Reli: Hardware/Software Checkpoint and Recovery Scheme for Embedded Processors

Tuo Li[‡], Roshan Ragel[†] and Sri Parameswaran[‡]
[‡]School of Computer Science and Engineering, University of New South Wales, Sydney, Australia
{tuol,sridevan}@cse.unsw.edu.au
[†]Department of Computer Engineering, University of Peradeniya, Peradeniya, Sri Lanka
roshanr@pdn.ac.lk

***Abstract*—Checkpoint and Recovery (CR) allows computer systems to operate correctly even when compromised by transient faults. While many software systems and hardware systems for CR do exist, they are usually either too large, require major modifications to the software, too slow, or require extensive modifications to the caching schemes. In this paper, we propose a novel error-recovery management scheme, which is based upon re-engineering the instruction set. We take the native instruction set of the processor and enhance the microinstructions with additional micro-operations which enable checkpointing. The recovery mechanism is implemented by three custom instructions, which recover the registers which were changed, the data memory values which were changed and the special registers (PC, status registers etc.) which were changed. Our checkpointing storage is changed according to the benchmark executed. Results show that our method degrades performance by just 1.45% under fault free conditions, and incurs area overhead of 45% on average and 79% in the worst case. The recovery takes just 62 clock cycles (worst case) in the examples which we examined.**

I. Introduction

Electronic systems must be protected against transient faults, so that they can be relied upon and used for longer periods [21], [22]. As the feature size shrinks, transistors within an embedded system become reportedly more susceptible to adverse effects such as *transient faults* (e.g. soft errors) due to energized particle hits [3], [16]. Addressing this problem firstly requires the ability to detect errors (this has been intensively studied in [4], [17], [24]) and to recover from errors When an error has been detected.

Checkpoint and Recovery (CR) has been studied as a viable methodology for error recovery of transient faults [26]. The basic concept of CR is to recover the current application process by using the most recent checkpoint. A checkpoint, which is generated periodically, is a set of data that keeps a copy of the verified system state. Depending on the particular mechanism and implementation, the checkpoint data size and checkpoint period vary. CR requires additional resources for both generating checkpoint and committing recovery. Recent research discusses two types of CR techniques. One is *software-based CR* that typically has large code size or considerable fault-free performance overhead [1] [12]. The other is *hardware-based CR* (e.g. cache-based) that introduces specific modifications to the microarchitectures of processors (e.g. cache replacement policy for cache-based system) and are not systematic [30].

Since embedded systems usually have to meet stringent design constraints (e.g. real-time, power, area, etc.), CR for embedded systems have to be small, fast and energy efficient. Using existing CR techniques will worsen size, time and energy constraints. *Custom instruction set* processor, e.g. tensilica Xtensa [29], is a HW/SW codesign methodology for designing embedded systems. In [13], [23], the authors have explored building online code integrity checking mechanisms via *customizing instruction set* for ASIP-based embedded systems. Their studies have established a method of using *custom instruction set* to augment the target system with fault-tolerance techniques.

In this paper, we propose a technique called Reli, which is a system-level error-recovery management scheme for ASIP-based embedded systems. To the best of our knowledge, Reli is the first of its kind. Reli reforms classic CR by leveraging *custom instruction set*. Consequently, the cost in terms of execution time and area are reduced significantly compared to existing systems.

The rest of the paper is structured as follows. Section **2** provides a discussion of related work. Section **3** gives an elaboration on the problem of designing CR techniques in context of embedded systems. Section **4** and **5** depict the concept and implementation of Reli technique respectively. Section **6** provides an experimental study that is followed by a discussion in Section **7**. Section **8** concludes this paper.

II. Related Work

Error-recovery techniques can be categorized into two: roll-forward; and, roll-backward (i.e. CR). Roll-forward techniques typically adopt redundant computations to be able to compare or vote on the fly to mitigate faults [2]. Roll-forward techniques incur additional cost in terms of chip area and power/energy consumption, but are fast.

CR techniques trade recovery latency to gain area savings. The goal of CR is to facilitate the regaining of normal functionality as soon as possible after the occurrence of transient faults [26]. CR techniques typically use additional storage to hold check point data, so that if an error occurs, then they can be rolled back to a point where the checkpointing was last performed. The additional storage for holding checkpoint data and the hardware needed to rollback can impact on the execution time in both the fault free scenario and in the recovery scenario, Existing CR techniques fall into two categories, depending on the manner in which they address the recovery problem.

Software-based CR techniques require no additional hardware [5], [7], [8], [12], [15], [18], [31]. In [12] the authors modify compilers to insert checkpoint routine code into native code. It is reasonably fast but induces large checkpoint data size, and the static code size is in increased. In [15], the authors propose a thread-level checkpoint and recovery mechanism, which needs OS support. In [31], the authors propose a loadable kernel module (RMK) to support application-level checkpointing. In [5], [7], [8], [18], the authors propose different approaches to perform checkpoint and recovery for parallel programs on shared memory symmetric multiprocessors.

