CSER: HW/SW **C**onfigurable **S**oft-**E**rror **R**esiliency for Application Specific Instruction-Set Processors

Tuo Li[†], Muhammad Shafique[‡], Semeen Rehman[‡], Swarnalatha Radhakrishnan[†], Roshan Ragel[†],
Jude Angelo Ambrose[†], Jörg Henkel[‡], and Sri Parameswaran[†]
[†]School of Computer Science and Engineering, University of New South Wales, Australia
{tuol,ajangelo,sridevan}@cse.unsw.edu.au;{swarna.radhakrishnan,ragelrg}@gmail.com
[‡]Chair for Embedded Systems, Karlsruhe Institute of Technology (KIT), Germany
semeen.rehman@student.kit.edu;{muhammad.shafique,henkel}@kit.edu

Abstract—**Soft error has been identified as one of the major challenges to CMOS technology based computing systems. To mitigate this problem, error recovery is a key component, which usually accounts for a substantial cost, since they must introduce redundancies in either time or space. Consequently, using state-of-art recovery techniques could heavily worsen the design constraint, which is fairly stringent for embedded system design. In this paper, we propose a HW/SW methodology that generates the processor, which performs finely configured error recovery functionality targeting the given design constraints (e.g., performance,area and power). Our methodology employs three application-specific optimization heuristics, which generate the optimized composition and configuration based on the two primitive error recovery techniques. The resultant processor is composed of selected primitive techniques at corresponding instruction execution, and configured to perform error recovery at run-time accordingly to the scheme determined at design time. The experiment results have shown that our methodology can at best achieve nine times reliability while maintaining the given constraints, in comparison to the state of the art.**

I. Introduction and Motivation

Deploying soft-error countermeasures in a computing system essentially requires designing extra functionality, and combining it with the native functionality. This extra functionality can be realized either in hardware, software, or both. No matter in which way the reliability functionality is realized, it would incur extra resources either in terms of the gate number (more hardware), or operation number (more software). Therefore, the resultant extra resources must introduce more execution time, area, power, and/or energy consumption.

Inevitably, the cost of the reliability functionality contradicts the design constraints (e.g., execution time, chip area, power, and energy) that come inherently with the application-specific nature of embedded systems. Consequently, it becomes quite difficult to keep high reliability while maintaining the design constraints. Traditional reliability enhancement techniques are very likely to yield overdesigned solutions since they are mostly monotonously treated (on or off for the entire system and the entire run-time).

Therefore, for reliability-oriented embedded systems, the design methodology must evolve to cope with the underlying contradictions (constraints). Firstly it is necessary to obtain the application-specific trade-off between reliability and other performance metrics (from execution time to power consumption) at the design time. Secondly, the system should be flexible to adapt its functionality at run-time to achieve the best overall performance (dependent to the specific application), given the reliability and other traditional constraints (perhaps with different priorities). Notwithstanding, most of the existing reliability solutions are designed from a general perspective, and thus lack the mechanism to customize the reliability functionality considering the underlying application.

Recent research [1], [13] has studied on estimating the vulnerability to soft errors by static analysis (or model). In the scope of instruction set processor systems, one derivative of those techniques is proposed for enabling soft-error vulnerability estimation for each instruction in a program at design time. It has been used in several software-implemented soft-error countermeasures [15], [16], which reportedly show an improvement in reliability at the software program level.

Fig. 1 plots the Instruction Vulnerability Index (IVI) values of the instructions that are executed during the function of adpcm_coder

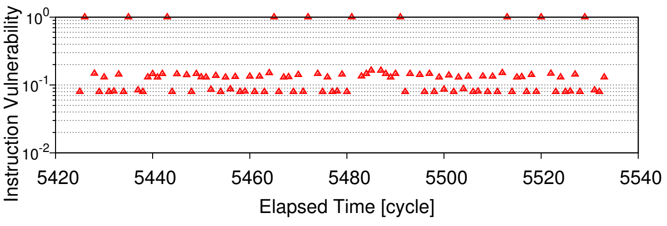


Fig. 1. Vulnerability of instructions at run time (the trace begins from PC=0x1F0 on the left to 0x22C on the right)

in application named ADPCM on a processor (using the technique in [15]). It shows that in 100 clock cycle execution, there are only a few (roughly 10%) instructions having significant probability to introduce soft errors. The rest of instructions are ten times less vulnerable. This observation provides unique evidence suggesting the inefficiency of pessimistically adding reliability operations for all the instructions, especially when considering stringent design constraints. Ultimately this finding supports our motivation of using static knowledge that is gained from design time to smartly guide the reliability functionality, which works at run time. In this way, we can flexibly choose the trade-off between reliability and other metrics (throughput, power, and area). In order to realize this concept, the reliability functionality should be designed in such a way: 1) capable of exploiting both temporal and spatial redundancy, and 2) can perform the recovery functionality at an optimal effort level for a particular run-time application.

