PREPRINT: "A. Rajendiran, S. Ananthanarayanan, H. Patel, M. V. Tripunitara, and S. Garg, "Reliable Computing with Ultra-Reduced Instruction-Set Co-processors," in proceedings of IEEE Design Automation Conference (DAC), New York, NY, USA, 2012, pp. 697-702."

Reliable Computing with Ultra-Reduced Instruction Set Co-processors

Dan Wang
dan.wang@uwaterloo.ca

Aravindkumar Rajendiran
a2rajend@uwaterloo.ca

Sundaram Ananthanarayanan
sanath2@stanford.edu

Hiren Patel
hiren.patel@uwaterloo.ca

Mahesh V. Tripunitara
tripunit@uwaterloo.ca

Siddharth Garg
siddharth.garg@uwaterloo.ca

ABSTRACT

This work presents a method to reliably perform computations in the presence of hard faults arising from aggressive technology scaling, and design defects from human error. Our method is based on the observation that a single Turing-complete instruction can mirror the semantics of any other instruction. One such instruction is the subleq instruction, which has been used for instructional purposes in the past. The scope of using such an instruction is far greater than that of instructional purposes, and thus, we present its applicability to fault tolerance. In particular, we extend a MIPS processor with a co-processor (called the ultra-reduced instruction set co-processor – URISC) that implements the subleq instruction. We use the URISC to execute sequences of subleq to mimic the semantics of instructions that are known to be faulty on the MIPS core after testing. Our LLVM compiler back-end generates the sequence of subleq for instructions marked as faulty. This presents a hardware-software approach to fault recovery. We experimentally evaluate the following: impact of single-upset faults on the instructions that are rendered faulty, the area overhead of the URISC, and the performance overhead of using URISC.

Keywords

Microprocessor reliability, Hard faults, Turing-complete ISA.

1. INTRODUCTION

Aggressive technology scaling in advanced CMOS manufacturing technology provides designers with an abundance of transistors on a single die. There is a consensus in the computing community that one way to obtain performance from these transistors is to design highly parallel multicore processors. However, an impediment to the performance benefits that accrue from parallelism is its increased susceptibility to hard faults caused by aggressive technology scaling. Such faults reduce the yield because the faulty core must either be discarded or disabled. Therefore, one of the major challenges in multicore designs is the design of cost-effective dependability [1] solutions that enable cores to continue operating correctly even under the presence of one or more hard faults.

Research efforts to address this challenge make use of either hardware redundancy [2, 3, 4] or software re-compilation [5, 6]. Hardware redundancy techniques migrate execution to spare cores when a fault is detected in the main core. Techniques in which the spare core (or cores) is identical to the main core incur a high area overhead, and provide no protection against systematic faults, or even design bugs. DIVA [7], on the other hand, extends an out-of-order core with a low-cost in-order core. However, DIVA cannot recover from faults in the decode stage of the main core. Furthermore, if the main core is itself an in-order core, the DIVA approach would have a $2\times$ hardware overhead. A purely software-based approach, called detouring [5], recompiles faulty instructions using instructions that are known to operate correctly. Recent work has shown that up to 26% of the instructions in x86 ISA have no or only partial equivalence [8]. Therefore, software-only approaches cannot guarantee 100% fault coverage. Finally, techniques such as Necromancer [9] and Multiplexed Redundant Execution [10] exploit faulty cores to enhance the performance of fault-free cores by forwarding architectural “hints" to the latter, but do not use the faulty cores for actual execution.

In this work, we argue for an approach that combines software and hardware techniques for cost-effective reliable computing. In particular, this work focuses primarily on recovering from hard faults. The novelty of our approach is based on the observation that we can represent the semantics of any faulty instruction with a single non-faulty Turing-complete instruction called subleq [11]. A co-processor that implements this single instruction ISA is referred to as an Ultra-Reduced Instruction Set Computer (URISC); hereafter referred to as URISC. We use URISC as a redundant core to execute sequences of subleq instructions that are semantically equivalent to any instructions that execute erroneously on the main core. This approach has two benefits: 1) the hardware overhead of the URISC is low because it implements primarily the subleq instruction, and 2) in theory, the URISC provides 100% fault coverage for a main core that executes any ISA.