Hardware-based CR techniques[2] use special customization and

[1]Fault-free performance is the speed while there is no fault happening. It is an important factor in measuring the practicality of CR based systems.

[2]Also called Backward Error Recovery (BER).

optimization in arbitrary microarchitectural components (mainly storage components which contain process state) to implement CR [1], [11], [20], [28], [30]. In [28], the authors propose a checkpoint and recovery approach that uses memory (and cache) for multiprocessor systems. However, no accurate experiment data is shown for recovery latency. In [20], the authors propose a CR technique for shared memory multiprocessor system , that has a rollback delay of 0.82 s for 80 ms checkpoint frequency in the worst case. The above CR scheme involves modification of the directory controller of the memory for recording (called "logging") the checkpoint data. Checkpointing and recovery are controlled by timer-interrupt and the protocol is implemented in software. In [1], [11], [30], the authors propose Cache-based CR techniques for single- and multi-processor systems. These techniques use specially designed cache as a buffer to hold the temporary data for computation in the middle of checkpoints. In addition, the register files are duplicated for backing up the register file state. They lack the flexibility of selecting when to commit checkpoint and are thus relatively unpredictable. In addition, the cache replacement policy must be modified in hardware.

The software based systems [5], [7], [8], [12], [15], [18], [31] checkpoint all necessary variable and thus incur enormous checkpointing data size, slowing down recovery. Additionally, they need the program to be modified to allow for CR functionality. The granularity of the checkpoints of software based systems are quite large in the order of milliseconds or even seconds.

Many of the hardware based techniques [1], [11], [20], [27], [28], [30] thus far have either relied on register, cache, or memory to back up checkpoint data. Register based techniques [27] can introduce up to 1000 cycles for recovery time. Cache based techniques [1], [11], [30] have to modify cache replacement policy and by its nature are dependent on having a cache in the system. Memory based systems [20], [28] are slow due to the fact that the checkpointing has to be performed in memory.

Our proposed technique, namely Reli, is a system-level CR technique for embedded processors. In contrast to the previous methods we integrate the CR algorithm with custom instructions into the processor. In each instruction (the state modifying instructions) the processor can commit checkpointing automatically. In addition, checkpoints are assigned at a far finer granularity (i.e. instruction and basic block level) than previously considered, so that the realtime constraints can be more easily met compared to existing techniques. Moreover the cost in terms of execution time (performance), recovery latency, and chip area is reduced to a modest level. The contributions of this paper are as follows:

(1) For the first time CR functionality is integrated into the instructions, such that the system is transparent to the programmer (i.e., the programmer does not have to modify the code in any way);
(2) CR is implemented at a finer granularity than previously possible, allowing frequent checkpointing, and rapid recovery, when needed;
(3) Any RISC architecture base system (with or without cache) can be altered using the method described in this paper to achieve a fast system with small area overhead.

III. Problem Statement

Given an embedded application, and a target RISC instruction set, create a processor which can perform both checkpointing and recovery with negligible performance reduction, and modest area increase. The created system should also be transparent to the high level programmer who uses this system.

A. Reli CR Model

Reli's philosophy uses *custom instructions* to realize CR functionality. The given application can be decomposed into multiple instructions of a given processor. These instructions typically are composed of micro-operations.

In this work we use the same instruction set of the given processor (we have three additional instructions for recovery, but they are not used in the application), but modify the micro-operations in those instructions which change the state of the processor. Examples of instructions which change the state of the processor are: instructions which modify registers or instructions which modify data memory values. At run-time, *Reli instructions* execute not only native functionalities (e.g., adding two operands of the ADD instruction), but also *Reli functionality* (e.g., generating checkpoint data of destination register for ADD instruction).

Although in principle, the processor's architecture needs to be modified to support *custom instructions*, with architectural descriptions, synthesis techniques (i.e. architectural synthesis) are available which take the micro-operations of the instruction set and create the processor itself. (e.g. extensible processor platforms) [14].

We assume a system with parameters specified as follows. The system is compatible and ASIP based (we assume single core at present). The baseline HW configuration (architecture) is assumed to have: on-chip memory (data RAM and instruction ROM), integer arithmetic unit (incl. multiplier and divider), RISC-compatible control unit, and without floating point support, delay slot support. Though the native instruction set is assumed to exclude floating point instructions, the same methodology can be applied to those instructions. We could have the system with or without cache, though all examples and experimentations in this paper did not use cache.

IV. Reli Approach

The entire approach can be divided into three parts. The checkpoint generation (i.e. backing up of data points necessary for successful recovery); the recovery (if an error has occurred, the data which was stored has to be restored, and the execution should seamlessly commence); and the testing and validation (where the system tests for correctness). This work concentrates only on the checkpoint generation and the recovery. The testing and validation have been extensively covered by other researchers [17], [24].

The entire methodology is performed by changing the micro-operations of instructions which change registers and data memory. In addition, the micro-operations of control flow instructions are changed. Three additional instructions are created which enable recovery. Note however, that these three instructions do not interleave within the legacy code. They are used in a separate routine which is used for recovery.

