Low Cost Permanent Fault Detection Using Ultra-Reduced Instruction Set Co-Processors

Sundaram Ananthanarayanan*, Siddharth Garg, and Hiren D. Patel
* Stanford University, University of Waterloo
sundaram.ananthanarayanan@stanford.edu, {s6garg,hdpatel}@uwaterloo.ca

I. INTRODUCTION

Technology scaling results in an increased rate of permanent hardware faults that arise because of number of failure mechanisms including bias temperature instability and electro-migration. If left unaddressed, in-field permanent faults can lead to a significant degradation in the reliability and dependability of next-generation computing systems. Thus, developing on-chip techniques for online detection of in-field permanent faults and subsequent fault recovery has become a critical design challenge.

The features desired from a practical online fault detection mechanism are: (i) low area and performance overhead, (ii) high fault coverage, (iii) low error detection latency, and (iv) support for fault diagnosis and recovery. However, a majority of the previously proposed micro-architectural techniques for online permanent fault detection do not simultaneously achieve all these objectives.

First, techniques in which a redundant copy of the main processor is used as a checker core to verify the execution on the main core suffer from $2 \times$ or more area overhead [1], [2]. We note that even though the DIVA architecture [3] uses a simple in-order processor as a checker core for a more complex out-of-order processor, it would have a $2 \times$ or more overhead if the main core is itself a simple in-order processor, which is the target of this paper. In addition to the high area overhead, using a redundant copy of the main core as a checker core suffers from a “who checks the checker” issue for permanent faults that are systematic in nature [4], i.e., the same fault that occurs in the main core is likely to occur in its redundant copy.

Second, software-only techniques have been proposed that execute alternative encodings of an instruction on the main core itself instead of on a redundant core [5], [6]. While very effective, these techniques cannot guarantee 100% fault coverage since there exist instructions in the ISA that have no or only partial equivalence, as shown by Foutris et al. [5]. Thus, a fault in any of these critical instructions cannot be checked by redundant execution in software, which compromises fault coverage.

The recently proposed Argus [7] and iSWAT [8] architectures achieve the first three objectives by introducing low overhead hardware that either verifies dataflow and control flow invariants during program execution (Argus) or detects program anomalies (iSWAT). However, the Argus and iSWAT hardware checkers can only be used for fault detection but not for fault recovery. In contrast to Argus and iSWAT, the proposed technique can be used for both fault detection and subsequent fault recovery. At the same time, the proposed technique is competitive with Argus and iSWAT in terms of area overhead, fault detection latency and fault coverage.

In this paper, we propose a new ISA level on-line permanent fault detection technique that has a low area overhead, low error detection latency and high fault coverage by making use of an ultra-reduced instruction set co-processor (URISC) [9] as a redundant checker core. A URISC core has only one instruction in its ISA (the subleq instruction) but this instruction is Turing complete and can therefore be used to 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 with a simple example that shows how a permanent fault can be detected in a dataflow instruction using redundant execution on the URISC followed by a check routine that also executes on the URISC to guarantee correctness.

The use of a URISC core as a reliable co-processor was first proposed in a recent paper by Rajendiran et al. [10] in the context of fault recovery. In their paper, the authors assume that the list of faulty instructions is known a priori, and simply execute these faulty instructions on the URISC co-processor instead of the main processor. However, the authors do not address how the list of faulty instructions can be determined at run-time and leave this as an open problem. In this paper, we address this challenge.

II. 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 [11] processor — a 5-stage pipelined implementation of the MIPS ISA — as the main core. Because of its simplicity, the URISC core itself requires only two pipeline stages and executes synchronously with the TigerMIPS core, i.e., at the same clock frequency. Instructions on the TigerMIPS-URISC architecture execute sequentially either on the main core or the URISC co-processor depending on whether they are MIPS or URISC instructions. Note that, in the context of this paper, we make two assumptions. First, we address permanent faults only in the core logic, and assume that the memory sub-system...
The incorrect jump is detected by the new flag check instruction (line 11) using the example in Figure 3, where the register can be checked to detect that the branch instruction in fact flow instruction is currently being checked. If the instruction executes the code, evading the URISC check routine altogether. To address this

B. Checking Control Flow Instructions

that has been inserted in the beginning of the add check routine from Section III-A.