Our fault model assumes that the subset of faulty instructions resulting from hard faults is known at compile time[1]. For each faulty instruction, we produce a sequence of subleq instructions that are semantically equivalent to that instruction. This allows us to recompile any program given its subset of faulty instructions to use the URISC for just those faulty instructions.

As a prototype to evaluate the proposed techniques, we extend

[1]Our recent work shows that URISC can be used to populate this list of faulty instructions [12].

Equivalent sequence for add			Equivalent sequence for beqz		
L0:	subleq	r4, r4, L1	L0:	subleq	r1, Z, L2
L1:	subleq	r2, r4, L2	L1:	subleq	Z, Z, L4
L2:	subleq	r3, r4, L3	L2:	subleq	Z, Z, L3
L3:	subleq	r1, r1, L4	L3:	subleq	Z, r1, r2
L4:	subleq	r4, r1, L5	L4:	...	
L5:	...				

(a) Example sequences of subleq for add and beqz MIPS instructions.

Inst.	I_{31-26}	I_{25-21}	I_{20-16}	I_{15-11}	I_{10-6}	I_{5-0}
subleq	010 000	11000	Rs	Rd	Rt	00 0000
subleqi	010 000	11100	Rs	Rd	Immediate	
mfu	010 000	10000	Rd	Rs	000 0000 0000	
mtu	010 000	10100	Rs	Rd	000 0000 0000	
mtui	010 000	10101	Rd	Immediate		
mtuiu	010 000	10110	Rd	Immediate		

(b) ISA extensions for the MIPS-URISC processor.

Figure 1: Example using subleq, and ISA extension encoding in MIPS-URISC.

the TigerMIPS [13] with the URISC. MIPS-URISC uses a two-stage pipeline for the URISC, and it separates its decode and execute stages from that of the TigerMIPS. At any time, either the TigerMIPS pipeline or the URISC pipeline is operational while the other is stalled. This augmented design is referred to as the MIPS-URISC. We modify LLVM's MIPS back-end to automatically generate programs that consist of a mix of MIPS and URISC instructions. We experimentally evaluate the following aspects of the proposed work: 1) we discover the impact of single-bit stuck-at-faults on the instructions that are rendered faulty, 2) we evaluate the area overhead of the URISC, and 3) we evaluate the performance overhead of the URISC for several benchmarks. Note that the MIPS-URISC only introduces a performance overhead only when one or more MIPS instructions are faulty. Otherwise, the URISC is not used at all.

2. SEMANTICS OF THE SUBLEQ INSTRUCTION

The original subleq instruction [11] was a CISC instruction. However, we adopt a modified RISC version of the instruction that only reads from, and writes to the register file. Memory accesses are enabled through the MIPS core, and assumed to operate correctly.

The semantics of the subleq instruction subleq ra,rb,rc is as follows: subtract the contents of ra from rb, and store the result in rb, and if the stored result in rb is less than or equal to zero, then set the program counter (PC) to the contents of rc. Otherwise, the PC increments to the next instruction. In Figure 1a, we illustrate how subleq can be used to emulate other instructions in the MIPS ISA using an example of the add r1, r2, r3, and the beqz r1, r2 instructions.

For the add, we use r4 as a temporary register to perform the addition of the negative values of registers r2 and r3. Then, we perform another subtraction to negate the negative addition to produce the correct output. Notice that the third operand we specify for subleq is a label denoting the target address when the branch condition is satisfied, which we use to jump to the next instruction in program order. Similarly, the equivalent sequence of subleq instructions for the beqz instruction, where r1 holds the value to be checked and r2 holds the target address. Note that Z is a register that holds the zero value.

3. MIPS-URISC ARCHITECTURE

The MIPS-URISC combines a five-stage pipelined implementation of the MIPS ISA known as the TigerMIPS [13] with the URISC. In doing this, we extend the MIPS ISA with instructions that facilitate transfer of register contents between the TigerMIPS and URISC register files in addition to the subleq instruction.

3.1 ISA Extensions to support URISC

We extend the MIPS ISA with the instructions shown in Figure 1b. These instructions provide a programming interface for the URISC. We use co-processor instructions from the MIPS ISA to encode URISC instructions. Note that Rs, and Rt denote source registers, and Rd denotes the destination register, and immediate identifies an immediate value operand.