A. Checkpointing

For checkpointing to be performed correctly, the registers, the status registers and the data memory elements which are changed since the last checkpoint have to be stored, as well as the program counter.

To enable a scalable recovery system, we utilize two stacks. These are used for storing the registers in the register file and for storing the data memory values which are changed. At the start of a basic block, the program counter and the status register are backed up in their own backup locations.

Algorithm 1 shows the methodology used for backing up the necessary components. For registers, the first time a register is changed in a basic block, the old value of that register is stored

Algorithm 1 Algorithmic description of Checkpointing for register file state

```
    $history_n$: value in logging history table (boolean) of the register $n$
    $dist$: destination register's address decoded from the native instruction
    $stack_p$: a stack data structure that stores checkpoint data with pointer value $p$
    $val_n$: value of register $n$
    $data_n$: checkpoint data of register $n$
    // After the decoding is finished, and $dist$ is known to instruction layer.
1: if $history_{dist} = true$ then
2:     do nothing
3: else
4:     $history_{dist} \leftarrow true$
5:     $data_{dist} \leftarrow dist \parallel val_{dist}$
6:     $stack_p \leftarrow data_{dist}$
7:     $p \leftarrow p + 1$
8: end if
```

(along with the name of the register). A subsequent change to that register within the basic block does not affect the backup stack. The reason for doing a single store of a register is to reduce the size of the stack. To enable this we have a history flag for each register, which is reset at the beginning of each basic block. For the particular implementation that is shown in this paper, though the register file contained 32 registers, the back up stack for the register file was only 16 locations. We profiled a number of applications (such as SPEC INT 2006 and MiBench suites [9]), to find this number.

For data memory the old values are simply stored in the data memory stack along with the address of the location. There is no history flag for the data memory, as the number of flags would be excessive. Every time a store instruction is encountered, the data value and the address is pushed on to the stack. After profiling, we found the maximum number of stack locations necessary was 66.

B. Recovery

If an error occurs, the the status registers and the program counter are restored from their respective back up locations. The registers checkpoint stack is popped out one by one, until the bottom of the stack is reached (since the register name was also stored, we do know where to restore it). The data memory checkpoint stack is also popped out one by one until the stack is empty. Algorithm 2 describes how the register file is restored.

C. Fault Response

Algorithm 3 shows the fault response mechanism. In this study we assume there is a fail-free detection mechanism (e.g., those in [17], [24]). If an error occurs a *fault* flag is set. At every control flow instruction, the fault flag is checked. If there is no fault then a new check point is established (the history table is reset, the stack pointers to the backup stacks are reset, and the execution continues. If there is a fault, the program counter jumps to a special location, where the recovery starts. Three additional instructions are created,

Algorithm 2 Algorithmic description of recovery for register file state

```
    $val_n$: value of register $n$ in register file
    $D_{dist}$: destination register address extracted from checkpoint data
    $D_{val}$: destination register value extracted from checkpoint data
    $data$: checkpoint data
    $stack_p$: checkpoint data stack with pointer value $p$
1: if p = 0 then
2:     end of recovery
3: else
4:     $p \leftarrow p - 1$
5:     $data \leftarrow stack_p$
6:     $(D_{dist}, D_{val}) \leftarrow data$
7:     $val_{D_{dist}} \leftarrow D_{val}$
8:     go to Line 1
9: end if
```

Algorithm 3 Algorithmic description of fault response mechanism

```
    $f\_status$: fault status information (0: number of fault = 0, 1: number of fault $\geq$ 1.)
    $PC$: PC register
    $r\_addr$: recovery/rollback routine's address
1: if $f\_status$ = 0 then
2:     establish the new checkpoint
3:     clear logging history table and checkpoint data stack
4: else
5:     flush the pipeline
6:     $PC \leftarrow r\_addr$
7: end if
```

which restores the data memory, the registers and the status registers (including PC), from the backup stores.

V. Implementation

We use SimpleScalar PISA instruction set architecture for the native processor (32×32 regfile, 64-bit instruction). The entire CR functionality lies into both hardware and software implementations. We use a commercial ASIP design platform [14] to do the high-level synthesis, and generate the hardware description. This platform provides a simple but sufficient set of predefined blocks (structural component library) to compose a single-core in-order processor. The main design entry of the platform includes: an architectural definition (pipeline stage attributes), and a micro-operation description, ASIPmeister's Architectural Description Language (ADL), for each instruction of ISA. Micro-operation description allows defining data transfers and processings (e.g. operations, read, write, etc.) in a instruction.

