniques where the main core itself is used to perform redundant computations using duplicate or alternate instruction encodings in the native ISA [6], [5], [13], [14], [15] have also been proposed. However, a number of instructions in the ISA may not have alternate encodings, and therefore faults in these instructions cannot be checked.

```
1  subleqi   t,t,1          # t = 0
2  subleqi   r1,t,1         # t=-r1
3  subleqi   r2,t,1         # t=-r1-r2        } URISC Add
4  subleqi   d2,d2,1        # d2=0
5  subleqi   t,d2,1         # d2=-t; d2=r1+r2
6  add       d1,r1,r2       # Add executes on TigerMIPS
7            # Add check on URISC
8  bneqsubleq(d1,d2,ERR) # if (d1!=d2) goto ERR
9  sub       r3,r1,r2
10 ERR:      # Error recovery routine on URISC
```

Fig. 2: Checking data flow instructions. Instructions in the shaded boxes execute on the URISC .

Argus [7] and iSWAT [8] are based on the same guiding principle as the URISC checker core, i.e., fault detection using low-overhead hardware. However, Argus and iSWAT can only be used for fault detection, while the URISC co-processor, by virtue of being Turing complete, can be used both to detect faults (this work) and to recover from faults when they occur (as shown by [10]).

IV. URISC Background

For clarity of exposition, we briefly review the basic URISC co-processor architecture proposed by Rajendiran et al. [10]. The URISC implements the capability of executing a Turing-complete instruction called the subleq instruction. The semantics for any given subleq instruction subleq ra,rb,rc is given by the following steps: First, subtract the contents of ra from rb, and store the result in rb. Then, if the stored result in rb is less than or equal to zero, set the program counter to the contents of rc. In effect, the subleq instruction performs a subtraction, and based on the result of the subtraction jumps to the target address specified in the rc register. Lines 1-5 in Figure 2 illustrate how an add r1, r2, r3 instruction can be encoded using a sequence of subleq instructions. Since the subleq instruction is Turing complete, the URISC core can be used as a co-processor for a main core implementing, in theory, *any* ISA. In this paper, we use the architecture proposed by Rajendiran et al. [10] which uses the TigerMIPS [16] processor — a 5-stage pipelined implementation of the MIPS ISA — as the main core.

V. Proposed Methodology

We begin by discussing the scenario in which every MIPS instruction is checked by a corresponding sequence of subleq instructions on the URISC core. In this context, it is important to distinguish dataflow instructions from control flow instructions. We will discuss the two cases separately.

A. Checking Dataflow Instructions

The snippet of assembly code in Figure 2 illustrates how a dataflow instruction, for example, an add instruction is checked.

The URISC core first executes the instruction using a semantically equivalent sequence of subleq instructions, after

which the instruction itself executes on the TigerMIPS. Finally, the URISC compares its result with the result from the TigerMIPS execution. If the results match it proceeds to the next TigerMIPS instruction, else it jumps to a fault recovery routine. Note that the fault recovery routine is not in the scope of this work. In our implementation, the URISC simply halts program execution and signals a fault in the instruction for which the check did not succeed.

We make two observations based on the code snippet shown above. First, we observe that the URISC check can take multiple clock cycles, thus introducing a significant performance overhead if every TigerMIPS instruction is checked. Second, note that the URISC check routine can detect errors both in the ALU stage, *and* errors that arise from the add being incorrectly decoded in the TigerMIPS pipeline. Other low cost co-processor implementations, for example the Diva co-processor, do *not* have the latter capability.

```
1   subleqi   -1,flag,1      # URISC flag = 1
2   beq       r1,r2,LABEL    # BEQ executes
3   subleqbeq(r1,r2,ERR)     # Branch not taken check
4   subleqi   1,flag,1       # flag = 0
5   ...                      # Branch not taken insns
6 LABEL:
7   subleqbeq(r1,r2,ERR)     # Branch taken check
8   subleqi   1,flag,1       # flag = 0
9   ...                      # Branch taken insns
10  add       d1,r1,r2       # Incorrect jump target
11  subleqbeq(flag,1,ERR)    # Fault detected
12  ...                      # Add check on URISC
13 ERR:                      # Error recovery
```

Incorrect jump

Fig. 3: Checking control flow instructions. Instructions in the shaded boxes execute on the URISC .