The URISC supports three types of instructions. The first type constitutes the subleq instructions. There are two variants: one where the third operand comes from a register, and one where the third operand is an immediate value. We rely on the load and store operations of the MIPS ISA to retrieve contents from the memory to the MIPS registers. However, it is possible to extend the URISC to have its own fault-free load and store instructions. The contents of these MIPS registers are transferred to the register file of the URISC by using the mtu (move to URISC) instructions. We provide the immediate variant of the move to URISC instruction (mtui) to allow directly loading the register with a sign-extended eleven-bit immediate value, and its counter-part mtuiu for its zero-extended variant. The mfu (move from URISC) instruction transfers register contents from the URISC to the TigerMIPS.

3.2 MIPS-URISC Hardware Design

The micro-architecture of the MIPS-URISC is shown in Figure 2, and it consists of the TigerMIPS processor integrated with an implementation of the URISC. The TigerMIPS implements a conventional five stage pipeline executing the MIPS ISA [13] — i.e., fetch (F), decode (D), execute (E), memory (M) and writeback (W).

3.2.1 URISC Design and Integration

Due to the straight forward semantics of the subleq instruction, the URISC implementation requires only two pipeline stages: the fetch (UF), and decode/execute/write-back (UDEW) stages. The UF stage in URISC is shared with the fetch stage in TigerMIPS. In this stage, we introduce a small pre-decoder to identify any instruction that requires the URISC. Along with the fetched instruction, we store the result of the pre-decoder in the respective pipeline registers that are input to the decode stages (F/D and UF/UDEW).

An instruction sequence for the MIPS-URISC consists of a mixture of MIPS and subleq instructions. For each instruction, the pre-decoder in the UF stage identifies whether the fetched instruction is a URISC instruction or a MIPS instruction. If it is the former, then we insert no-operations (nops) in the TigerMIPS pipeline. However, if it is the latter, then nops are inserted into the URISC pipeline. When a URISC instruction follows a MIPS instruction, we ensure that the URISC instruction stalls until the write-back of the MIPS instructions. Similarly, the TigerMIPS instruction stall until any preceding URISC instruction completes its execution in the URISC pipeline.

The UDEW stage uses of a small register file with ten registers, a subtractor and a comparator. This stage decodes the incoming instruction, reads operands from the register file, performs a subtract

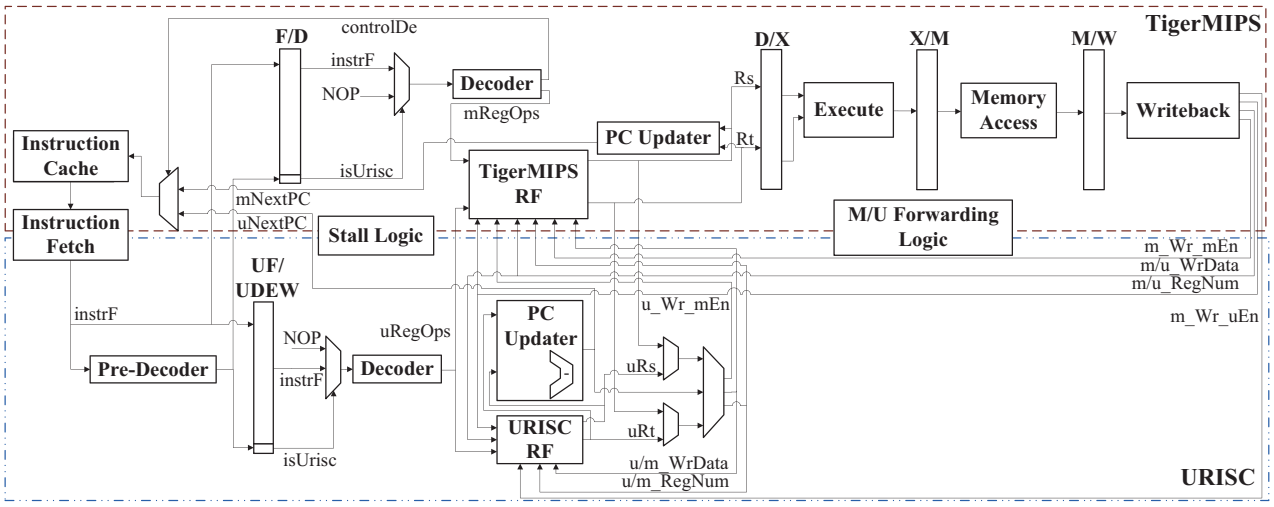


Figure 2: Hardware design for MIPS-URISC.