The design flow is shown in Fig. 1. At first, the CR functionality is allocated, and sequenced. Allocation is the process of choosing the components for CR functionality (e.g. how to build the stack with predefined blocks). The sequencing process determines the sequence of each element in terms of instruction and pipeline stage. We use as-soon-as-possible (ASAP) as the rule-of-thumb for the sequencing process: putting the Reli micro-operations as early as possible for each instruction as long as the hardware permits. These two stages are done in iterations to avoid any data, resources, and control hazards. Then, the CR functionality is mapped into micro-operations. These CR-related micro-operations are then integrated into native instructions' micro-operations to form a complete ADL model of Reli processor. At last, the ADL model of the processor is synthesized by ASIPmeister and outcomes a HDL model[3]. Examples of Reli ADL model in ASIPmeister micro-operation language are shown in Fig. 2 and Fig. 3. Fig. 2 shows the ADD instruction for a native processor as well as for a processor which does checking and recovery. We have omitted explaining this in detail, due to lack of space. Fig. 3 shows the ADL description for three instructions necessary for recovery. Only the stages which have micro-operations are shown in the figure. Once again details are omitted due to limited availability of space. Micro-operations in stage 1 are not shown since they are the same for all instructions.

We modify the methodology proposed in [19] to do the memory generation. The methodology inserts three extra instructions as the recovery software routine into the native application code. The recovery routine address is given to hardware design flow before synthesis (micro-operation integration), so that the hardware and software can co-work together.

[3]The detail of ASIPmeister synthesis methodology is given in [25].

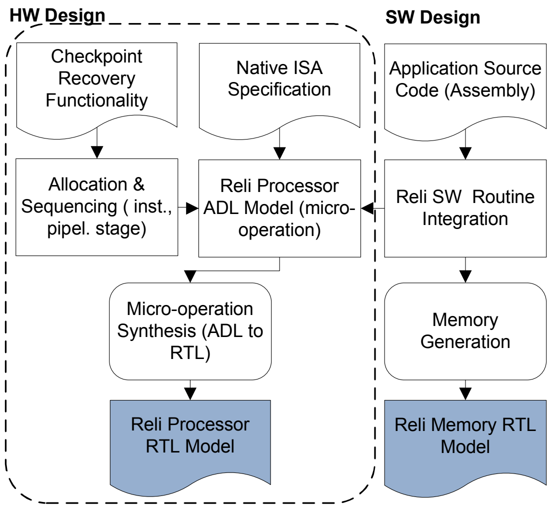
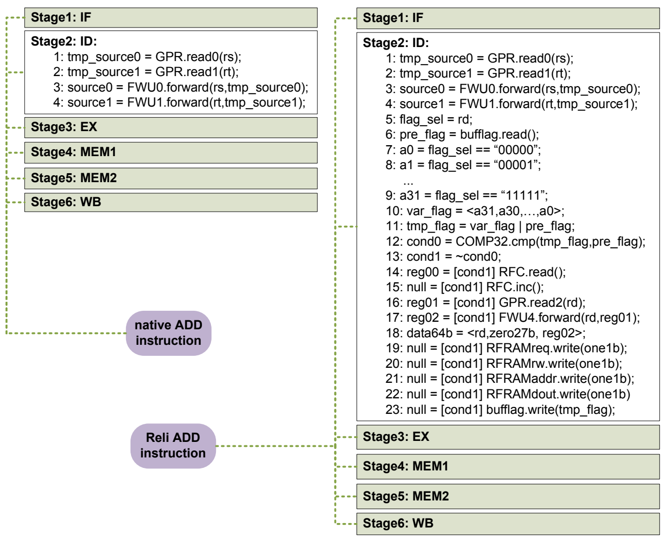

HW Design
SW Design
Checkpoint Recovery Functionality
Native ISA Specification
Application Source Code (Assembly)
Allocation & Sequencing (inst., pipel. stage)
Reli Processor ADL Model (micro-operation)
Reli SW Routine Integration
Micro-operation Synthesis (ADL to RTL)
Memory Generation
Reli Processor RTL Model
Reli Memory RTL Model

Fig. 1. Flowchart of the implementation flow



```
Stage1: IF
Stage2: ID:
    1: tmp_source0 = GPR.read0(rs);
    2: tmp_source1 = GPR.read1(rt);
    3: source0 = FWU0.forward(rs,tmp_source0);
    4: source1 = FWU1.forward(rt,tmp_source1);
Stage3: EX
Stage4: MEM1
Stage5: MEM2
Stage6: WB
```