C. Check Window Based Instruction Sampling

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

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

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

is fault-free. Second, like [3], [7], [8], we assume that the low-overhead URISC co-processor can be hardened against faults using conservative design techniques and thus always operates correctly.

III. FAULT DETECTION WITH URISC

We begin by discussing the scenario in which every MIPS instruction is checked by a corresponding sequence of subeq 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 sequence of operations in the code above is critical — the URISC core first executes the instruction using a semantically equivalent sequence of subeq instructions, after which the instruction itself executes on the TigerMIPS. Executing the instruction on the TigerMIPS before the URISC results in a false negative if one of the operand registers is the same as the destination register.

After the instruction executes on both cores, 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.

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)
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:

\[
\min \sum_{i=1}^{N} x_i \\
\text{subject to:} \sum_{j: j \in [i+1+N], t(j)=t(i)} x_j \geq 1 \quad \forall i \in [1, N]
\]

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

2) Set Cover Based Greedy Solution: The ILP formulation in Section III-C1 can be cast as a weighted set cover problem, which is known to be NP-complete [13]. 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 \( U \).

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.

We have modified the back-end of the LLVM compiler to insert the appropriate URISC check routines (as described in Section III-B and Section III-A) that check the selected PCs. The resulting assembly code consists of a mix of MIPS and addiu 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 core results in the check execution of 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 uriscorrid (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 URISC ISA enhanced with these three additional instructions is referred to as URISC++. We note that neither of the three new instructions are, by themselves or in conjunction, Turing complete — the subleq instruction is still critical in providing full functionality and fault coverage.

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

IV. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Fault Name</th>
<th>Block</th>
<th>Fault Impact</th>
<th>Data Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Fault</td>
<td>Decode</td>
<td>addiu decoded as sub</td>
<td>Yes</td>
</tr>
<tr>
<td>Branch Taken Fault</td>
<td>Branch</td>
<td>Branch always not taken</td>
<td>No</td>
</tr>
<tr>
<td>Branch Offset Fault</td>
<td>Branch</td>
<td>Stuck-at in branch offset</td>
<td>Yes</td>
</tr>
<tr>
<td>SLL Fault</td>
<td>ALU</td>
<td>Stuck-at in sll output</td>
<td>Yes</td>
</tr>
<tr>
<td>Mult Fault</td>
<td>ALU</td>
<td>Bit flip in mult output</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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

As benchmarks, we use four applications from the Mibench benchmark suite [15]: Bubble Sort, String Search, RSA and Dijkstra. 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>
<thead>
<tr>
<th></th>
<th>Logic Elements</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TigerMIPS</td>
<td>12579</td>
<td>4578</td>
</tr>
<tr>
<td>with URISC</td>
<td>15019 (21.3%)</td>
<td>5232 (14.3%)</td>
</tr>
<tr>
<td>with URISC++</td>
<td>15081 (21.8%)</td>
<td>5233 (14.3%)</td>
</tr>
</tbody>
</table>

TABLE II: FPGA synthesis results for the TigerMIPS, TigerMIPS-URISC and TigerMIPS-URISC++ architectures.
While the performance overhead of URISC++ is greater than that of low complexity hardware to detect faults in simple, in-order main cores, redundancy techniques like [16]. The reported performance overheads for online, per-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 corruptions (SDC), i.e., cases when the program output was incorrectly modified without either fault detection or a time-out/crash. 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%. We note that in theory, the proposed technique has a 100% fault detection rate with a window size of \( W = 1 \), since each instruction will be checked for correctness on the URISC.

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% percent probability of SDCs. In our experiments, we did not observe any SDCs (lower rate of SDCs is better).

C. Performance Overhead

Figure 6 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 [14].

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 [16].

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 by virtue of being Turing complete, the URISC++ co-processor can be used for both fault detection and fault recovery once the fault has been detected. The redundant hardware in Argus does not offer this distinguishing feature.

V. 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. 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.

REFERENCES