and compare-with-less-than-zero operation, writes back to the register file, and updates the next PC address. When the execution of a subleq instruction satisfies the less than equal to zero operation, the PC receives the target address from the third operand. This is when the URISC incurs a one cycle penalty. Note that our implementation of the URISC does not have direct access to any of the memories. Memory operations are carried out using the TigerMIPS, and mtu instructions are used to transfer from the TigerMIPS register file to the URISC register file. Finally, the URISC consists of a forwarding unit such that read-after-write (RAW) dependencies between any URISC instructions are avoided. Since the URISC cannot directly access memories, there are also no stalls resulting from cache misses, and therefore does not require any stall logic for in-flight instructions.

3.3 MIPS-URISC Compiler Toolchain

We provide a compiler toolchain based on the LLVM [14] that generates assembly programs with combinations of MIPS and subleq instructions executable on the MIPS-URISC. To support the URISC, we extend the MIPS back-end of LLVM to perform the necessary translation to subleq instructions for a given set of faulty instructions. This back-end contains a mapping from the MIPS instructions to its equivalent sequence of subleq instructions. Currently, we construct these sequences manually. Running our compiler on the source program yields an assembly program with a mix of MIPS and subleq instructions. For every URISC instruction in the assembly program, we use the .word directive to encode the binary equivalent for that instruction. This allows TigerMIPS's GCC assembler to accept the extended MIPS-URISC instruction without any alterations to the assembler itself.

4. ILLUSTRATIVE EXAMPLE

Although the primary purpose of the URISC is to allow the main processor to recover from permanent faults, the URISC can be used to correct design errors as well. We illustrate this with an example in which the URISC is used to patch an error in the jump instruction as outlined in the MIPS R4400 errata sheet [15].

The jump instructions allow branching to an address that is within 256MB memory region of the address for the delay slot following the jump instruction. The target address is computed by left shifting the immediate by two, and concatenating the high order bits of the address from the delay slot. The error occurs when there is a jump or jump-and-link instruction in any of the three words before the last word of a 256MB segment. We illustrate this issue with the sequence of instructions below as shown in the MIPS R4400 errata [15].

```
0x0fff fff0   J  imm1
0x0fff fff4   J  imm2
0x0fff fff8   J  imm3
0x0fff fffc
----256 MB End of Region-----
0x1000 0000
```

The design error is that the higher order bits are taken from an address that belongs to the next 256MB region. We replace the jump instructions with subleq instructions. However, for brevity, we use subroutine calls for basic operations such as a logical AND/OR, which we have already encoded using subleq instructions. Register u1 holds the immediate value, and PC holds the program counter.

```
subleqi  u1,u2,0      [u2 = -imm]
subleqi  u1,u2,0      [u2=-2*imm]
subleqi  u3,u3,0      [u3=0]
subleqi  u2,u3,0      [u3=-u2; u3=2*imm]
subleqi  u2,u3,0      [u3=u3-u2;2*imm-(-2*imm);
                       u3=4*imm]

subleqi  u2,u2,0      [u2=0]
subleqi  PC,u2,0      [u2=-PC]
subleqi  u4,u4,0      [u4=0]
subleqi  u2,u4,0      [u4=PC]

andsubleq(u5,u6,u4)    [args u4 and u6 as arguments;
                       the result gets stored in u5;
                       u5=u4&u6; u5 = PC & 0xF000 0000]

orsubleq(u2,u3,u5)     [call subroutine or;
                       the result is stored in u2;
                       u2=(PC&0xF000 0000)|(imm<<2)]
```

```
subleq   u4,u4,u2    [branch to absolute address
                     pointed by u2];
```

By using the URISC, we overcome the shortcomings of the jump instruction. Firstly, since there is a variant of the subleq instruction that has a register as its third operand, we can jump to a 32 bit target address by populating that register. Secondly, since the subleq instruction does not rely on a following instruction's address to get the higher-order 32 bits, there is no case where it would jump to the incorrect target address. This illustrative example shows that the URISC can be used to resolve design errors in addition to errors resulting from hard faults.