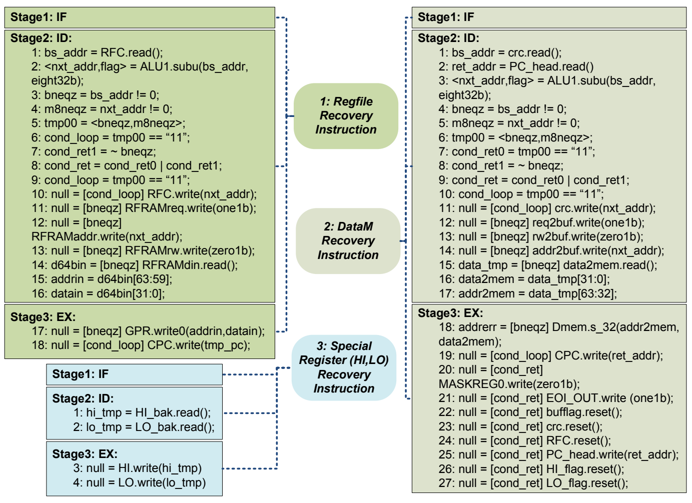
```
Stage1: IF
Stage2: ID:
    1: tmp_source0 = GPR.read0(rs);
    2: tmp_source1 = GPR.read1(rt);
    3: source0 = FWU0.forward(rs,tmp_source0);
    4: source1 = FWU1.forward(rt,tmp_source1);
    5: flag_sel = rd;
    6: pre_flag = bufflag.read();
    7: a0 = flag_sel == “00000”;
    8: a1 = flag_sel == “00001”;
      ...
    9: a31 = flag_sel == “11111”;
    10: var_flag = <a31,a30,…,a0>;
    11: tmp_flag = var_flag | pre_flag;
    12: cond0 = COMP32.cmp(tmp_flag,pre_flag);
    13: cond1 = ~cond0;
    14: reg00 = [cond1] RFC.read();
    15: null = [cond1] RFC.inc();
    16: reg01 = [cond1] GPR.read2(rd);
    17: reg02 = [cond1] FWU4.forward(rd,reg01);
    18: data64b = <rd,zero27b, reg02>;
    19: null = [cond1] RFRAMreq.write(one1b);
    20: null = [cond1] RFRAMrw.write(one1b);
    21: null = [cond1] RFRAMaddr.write(one1b);
    22: null = [cond1] RFRAMdout.write(one1b)
    23: null = [cond1] bufflag.write(tmp_flag);
Stage3: EX
Stage4: MEM1
Stage5: MEM2
Stage6: WB
```

Fig. 2. Illustration of native and Reli ADD instructions (micro-operation description)

VI. Evaluation and Results

Experiments were conducted in a server with the following configuration: Intel Xeon X7560 CPU (2.27 GHz), 24576 KB cache, and 256 GB main memory. The RTL simulation environment used was Mentor Graphics ModelSim (HDL simulator). The logic synthesis used was Synopsys Design Compiler (hardware synthesis). The application binaries (from MiBench benchmark suite [9]) were compiled using SimpleScalar PISA compiler [6]. Before going through evaluation, the baseline processors were generated, and their functionality was verified prior to experimentation.

Our evaluation methodology consists of three basic flows demonstrated in Fig. 4(a): 1) Fault-free RTL simulation providing the cycle-accurate fault-free execution time (Section VI-A), 2) Fault injection test [10] examining fault recovery time, and 3) Logic synthesis showing hardware cost that is bound to a real-world fabrication technology. In each of the flows, the results of baseline and Reli are compared, to observe Reli's overhead for each testcase application described in Fig. 4(b). The baseline processor is defined as the processor shares the basic system model with a Reli processor and

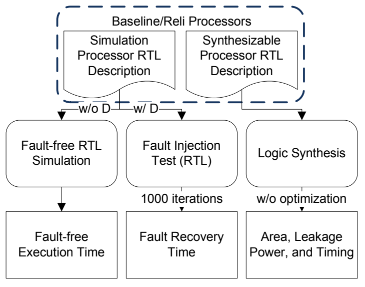

1: Regfile Recovery Instruction
2: DataM Recovery Instruction
3: Special Register (HI,LO) Recovery Instruction

```
Stage1: IF
Stage2: ID:
    1: bs_addr = RFC.read();
    2: <nxt_addr,flag> = ALU1.subu(bs_addr,
    eight32b);
    3: bneqz = bs_addr != 0;
    4: m8neqz = nxt_addr != 0;
    5: tmp00 = <bneqz,m8neqz>;
    6: cond_loop = tmp00 == “11”;
    7: cond_ret1 = ~ bneqz;
    8: cond_ret = cond_ret0 | cond_ret1;
    9: cond_loop = tmp00 == “11”;
    10: null = [cond_loop] RFC.write(nxt_addr);
    11: null = [bneqz] RFRAMreq.write(one1b);
    12: null = [bneqz]
    RFRAMaddr.write(nxt_addr);
    13: null = [bneqz] RFRAMrw.write(zero1b);
    14: d64bin = [bneqz] RFRAMdin.read();
    15: addrin = d64bin[63:59];
    16: datain = d64bin[31:0];
Stage3: EX:
    17: null = [bneqz] GPR.write0(addrin,datain);
    18: null = [cond_loop] CPC.write(tmp_pc);
```

```
Stage1: IF
Stage2: ID:
    1: hi_tmp = HI_bak.read();
    2: lo_tmp = LO_bak.read();
Stage3: EX:
    3: null = HI.write(hi_tmp)
    4: null = LO.write(lo_tmp)
```

```
Stage1: IF
Stage2: ID:
    1: bs_addr = crc.read();
    2: ret_addr = PC_head.read()
    3: <nxt_addr,flag> = ALU1.subu(bs_addr,
    eight32b);
    4: bneqz = bs_addr != 0;
    5: m8neqz = nxt_addr != 0;
    6: tmp00 = <bneqz,m8neqz>;
    7: cond_ret0 = tmp00 == “11”;
    8: cond_ret1 = ~ bneqz;
    9: cond_ret = cond_ret0 | cond_ret1;
    10: cond_loop = tmp00 == “11”;
    11: null = [cond_loop] crc.write(nxt_addr);
    12: null = [bneqz] req2buf.write(one1b);
    13: null = [bneqz] rw2buf.write(zero1b);
    14: null = [bneqz] addr2buf.write(nxt_addr);
    15: data_tmp = [bneqz] data2mem.read();
    16: data2mem = data_tmp[31:0];
    17: addr2mem = data_tmp[63:32];
Stage3: EX:
    18: addrerr = [bneqz] Dmem.s_32(addr2mem,
    data2mem);
    19: null = [cond_loop] CPC.write(ret_addr);
    20: null = [cond_ret]
    MASKREG0.write(zero1b);
    21: null = [cond_ret] EOI_OUT.write (one1b);
    22: null = [cond_ret] bufflag.reset();
    23: null = [cond_ret] crc.reset();
    24: null = [cond_ret] RFC.reset();
    25: null = [cond_ret] PC_head.write(ret_addr);
    26: null = [cond_ret] HI_flag.reset();
    27: null = [cond_ret] LO_flag.reset();
```