In this paper we propose a HW/SW reliability solution, namely CSER, which focuses on soft-error recovery for application specific instruction set processors (ASIPs)[1]. To the best of our knowledge, our solution is the first to realize the configurable HW/SW error-recovery functionalities in the processor architecture, and to allow them serve as intelligent handlers that leverages the reliability of the system at the instruction and basic-block levels. The proposed solution exploits the characteristics of the recovery functionalities and forms the optimized recovery schemes at varied levels of granularity, to increase efficiency. Guided by the particular configuration that is derived from design input information, the instructions on the CSER architecture would perform the deployed recovery operations in one of the two types of the state:

Error-Resiliency: commit the additional operations so that to allow the system to be able to recover from errors. The additional operations are selected w.r.t. the configuration and integrated into the execution of instructions through instruction customization.

Low-Cost: throttle the additional operations, thus to allow tuning the run-time performance or other metrics up again, while sacrificing reliability (ability to recovery).

Our contributions are as follows:

(1) A HW/SW instruction-based configurable error-recovery system that supports tailored soft-error recovery functionality that is determined at design time.

[1]A representative type of embedded processors that supports customizable instruction set and are designed through architectural description language (ADL) to HDL synthesis tools (e.g., ASIPmeister and Xtensa [12]).

(2) An application-specific framework using static Soft Error vulnerability estimation to guide the run-time error-resiliency for flexible design trade-off.

II. Related Work and Background

Traditional error-resilient techniques are implemented in pure hardware or software for general purpose platforms. They *do not consider application specific characteristics*, and therefore have difficulty in providing flexible error-recovery schemes given design constraints. HW techniques introduce no modification to SW, rather increase the number of gates in the system. In contrast, SW ones do not enlarge the system but raise the number of instructions. This means the system must run more code and thus have performance degradation and more dynamic power/energy consumption. Moreover, SW ones cannot touch the resources/state other than those transparent to the program in the system.

On the HW side, Cache-Aided Rollback Error Recovery [7] modifies the cache replacement policy and utilizes the cache as a buffer to hold the unsure data for computation. This technique is sound in not incurring a new buffer at the cost of the memory traffic (especially with the associativity lower than 4-way). Afterwards, Sequoia [2] solves HW error recovery in multi-processor systems; whereas SWICH [20] improves in the rollback window size. On the SW side, Software-Implemented Fault Tolerance [17] optimizes the compiler to generate additional lines of code that allow the system to use majority voting algorithm to recover errors with almost doubled code; while TRUMP [17] uses AN-code in replacement of majority-vote in arithmetic instructions to reduce the overhead. Because AN-code does not propagate through many logical operations, the applicability of this technique is limited. Some HW/SW fault tolerant techniques [10] are proposed to enhance the systems with the architecture that allow instructions to manage checkpoint and recovery. Those pieces of research above do not include the consideration of configuring the effort level of the technique to provide flexibility to system designers. In addition, they do not include any estimation model to gauge the application-specific vulnerability so that to help optimize the technique.

Recently there are a few pieces of research [5], [8], [11], [15], [21] exploring with the incentive that reliability-oriented design can be greatly improved in efficiency by using adequately advised relaxation. The approaches [8], [11], [15], [21] differ with ours in such aspects as: 1) they do not focus on error-recovery, and 2) their techniques are software-only and compiler-based. Due to this difference, their techniques still face such issues as the code size increase and unreachable state/resources limited to the level of techniques (software). On the other side, the approach in [5] addresses the recovery problem by utilizing the statistically idempotent property of the program to reduce the recovery cost and provide flexibility in the degree of fault tolerance. Beside the difference in the theoretic proof that derives the reduction criteria, this approach differs in the second aspect stated above, in comparison to ours.