B. Checking Control Flow Instructions

The same technique that was used to detect faults in dataflow instructions *cannot* be used for control flow instructions because a fault in a control flow instruction may cause it to jump to an *arbitrary* location in the code, evading the URISC check routine altogether. To address this issue, we propose a low cost solution by introducing a dedicated flag register on the URISC core. The flag register is set before the control flow instruction executes on the TigerMIPS to indicate that a control flow instruction is currently being checked. If the instruction executes successfully, the URISC check code at the correct target location unsets the flag register.

However, if the TigerMIPS control flow instruction incorrectly transfers execution to another section of code altogether, the value of the flag register can be checked to detect that the branch instruction in fact executed incorrectly. Thus before checking its own instruction, *every* URISC check sub-routine in the code first checks the value of the flag register and signals a branch fault if the flag is set. This is illustrated using the example in Figure 3, where the beq instruction being checked incorrectly jumps to an add instruction instead of its correct target address. The incorrect jump is detected by the new flag check instruction (line 11) that has been inserted in the beginning of the add check routine from Section V-A.

Note that, although not illustrated in Figure 3 for simplicity, the URISC check routine also saves the branch target address in a second dedicated flag register to compare with the target address of the branch instruction on the TigerMIPS. This guards against the pathological case in which the branch target is reached via an incorrect jump from another control flow instruction.

C. Check Window Based Instruction Sampling

If *every* instruction in the program is checked using the proposed techniques, permanent faults can be immediately detected. However, since each URISC check routines takes multiple cycles, this would incur a significant performance overhead. To address this issue, we propose checking only a *subset* of instructions in the code — because a permanent fault typically impacts multiple instances of an instruction after it first appears, sampling can significantly reduce the performance overhead of fault detection while still offering low fault detection latency and high error detection probability.

Selecting instructions based on random sampling [17] can result in some instruction types not being checked at all which reduces fault coverage. In addition, the distance between an unchecked instruction and the next checked instruction of the same type can be large, which results in greater fault detection latency. To address these issues, we propose a new systematic, profile-guided approach that guarantees the following property: for every unchecked instruction of a particular type, there exists an instruction of the same type that is checked for correctness on the URISC within the next W executed instructions. W is referred to as the *check window*, and provides designers with a powerful knob to trade-off performance and detection latency.

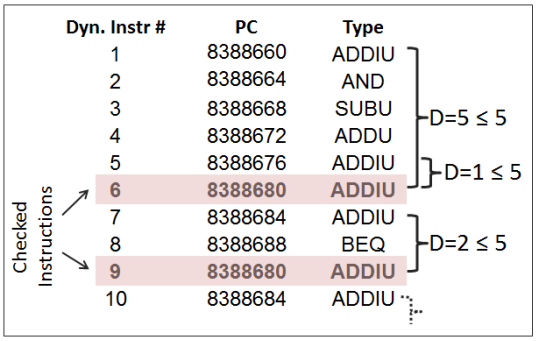

Fig. 4: A sequence of dynamic instructions. The shaded addiu instructions are checked on the URISC and $W = 5$ for this example.

Figure 4 illustrates the idea using a sequence of dynamic instructions taken from a run of the RSA benchmark for which W has been set to 5 and for simplicity, we focus only on the addiu instructions . It can be verified that every unchecked addiu instruction is followed by a checked addiu instruction within the next five instructions. Note also that since our technique *statically* inserts URISC checks in the source code, all dynamic instances corresponding to a unique

PC will either all be checked or all be unchecked. Given a dynamic instruction profile, we now propose an efficient algorithm to determine the *smallest* subset of instructions to check such that the check window constraint W is satisfied for every instruction type.