Fig. 3. Three new recovery instructions (1: Register File State Recovery, 2: Data Memory State Recovery, and 3: Special Register State Recovery Instructions, each of which uses three pipeline stages.)


Baseline/Reli Processors
Simulation Processor RTL Description
Synthesizable Processor RTL Description
w/o D
w/ D
Fault-free RTL Simulation
Fault Injection Test (RTL)
Logic Synthesis
1000 iterations
w/o optimization
Fault-free Execution Time
Fault Recovery Time
Area, Leakage Power, and Timing

(a) Evaluation flows

Application	Description
ADPCM.enc	Encode part of ADPCM. Computation intensive.
ADPCM.dec	Decode part of ADPCM.
blowfish.enc	Encrypt part of blowfish. Computation intensive.
blowfish.dec	Decrypt part of blowfish.
CRC32	32-bit cyclic redundancy check. Control intensive.
stringsearch	Searchs for given words in phrases. Control and memory intensive.

(b) Testcase applications

Fig. 4. Evaluation methodology overview (D: detection mechanism)

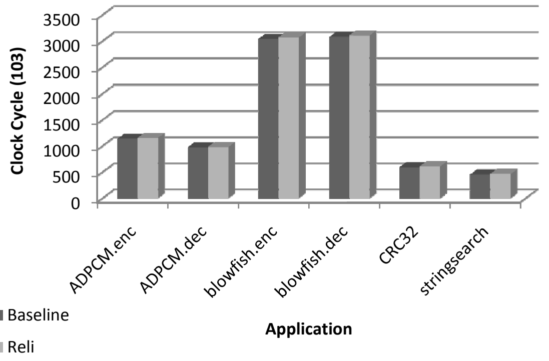
only performs native instruction set.

A. *Fault-free Execution Time*

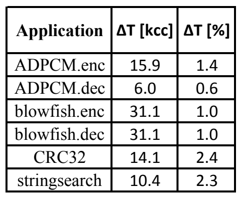
Baseline processor (without any recovery mechanism installed) and the Reli processor are compared for fault free performance. Fig. 5 shows the fault-free execution time penalty of processors equipped with Reli technique for five applications. We compared Reli's performance with that of the baseline processor for each application. The worst case is 2.4% in additional penalty for CRC32 while the best case was 0.6% for the ADPCM.dec. The average overhead was 1.45%.

B. *Fault Recovery Time*

1) Fault Injection Test: We used a single bit-flip as the fault model for this study. To yield statistically significant results, the number of fault injections was 1000 for each application. We injected one fault for each iteration. The test had three procedures: 1) Injecting a fault at compile time to the instruction memory data file; 2) Invoking the HDL simulator to run the application; 3) Collecting the run-time behavior from the simulation transcript. The test environment is largely implemented via Python scripts. To inject a fault, a random address of the instruction memory is chosen for each iteration. Then a random bit of the chosen address is flipped to the opposite binary value. To make the simulation as close as possible to a realistic one, we implement a detection technique similar to IMPRES [23] to work with Reli. The detection mechanism monitors bit-flips of instructions,



(a) Bar illustration



Application	ΔT [kcc]	ΔT [%]
ADPCM.enc	15.9	1.4
ADPCM.dec	6.0	0.6
blowfish.enc	31.1	1.0
blowfish.dec	31.1	1.0
CRC32	14.1	2.4
stringsearch	10.4	2.3

(b) Quantification

Fig. 5. Fault-free execution time (kcc: 10^3 clock cycle; ΔT: time overhead over baseline)

TABLE I
STATISTICS OF A THOUSAND TIMES FAULT INJECTION TEST

Application	T_R [cc] min	T_R [cc] max	$\bar{T}_R$ [cc]
ADPCM.enc	5	50	13.8
ADPCM.dec	5	47	14.2
blowfish.enc	6	60	17.7
blowfish.dec	5	62	17.9
CRC32	5	47	16.0
stringsearch	8	35	13.9

and communicates to Reli at the end of every basic block. Since library code is difficult to modify, the faults in library code were excluded in this test.