A. Basic Notions and Preliminaries

In this paper, a few basic notions are applied and they are defined as follows:

Instruction: The fundamental node of a computing task (i.e. program). Thereafter, we denote **static instructions** as the elements of instruction set; whereas **dynamic instructions** as those, equivalent to program counter (PC) values, in the code generated by compilers.

Basic Block: The high-level node consisting of a number of instructions and constitutes a computing task. One block in principle has only one entrance and one exit instruction, and does not include control flow instructions in the middle.

Architectural State: The state of the architectural components in a processor. Typical architectural components are registerfile, data memory, and other special registers (e.g. status registers).

Functionality: A group of operations. Each operation is either a data transfer or processing of functional unit.

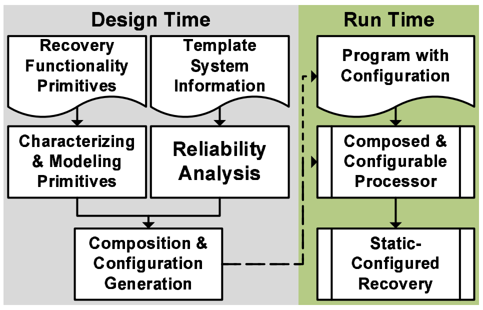

Design Time
Run Time
Recovery Functionality Primitives
Template System Information
Program with Configuration
Characterizing & Modeling Primitives
Reliability Analysis
Composed & Configurable Processor
Composition & Configuration Generation
Static-Configured Recovery

Fig. 2. CSER Overview

Composition: Composing a functionality primitive denotes the process of allocating the underlying operations to the determined static instruction. E.g., if Instruction A (static) is composed of Operation B, then Operation B can be supported by the occurrences of Instruction A.

Configuration: Configuring a functionality primitive denotes the process of activating and deactivating the underlying operations. E.g., if Operation B is composed in Instruction A, but not configured at a dynamic occurrence (PC=0x100), then Operation B would not be performed at that occurrence.

Moreover, in this study we mainly consider Single-Event Upset (SEU), which is reported as the most common soft-error effect in system level [3]. To facilitate the description of our approach, we assume and employ two primitive error-recovery functionalities that represent the common "naive" error-recovery techniques (refer to Sec. III-A) that take different kinds of redundancies (space and time). In CSER approach, they are characterized, modeled, selectively incorporated into the instruction set via HW/SW integration, and finally configured at run time (refer to Sec. III-B).

III. CSER Approach

Fig. 2 shows the overview of the proposed approach. Our approach comprises two main components: 1) design-time optimization, and 2) run-time operation. Design-time optimization principally composes the recovery functionality primitives that can recover the system from soft-error effect. It also generates the configuration on the selected error-recovery functionality primitives. This process is based on the provided template system information such as instruction set architecture and design constraint. Finally it produces the processor (HDL model) that is composed of primitives and also available to perform the configuration at run time.

A. Characterizing Primitives

Error recovery functionality primitives (summarized in Fig. 3) are briefly described as follows:

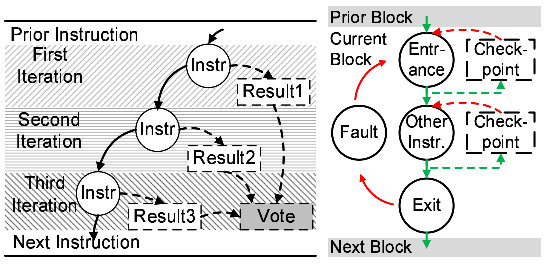
Instruction Replay and Vote (IRV)[2] performs instruction-level recovery (Fig. 3(a)), via iterating the selected instruction two more times, and then voting for the correct instruction result from the majority. This mechanism basically invokes two more redundant iterations for a particular objective instruction.

Block Checkpoint and Recovery (BCR)[3] fulfills block-level recovery, by saving the original values of the architectural state at every update in a basic block, and then undoing the update in case the block's executions are faulty (Fig. 3(b)). In comparison to IRV, BCR is a coarse-grained recovery technique, where the granularity of protection is the basic block.

These two primitives demonstrate great orthogonality in their features and thus the costs induced. This orthogonality suggests a complementary benefit in mixing the two primitives in one solution and using them in a configurable way. Note CSER is not limited to

[2]IRV comes from the notion of triple modular redundancy (TMR) which is widely applied in other bodies of research [18], [19].