1) ILP Formulation: Given a sequence of N dynamic instructions, let PC_i represent the PC of the i^{th} dynamic instruction, t_i represent the instruction type and x_i ($x_i \in \{0,1\}\, \forall i \in [1,N]$) represent whether or not the instruction is checked on the URISC core. The minimum number of checked instructions that satisfy the check window condition can be determined by solving the following ILP:

$$\min \sum_{i=1}^{N} x_i$$

subject to:

$$\sum_{j:j\in[i,i+W],t(j)=t(i)} x_i \geq 1 \quad \forall i \in [1,N]$$

$$x_i = x_j \quad \forall i,j : PC_i = PC_j$$

Based on the solution to this ILP, the set C of all PCs that need to be checked can be determined as $C = \bigcup_{i:x(i)=1} PC_i$. In practice, however, the run-time of the ILP for even medium-sized benchmarks can be prohibitive.

2) Set Cover Based Solution: The ILP formulation in Section V-C1 can be cast as a weighted set cover problem, which is known to be NP-complete [18]. Given a universe $U = \{1,2,\ldots,N\}$, a set $S = \{S_1, S_2, \ldots, S_P\}$ of subsets of U, and weights $\{w_1, w_2, \ldots, w_P\}$ associated with each subset of S, the objective is to determine the subfamily $C \subseteq S$ with the smallest total weight such that the union of all the sets in C is U.

Lemma 1: The ILP formulation in Section V-C1 can be reduced to the weighted set cover problem.

Proof: The elements of the weighted set cover problem are constructed as follows:

1) The universe U of elements consists of the N dynamic instructions that the program executes.
2) Each subset $S_i (i \in [1,P])$ of the set S corresponds to one of the P PCs (static instructions) in the source code. The elements of S_i consist of all dynamic instructions that are checked if PC i is checked *and* all dynamic instructions that no longer need to be checked because PC i is being checked. Formally:
$$S_i = \{j,k : PC_j = i; k \in [j-W, j-1] \cap t_k = t_j\}$$
3) The weight w_i of S_i is equal to the number of dynamic instructions corresponding to PC i, i.e., $w_i = |\{j : PC_j = i\}|$.

The optimal solution of this instance of weighted set cover corresponds the the optimal solution of the ILP in in Section V-C1.

■

The best known polynomial time algorithm for the weighted set cover problem is a *greedy* algorithm that, in each iteration, picks the subset (PC) that covers the most number of elements (dynamic instructions) not already covered, normalized by the weight of the subset. Algorithm 1 formally describes the proposed greedy set cover based instruction selection technique.

Algorithm 1 Greedy Set Cover Based Instruction Selection

$U \leftarrow \{1,2,\ldots,N\}$
$S_i \leftarrow \{j,k : PC_j = i; k \in [j-W,j) \cap t_k = t_j\}\, i \in [1,P]$
$w_i \leftarrow |\{j : PC_j = i\}|$
$C \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
  $q \leftarrow arg\max_i \frac{|S_i \cap U - C|}{w_i}$
  $C \leftarrow C \cup q$
  $U \leftarrow U - S_q$
end while
return C

We have modified the back-end of the LLVM compiler to insert the appropriate URISC check routines (as described in Section V-B and Section V-A) that check the PCs selected by Algorithm 1. The resulting assembly code consists of a mix of MIPS and subleq instructions that are then pushed through the TigerMIPS GCC assembler to generate execution binaries.

D. URISC ISA Extensions: URISC++

Every check routine on the URISC compares the result of executing an instruction on the TigerMIPS core and the URISC core, which can take between 5 to 11 cycles if only subleq instructions are used (the execution latency is data dependent). Keeping in mind that operand comparison is common to all check routines, we propose adding the uriscbeq instruction (URISC equivalent of the beq instruction) to the URISC ISA. This reduces the latency of an operand comparison to just one cycle and, at the same time, the hardware overhead of adding this instruction is minimal since the URISC processor already has an in-built comparator to execute the subleq instruction which is re-used for the uriscbeq .