2) Result Discussion: Table I shows the impact of the proposed technique on performance when faults occur. We have six categories for six selected applications. For each category, the average latency of recovery ($\bar{T}_R$), and the minimum and maximum latency of recovery (T_R) are provided.

Among the six applications, blowfish.dec has the highest average recovery latency, i.e. 17.9 clock cycles. Whereas ADPCM.dec has the smallest average recovery latency (13.8 clock cycles). The worst recovery latency (62 clock cycles) is observed in blowfish.dec, while the best case (5 clock cycles) is found in ADPCM.enc, ADPCM.dec, blowfish.dec and CRC32. Importantly, all the effective faults are successfully recovered by the proposed scheme in the test. This result suggests Reli is capable of recovering from all occurrences of faults from the fault model.

C. Core Results

We obtained synthesis results with TSMC $65nm$ library using the Synopsys logic synthesis environment. To yield results which were comparable, no optimizations were applied in this process. The results are shown in Table II and discussed below in terms of area and leakage power between seven typical prototypes.

1) Impact on Area and Leakage Power: We compare the baseline processor with: (1) Six Reli processors, each targeting one of the six applications selected from MiBench benchmark suite, and (2), combined single processor with multi-applications, which is used to examine the the worst case. In each application, the size of checkpoint stacks is specific and determined by that application's behavior (the number of writes in basic blocks), as explained in Section IV. These stacks contribute significantly to the area and leakage power increase (particularly in the worst case scenario). To decide the worst case, we have analyzed SPEC-INT2006 and MiBench applications, from which the largest numbers of register file and data memory writes are calculated within a basic block. These are found in H264 and GCC applications. The worst-case result aims to show the cost when a random application from embedded application domain is executed. The overhead in percentage is calculated against the baseline processor. As is shown, without optimization, Reli costs from 37.8% to 52.0% more area for the examined applications and 79.3% for the worst case; while 39.6% to 52.4% more leakage power for examined applications and 77.8% at the worst case.

VII. DISCUSSION

Comparison — Due to the fact that extensive data does not exist for previous techniques, we have tried to compare with existing techniques as much as it is possible. Reli outperforms most of existing CR techniques (e.g., up to 20% for software-based [4]) for fault-free execution. Our system increases time by just 2.4% clock cycles in the worst case. On the recovery time, Reli takes at most 64 clock cycles, and is faster than others (e.g., 1000 cycles in [27], 0.1 to 1 seconds in [20]). In addition, Reli's checkpoint data size (624 bytes in the worst case) is much smaller compared to others (e.g. mega bytes for software-based techniques).

Impact on Clock Period — Reli increases the number of data transfers and operations for every instruction of the native instruction set. Therefore, it is intuitive to hypothesize that Reli might incur a certain amount of overhead on the minimum achievable clock period of the circuit. However, our synthesis result, which is obtained before placement-and-routing, shows the target processor's clock period (52 ns, obtained without optimization) can be achieved even without any delay optimization for Reli. If optimizations are considered during synthesis, Reli's impact on clock period could well be negligible. It is likely that many of the micro-operations are performed in parallel. And the critical path is not affected by any of the added micro-operations.

Reliability — When a fault occurs in Relis checkpoint stacks, given the fact that such memory units are error-prone to soft errors, rapid error correcting techniques (e.g., ECC) can be considered to improve the reliability of checkpoint stacks. Moreover, similar techniques (e.g., two-time-recovery) used in the recovery mode in IBM G5 [27] can be adopted in Reli's framework to guarantee that the recovery is executed correctly.

Scalability — Reli currently is studied on a uni-processor system. However, this technique can be scaled to multi-processor system-on-a-chip (MPSoC) systems by adding taking a communication mechanism into consideration. Further experiment on MPSoC systems would be necessary.

VIII. CONCLUSION

In this paper we have presented a novel approach for recovering embedded applications from transient faults by using custom instructions. We have realized a classic recovery algorithm, checkpoint and recovery (CR By integrating CR functionality into native instructions, we build custom instruction set processors that have built-in CR functionality. This allows CR to be executed at a finer granularity

[4]Data for the techniques in [1], [11], [27] are not available.

TABLE II
SYNTHESIS RESULT WITH TSMC $65nm$ TECHNOLOGY (S_{DM}: SIZE OF THE DATA MEMORY STACK. S_{RF}: SIZE OF THE REGISTER FILE STACK SIZE. A_{stack}: AREA OF DATA MEMORY AND REGISTER FILE STATE STACKS. A_{total}: AREA OF TOTAL PROCESSOR. Δ: OVERHEAD COMPARED TO BASELINE. P: LEAKAGE POWER. WC: WORST CASE.)