5. EXPERIMENTAL EVALUATION

Our experimental evaluation of the proposed MIPS-URISC approach investigates the area and performance overhead introduced by the URISC. For the area overhead we synthesize our design for an Altera DE2 FPGA. Next, we conduct a series of detailed fault injection experiments to determine the instructions that are rendered faulty due to single stuck-at 0/1 faults introduced in various modules of the TigerMIPS. These faulty instructions are then replaced with their subleq equivalents, and the execution time is measured and compared with the ideal execution time on the TigerMIPS alone. This provides us with the performance overhead.

5.1 Synthesis Results

We synthesized the MIPS-URISC for the Altera DE2 FPGA, and confirmed that the addition of the URISC does not impact the clock frequency (approximately 27Mhz) of the TigerMIPS. Thus, the largest critical path remains in the TigerMIPS. The number of look-up tables (LUT) increased by approximately 30% when compared to the TigerMIPS alone. However, we note that this area increase does not account for the embedded multipliers in the TigerMIPS that do not make use of any LUT resources. Therefore, we expect that in an ASIC design, the actual overhead of the URISC will be even lower.

5.2 Fault Injection Experiments

To determine the impact of stuck-at faults in the TigerMIPS core on performance, we first identify the instructions that are rendered faulty due to a given stuck-at fault.

5.2.1 Identifying Faulty Instructions

We replace each module in the TigerMIPS with an equivalent structural, gate-level net-list using ABC [16]. Stuck-at faults can be inserted at any desired gate by forcing its output to logic 0/1. We refer to this TigerMIPS core in which we insert faults as the faulty core. We detect faulty instructions by introducing a second fault-free TigerMIPS core which we call the original core.

The faulty core and the original core are fed the same stream of instructions. At each stage of the pipeline, we insert logic to detect whether the values being passed on to the next stage are different between the faulty core and the original core. If there is a difference, we mark these instructions as faulty.

Once an instruction is identified as faulty, we must ensure that its incorrect results do not propagate to the following instructions. Hence, we introduce additional logic that updates the pipeline registers of the faulty core with correct values from the original core whenever a fault is observed. By doing so, we can isolate instructions that are rendered faulty *only* as a result of the introduced stuck-at fault, and not because of data dependencies between the current instruction and an older, faulty instruction.

Using the set-up described above, we introduced stuck-at-zero and stuck-at-one faults in each gate of the Decode, ALU and Shifter modules of the TigerMIPS core. With one unique stuck-at fault in each experiment, this resulted in more than 3500 experimental runs. For each experiment, we recorded the number of faulty instruction types over five different benchmarks. The cumulative density function (CDF) of the number of faulty instruction types for each module is shown in Figure 3a.

Intuitively, we expect faults in the Decode module to render the largest number of instructions faulty. This is indeed the case. In fact, one of the stuck-at faults in the Decode module renders 35 of the 36 instruction types used across all five benchmarks faulty. On the other hand, faults in the Shifter module cause relatively less damage. In the worst case, a fault in the Shifter results in at most five faulty instruction types. We note that the hardware modifications described in this section are for the purposes of experimental evaluation only, and do not constitute a part of the proposed MIPS-URISC design.

5.2.2 Performance Overhead

We report the average execution time overhead based on the stuck-at fault injection experiments in Table 1. The overhead is computed with respect to the execution time of each benchmark executing on the TigerMIPS alone, in the absence of any faults.

Execution Time Overhead (%)					
	adpcm	**q_srt**	**bb_srt**	**miller_rabin**	**RSA**
Decoder	746.2	601.5	543.8	1233.6	2738.5
ALU	490.7	236	188.2	3875.9	7129.7
Shifter	67.9	103.1	98.4	947.6	5.1
Average	434.9	313.5	276.8	817	4203.3

Table 1: Average execution time overhead for different benchmarks and modules.