Furthermore, to decrease the performance overhead of common data-flow instructions such as the and, or, xor, mult and div MIPS instructions, the uriscand (URISC equivalent of and) and uriscsrri (URISC shift right by one) instructions are added. As with the uriscbeq instruction, these additional instructions significantly reduce the performance overhead of permanent fault detection but introduce very little new hardware. The uriscand requires 32 additional *and* gates in the execute stage of the URISC processor. The uriscsrri does not require any additional logic gates in the execute stage since it only shifts right by a *constant* value.

E. Tool Flow

Starting with C/C++ source code, we first use a source code transformation, in this case loop unrolling, to create three versions of every benchmark: (i) **original**: unmodified source code; (ii) **partially unrolled**: only the inner most loops are unrolled; and (iii) **fully unrolled**: every loop is unrolled. Loop unrolling was first proposed by Hong et al. [19] in the context of permanent fault detection using software-only techniques. The idea is that loop unrolling results in fewer dynamic instructions per static PC and offers greater opportunities for static instrumentation of the source code. In our experimental results

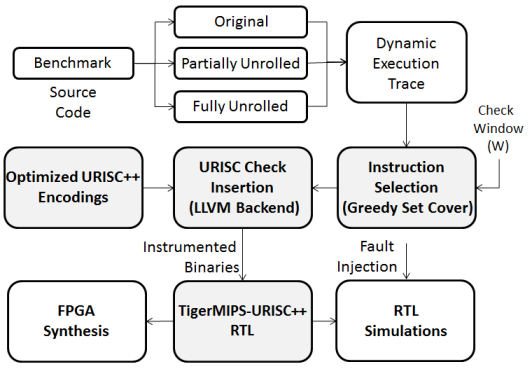

Fig. 5: Tool flow developed to obtain experimental results.

we observe that the two unrolled versions of each benchmark have smaller performance overhead than the original version for the *same* window size.

Representative inputs are used to generate a dynamic instruction trace for each binary and PCs are selected for checking based on the proposed greedy set cover algorithm. Static code to check the selected PCs is inserted using optimized URISC++ encodings and the techniques proposed in Section V-A and V-B to generate combined MIPS-URISC++ binaries. These techniques are all directly integrated in the back-end of the LLVM compiler tool-chain. The binaries are simulated on the TigerMIPS-URISC++ RTL, while FPGA synthesis of the RTL provides estimates of the area overhead of the proposed architecture.

VI. Experimental Results

Fault Name	Block	Fault Impact	Data Dependent
Add Fault	Decode	addiu decoded as sub	Yes
Branch Taken Fault	Branch	Branch always not taken	No
Branch Offset Fault	Branch	Stuck-at in branch offset	Yes
SLL Fault	ALU	Stuck-at in sll output	Yes
Mult Fault	ALU	Bit flip in mult output	Yes

TABLE I: Description of the faults injected in the main core.

As benchmarks, we use four applications from the Mibench benchmark suite [20]: Bubble Sort, String Search, RSA and Dijikstra. Permanent faults are directly injected in to the RTL code of the TigerMIPS processor as either stuck-at faults or bit-flips in architectural registers and internal wires. Faults were injected at random times during program execution. Table I provides a detailed description of the fault library that we experimented with. Over all benchmarks, check window sizes, fault types and fault injection times, we conducted more than 900 fault injection experiments.

A. FPGA Synthesis Results

Table II shows the results obtained from synthesizing the TigerMIPS, TigerMIPS-URISC and TigerMIPS-URISC++ architectures on an Altera Cyclone II FPGA. The TigerMIPS-URISC architecture has 21.3% more logic elements and 14.3% more registers than TigerMIPS alone. Moving from the URISC core to the URISC++ core adds *only* 62 more

	Logic Elements	Registers
TigerMIPS	12379	4578
with URISC	15019 (21.3%)	5232 (14.3%)
with URISC++	15081 (21.8%)	5233 (14.3%)

TABLE II: FPGA synthesis results URISC and URISC++ architectures.

logic elements and 1 additional register to the design. In fact, Although URISC++ adds *only* 0.5% more area overhead compared to URISC, TigerMIPS+URISC++ binaries have a 2.1× *lower* execution latency compared to TigerMIPS-URISC binaries. The rest of the experimental results are presented on the TigerMIPS-URISC++ only. We note that the hardware overhead of TigerMIPS-URISC++ is comparable to the low-cost Argus co-processor, for which a 17% area overhead was reported [7].