Processor	S_{DM} [byte]	S_{RF} [byte]	A_{stack} $[um^2]$	A_{total} $[um^2]$	ΔA [%]	P_{stack} $[uw]$	P_{total} $[uw]$	ΔP [%]
Baseline	0	0	0	120981	0.0	0	613	0.00
ADPCM.enc	88	48	12322	171661	41.9	56	879	43.4
ADPCM.dec	88	48	12322	171661	41.9	56	879	43.4
blowfish.enc	192	80	24600	183931	52.0	112	934	52.4
blowfish.dec	192	80	24600	183931	52.0	112	934	52.4
CRC32	72	48	10951	170294	40.8	50	872	42.3
stringsearch	40	40	7401	166737	37.8	34	856	39.6
WC(h264/gcc)	528	96	58015	209474	79.3	263	1090	77.8

than previously possible, such that the checkpoint data size is reduced greatly. To implement our approach, we have used a commercial ASIP design tool to handle the ADL to RTL synthesis. We have simulated our approach using assembly code that are compiled using the SimpleScalar PISA tool set from MiBench benchmark suite. Experiment results show that the fault-free performance overhead is only 1.45% on average. From the fault injection test, we also found that in the worst case the recovery delay is only 62 cycles. Our approach costs 44.4% area and 45.6% leakage power overhead in average, and 79.3% and 77.8% in the worst case found in SPEC INT 2006 and MiBench suites.

REFERENCES

[1] R. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (carer) algorithm for shared-memory multiprocessor systems. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 82 –88, jun 1990.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.

[3] R. Baumann. Soft errors in commercial semiconductor technology: overview and scaling trends. In *international reliability physics symposium*, 2002.

[4] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 421–431, New York, NY, USA, 2006. ACM.

[5] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz. Application-level checkpointing for shared memory programs. *SIGOPS Oper. Syst. Rev.*, 38:235–247, October 2004.

[6] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[7] W. R. Dieter and J. E. Lumpp Jr. A user-level checkpointing library for posix threads programs. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, pages 224–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. 2003.

[9] M. R. Guthaus, J. Ringenberg, D. Ernst, T. Mudge, R. Brown, and T. Austin. MiBench: a free, commercially representative embedded benchmark suite. In *IEEE International Symposium on Workload Characterization*, 2001.

[10] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30:75–82, April 1997.

[11] D. Hunt and P. Marinos. A general purpose cache-aided rollback error recovery (CARER) technique. In *proceedings of the 17th international symposium on fault-tolerant coputing systems*, pages 170–175, 1987.

[12] C.-C. Li and W. Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74 –81, jun 1990.

[13] H. Lin, Y. Fei, X. Guan, and Z. J. Shi. Architectural enhancement and system software support for program code integrity monitoring in application-specific instruction-set processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 18:1519–1532, November 2010.

[14] P. Mishra and N. Dutt. *Processor Description Languages, Volume 1*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[15] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An architectural framework for providing reliability and security support. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 585, Washington, DC, USA, 2004. IEEE Computer Society.

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

[17] N. Oh, P. Shirvani, and E. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63 –75, mar 2002.

[18] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. Panda. Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 99 –108, dec. 2009.

[19] J. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran. Rapid embedded hardware/software system generation. *VLSI Design, International Conference on*, 0:111–116, 2005.

[20] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 111–122, Washington, DC, USA, 2002. IEEE Computer Society.

[21] J. M. Rabaey. Design at the end of the silicon roadmap. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 1–2, New York, NY, USA, 2005. ACM.

[22] J. M. Rabaey and S. Malik. Challenges and solutions for late- and post-silicon design. *IEEE Des. Test*, 25:296–302, July 2008.

[23] R. G. Ragel and S. Parameswaran. IMPRES: integrated monitoring for processor reliability and security. In *Design Automation Conference*, pages 502–505, 2006.

[24] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000.

[25] T. Shiro, M. Abe, K. Sakanushi, Y. Takeuchi, and M. Imai. A processor generation method from instruction behavior description based on specification of pipeline stages and functional units. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 286–291, Washington, DC, USA, 2007. IEEE Computer Society.

[26] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[27] T. Slegel, I. Averill, R.M., M. Check, B. Giamei, B. Krumm, C. Krygowski, W. Li, J. Liptay, J. MacDougall, T. McPherson, J. Navarro, E. Schwarz, K. Shum, and C. Webb. Ibm's s/390 g5 microprocessor design. *Micro, IEEE*, 19(2):12 –23, mar/apr 1999.

[28] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *SIGARCH Comput. Archit. News*, 30:123–134, May 2002.

[29] tensilica. http://www.tensilica.com.

[30] R. Teodorescu, J. Nakano, and J. Torrellas. SWICH: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro*, 26:28–40, 2006.

[31] L. Wang, Z. Kalbarczyk, W. Gu, and R. K. Iyer. An OS-level framework for providing application-aware reliability. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 55–62, Washington, DC, USA, 2006. IEEE Computer Society.