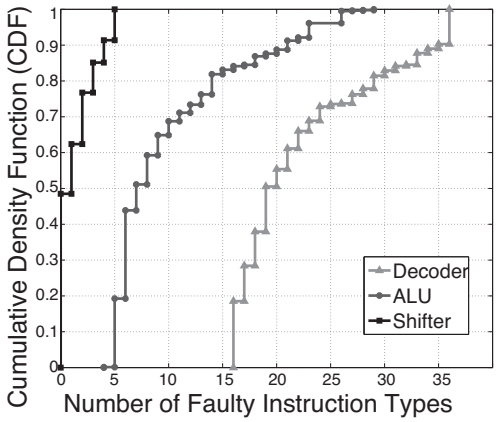
Note that the performance overheads are benchmark dependent — for example, the average performance overhead for **bb_srt** over all injected faults is only 276.8%, while that of **RSA** is 4203%. In addition, some modules have a greater performance impact than others — faults in the Shifter, for example, have on average much lower performance impact than faults in the Decoder. Figure 3b shows the CDF of performance overhead for faults in each of the three modules. While the worst case performance impact can be significant, particularly for the Decoder and ALU modules, 20% of the Decode stage faults and 40% of ALU faults have an execution time that is within $3\times$ of the fault-free execution time.

Figure 3c shows the contribution of each faulty instruction type to the total performance overhead for a single Decoder fault and a single ALU fault for the **adpcm** benchmark. The ALU fault disables four instructions, while the Decoder fault disables seven. Instructions such as MULT, DIV and ANDI introduce a greater performance impact than the jump and branch instructions, since the latter have simple subleq encodings. Notwithstanding the performance overhead, the MIPS-URISC does provide full fault coverage over all ≈ 3500 faults that were inserted in the TigerMIPS core in our experimental evaluation.

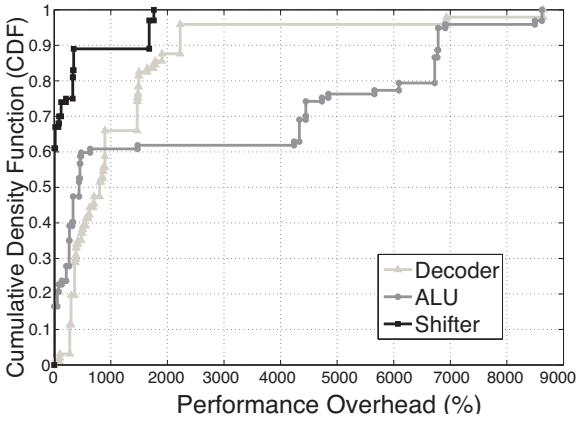
5.2.3 Comparison With Detouring

Although the performance impact in some cases is significant, the proposed techniques can be used in conjunction with software mechanisms such as detouring [5] to reduce the performance overhead. In this context, the compiler can first attempt to re-code faulty instructions using only fault-free MIPS instructions (i.e., detouring), and use subleq sequences only when alternate MIPS based re-encodings do not exist.

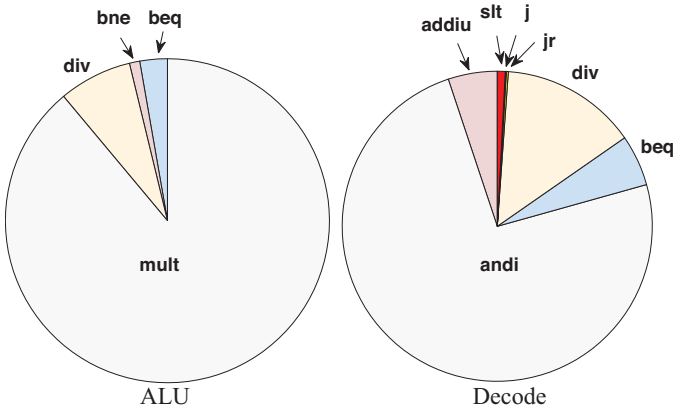
In fact, in their evaluation of detouring, Meixner and Sorin observe that recovering from permanent faults in the Decoder module



(a) CDF of the number of faulty instruction types for stuck-at faults in different modules.



(b) CDF of the performance overhead for different stuck-at faults in the Decoder, ALU and Shifter modules.



(c) Contribution of each faulty instruction to the performance overhead for a stuck-at fault in the ALU and Decoder module.