B. Permanent Fault Detection

We now present results on permanent fault detection latency and probability using URISC++.

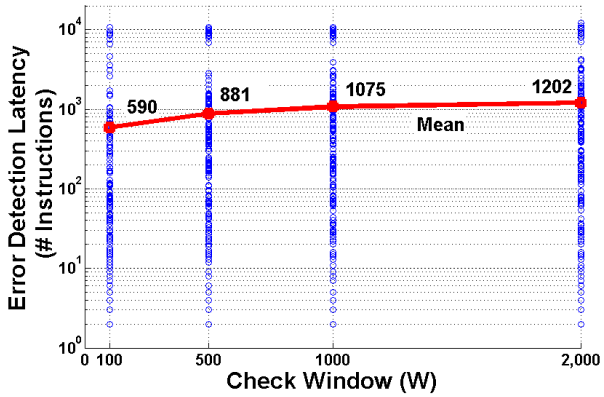

Fig. 6: Scatter plot of fault detection latency versus window size. Also shown is the mean fault detection latency.

1) Fault Detection Latency: Figure 6 illustrates that, as expected, the fault detection latency increases with increasing check window sizes. The average fault detection latency for the largest window size, $W = 2000$, is 1200 MIPS instructions.

As a basis for comparison, iSWAT [8] detects only 50.7% of injected permanent faults within 10,000 instructions. In contrast, URISC++ performs much better and detects more than 95% of injected faults within 10,000 instructions. Argus [7] does not explicitly report fault detection latency numbers. Finally, we also implemented the random sampling approach proposed by [17] and found that random sampling results in an 8× *increase* in average error detection latency compared to the proposed check window based sampling approach.

2) Fault Detection Probability: Averaged across all fault injection experiments and all window sizes, 95.63% of injected faults are correctly detected by the proposed technique, while 2.92% result in time-outs and 1.46% result in crashes. Importantly, none of our experiments resulted in *silent data*

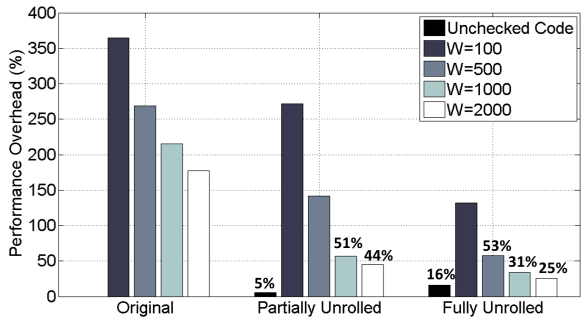

Fig. 7: Performance overhead results averaged over benchmarks.

corruptions (SDC). The fault detection rate for a small window size of $W = 100$ is 98.03%, while for the largest window size of $W = 2000$ the fault detection rate is 93.63%. In comparison, the reported fault detection probabilities for iSWAT [8] and Argus [7] are 97.2% and 98%, respectively. Additionally, iSWAT and Argus report a 0.3% and 0.46% SDC rate.

C. Performance Overhead

Figure 7 shows the average performance overhead of the proposed technique across all benchmarks and fault injection experiments for different window sizes and for different levels of loop unrolling. Recall that for a given window size W, loop unrolling allows fewer instructions to be checked and thus reduces the performance overhead of fault detection [19].

For the largest window size, $W = 2000$, partial and full unrolling results in a performance overhead of 44% and 25%, respectively, over baseline execution. The reported performance overheads for online, permanent fault detection reported in literature vary from 4% for Argus, 25% for iSWAT, 30% for RMT [2], and between $50\% - 100\%$ for software redundancy techniques like [14].

D. Critical Analysis of Results.

We have shown, based on extensive permanent fault injection experiments in RTL, that the proposed techniques are competitive with the state-of-the-art on all important metrics relevant to online permanent fault detection. In spirit, the closest work to ours is Argus, which also proposes adding low complexity hardware to detect faults in simple, in-order main cores. While the performance overhead of URISC++ is greater than that of Argus, we note that as opposed to Argus, the URISC++ co-processor can be used for *both* fault detection *and* fault recovery once the fault has been detected.

VII. Conclusion

In this paper, we have presented a new, low hardware overhead permanent fault detection architecture for high defect rate technologies. The proposed architecture is based on the use of a URISC checker core that, in theory, only executes one Turing complete instruction and can therefore be used to emulate and check the correctness of any instruction that executes on the main core. In support of this idea, a number of novel techniques, both hardware and software, are proposed that enable the use of the URISC co-processor for online permanent fault detection with high fault coverage, bounded fault detection latency and minimal performance impact. Extensive experimental results illustrate the promise of the proposed approach as a solution that enables *both* fault detection (this work) and subsequent fault recovery using the same low area overhead hardware. As future work, we are looking at parallel URISC execution to further reduce performance overhead.

References

[1] N. Aggarwal *et al.*, "Configurable isolation: building high availability systems with commodity multi-core processors," *SIGARCH Computer Architecure News*, vol. 35, no. 2, pp. 470–481, 2007.

[2] S. Mukherjee *et al.*, "Detailed design and evaluation of redundant multi-threading alternatives," in *Proceedings of the 29th international symposium on computer architecture*, 2002, pp. 99–110.

[3] T. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd IEEE international symposium on microarchitecture*, 1999, pp. 196–207.

[4] T. Lovric, "Systematic and design diversitysoftware techniques for hardware fault detection," *Proceedings of the 1st European dependable computing conference*, pp. 307–326, 1994.

[5] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, "Accelerating microprocessor silicon validation by exposing isa diversity," in *Proceedings of the 44th IEEE international symposium on microarchitecture*, 2011, pp. 386–397.

[6] A. Meixner and D. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *Proceedings of IEEE international conference on dependable systems and networks*, 2008, pp. 80–89.

[7] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proceedings of the 40th IEEE international symposium on microarchitecture*, 2007, pp. 210–222.

[8] S. Sahoo *et al.*, "Using likely program invariants to detect hardware errors," in *Proceedings of the IEEE International Conference on dependable systems and networks*. IEEE, 2008.

[9] F. Mavaddat and B. Parhami, "URISC: the ultimate reduced instruction set computer," *International Journal of Electrical Engineering Education*, 1988.

[10] A. Rajendiran *et al.*, "Reliable computing with ultra-reduced instruction set co-processors," in *Proceedings of the 49th annual design automation conference*, 2012.

[11] D. Gizopoulos *et al.*, "Architectures for online error detection and recovery in multicore processors," in *Proceedings of IEEE design, automation and test in Europe (DATE)*, 2011, pp. 1–6.

[12] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[13] C. Bolchini, "A software methodology for detecting hardware faults in vliw data paths," *IEEE Transactions on Reliability*, 2003.

[14] N. Oh *et al.*, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, 2002.

[15] S. Rehman *et al.*, "Instruction scheduling for reliability-aware compilation," in *Proceedings of the design automation conference*, 2012.

[16] S. Moore and G. Chadwick, http://www.cl.cam.ac.uk/teaching/.

[17] S. Nomura *et al.*, "Sampling+ dmr: practical and low-overhead permanent fault detection," in *Proceedings of the 38th annual international symposium on computer architecture*, 2011.

[18] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithms."

[19] T. Hong *et al.*, "Qed: Quick error detection tests for effective post-silicon validation," in *Proceedings of the IEEE international test conference*, 2010.

[20] M. Guthaus *et al.*, http://www.eecs.umich.edu/mibench/.