Figure 3: Experimental evaluation results.

and some ALU module faults is particularly challenging. Specifically, Meixner and Sorin point out that faults in the Decoder will likely result in a large number of instructions being disabled (our experimental results in Figure 3a validate this observation), which might make it impossible to detour some faulty instructions. For the example shown in Figure 3c, it might be possible to detour the ANDI, ADDIU and DIV instructions, but not the faulty jump and branch instructions (although this needs to be formally proven and constitutes a research challenge in itself). The URISC can be used for these instructions. Similarly, ALU faults that disable both add and subtract functionality are difficult to recover from using detouring, although they have low overhead subleq encodings.

6. CONCLUSION

This work presents MIPS-URISC as a hardware-software approach to tolerate hard faults arising from aggressive technology scaling, and design defects from human error. The URISC implements a Turing-complete instruction called subleq that can be used to recode faulty instructions resulting from defects in the main core. We implement the MIPS-URISC as a prototype on an Altera FPGA, and we provide an LLVM-based compiler to generate the subleq instructions for the instructions identified as faulty. Our experimental evaluation considers the area and performance of extending a TigerMIPS with the URISC. In addition, we explore the impact of injecting stuck-at 0/1 hard faults in the main core on the types and number of instructions that are rendered faulty, and its effect on performance.

7. REFERENCES

[1] D. Gizopoulos, M. Psarakis, S. Adve, P. Ramachandran, S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "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.

[2] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," vol. 35, no. 2. New York, NY, USA: ACM, Jun. 2007, pp. 470–481.

[3] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *proceedings of the 29th International Symposium on Computer Architecture*. ACM, 2002, pp. 99–110.

[4] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling+ dmr: practical and low-overhead permanent fault detection," in *Computer Architecture, 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 201–212.

[5] 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.

[6] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 497–508.

[7] 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.

[8] 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.

[9] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Necromancer: enhancing system throughput by animating dead cores," *ACM SIGARCH-Computer Architecture News*, vol. 38, no. 3, p. 473, 2010.

[10] P. Subramanyan, V. Singh, K. K. Saluja, and E. Larsson, "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 1572–1577.

[11] F. Mavaddat and B. Parhami, "URISC: the ultimate reduced instruction set computer," *International Journal of Electrical Engineering Education*, vol. 25, pp. 327–34, 1988.

[12] S. Ananthanarayanan, S. Garg, and H. D. 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. 1–6.

[13] S. Moore and G. Chadwick, http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html.

[14] LLVM Team, http://llvm.org/.

[15] MIPS, http://www.mips.com/media/files/archives/R4400PC_SCErrata,ProcessorRevision1.0.pdf.

[16] A. Team, "ABC: A System for Sequential Synthesis and Verification," http://www.eecs.berkeley.edu/ alanmi/abc/.

Dan Wang is a research associate in the Department of Electrical and Computer Engineering at University of Waterloo. She received the M.S. degree in Computer engineering from Polytechnic Institute of New York University. She also received the M.S. degree in microelectronics and solid-state electronics from Xidian University. Her current research interests are in the field of fault-tolerant computer architectures.

Aravindkumar Rajendiran received a BASc degree in the Department of Electrical and Computer Engineering at University of Waterloo. He is currently a software engineer at LinkedIn.

Sundaram Ananthanrayanan received the B.E. degree in Computer Science from Anna University, Guindy. He is currently pursuing the Masters degree with the Department of Electrical Engineering, Stanford University, Stanford. His research interests include computer architecture, and real-time embedded systems.

Hiren Patel is an assistant professor in the Department of Electrical and Computer Engineering at the University of Waterloo. He received his Ph.D. in Computer Engineering from Virginia Tech. His research interests are in embedded software and hardware systems. This includes models of computation, real-time systems, computer architecture, and system-level design.

Mahesh V. Tripunitara is an assistant professor in the Department of Electrical and Computer Engineering at the University of Waterloo. He has a PhD in computer science from Purdue University, and a BSc(Hons.) in computer science from Dalhousie University, Halifax, Canada. He researches various aspects of security and reliability, including access control, conditional payments and ultra-reduced instruction set computing.

Siddharth Garg is an assistant professor in the Department of Electrical and Computer Engineering at the University of Waterloo. He received a Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University, an M.S. degree in Electrical Engineering from Stanford University and a B.Tech. degree in Electrical Engineering from the Indian Institute of Technology (IIT) Madras. His primary research interests are in computer hardware and digital system design.