[3]Existing HW/SW techniques of BCR can be seen in [10], where the error is assumed to be checked at the end of the blocks.



Primitive	Granularity	Redundancy	Principal Cost
IRV	instruction level	time	clock cycles
BCR	block level	space	logic gates

(c)

Fig. 3. Error Recovery Primitives: (a) Instruction Replay and Vote; (b) Block Checkpoint and Recovery; (c) Comparison

these two primitives, other primitive techniques can also be applied in the framework of CSER approach.

1) Modeling the Cost of IRV: The cost of this technique lies in two parts:

Execution Time can be estimated to be bounded by $C_{time} = 3 \cdot T_{instr} \cdot N_{instr}$ where N_{instr} stands for the number of the executions of protected instructions, and T_{instr} denotes the execution time (in clock cycles) of one arbitrary instruction.

Additional HW Structures are introduced as well (mainly including two more registers and a few multiplexors). This cost is twofold and can be formulated as $C_{HW} = C_{static} + C_{dynamic}$ where C_{static} is the static cost of the additional HW structures in terms of area and static power, and $C_{dynamic}$ is the dynamic cost that is a variant value, in terms of dynamic power or energy, dependent to the run-time behavior of the processor. For example, suppose we put replay operations that are realized by additional HW. This augmentation adds a number of gates into the system; however if this operation is not behaved by the system at run-time, then the dynamic HW cost (i.e. dynamic power and energy) is very limited (negligible). Basically, we can assume $C_{dynamic}$ linearly increases with N_{instr}. Because the HW structure increase is relatively small, the major cost of IRV is in execution time (or performance).

2) Modeling the Cost of BCR: BCR involves additional HW structures such as checkpoint buffer (the container of checkpoint), pipeline flip/flops, and multiplexors. As a result, BCR costs mainly on HW side (spatial redundancy). The cost of BCR is also in two parts. Fault-free part is more dominant since the processor is in the fault-free scenario in most of the service time. This part of cost has three major factors from HW aspect: constant HW cost $C_{constant}$, variant HW cost $C_{variant}$, and dynamic HW cost $C_{dynamic}$. The former two factors constitute the static HW cost. Their relationship can be expressed as $C_{HW} = C_{constant} + C_{variant} + C_{dynamic}$.

Constant HW Cost can be break down as $C_{constant} = \sum_{comp \in ad\_HW} C_{comp}$ where $comp$ is one of the additional pipeline components ad_HW that are induced by the underlying technique.

Variant HW Cost is $C_{variant} \equiv C_{checkpoint} \propto \max_{block \in program} N_{update\_block}$ where $C_{checkpoint}$ is the checkpoint operation number and buffer size, and its size is bounded by the maximum update number, N_{update_block}[4] in a block.

Dynamic HW Cost can be described as $C_{dynamic} \propto \sum_{block \in program} N_{update\_block} \cdot Freq_{block}$ is in terms of dynamic power or energy, similarly to IRV. However, there is a difference that the key factor to $C_{dynamic}$ in BCR is N_{update_block}.

The reason is that dynamic cost is correlated to the number of additional operations the processor performs, which is dominated by checkpoint operations (quantified by N_{update_block} in a block). To summarize, in BCR, both $C_{variant}$ and $C_{dynamic}$ are significantly influenced by N_{update}; however their correlations are different.

[4]Precisely, the update means unique update in the case of register file [10].

Algorithm 1: Instruction Level Selection Heuristic

```
1  Initialize $C$: set the capacity/constraint
2  Initialize $\Delta C$: set the capacity modifier constant
   /* Initialize profit table $P = \{p_1, p_2, \dots, p_n\}$ */
3  for $v \in V$ do
4      $p_v \leftarrow \mathcal{P}(v) \leftarrow \mathcal{IVI}(v) \cdot \mathcal{F}req(v)$
   /* Sort the instructions by their profits */
5  $Sorted\_List = \emptyset$
6  for $v \in V$ do
7      $key \leftarrow p_v$
8      $Sorted\_List \leftarrow SortList(key, v, ascending\_order)$
   /* Select from the most profitable */
9  $S = \emptyset$
10 for $v \in Sorted\_List$ do
11     $c_{remain} \leftarrow C - \Delta C$
12     if $c_{remain} > 0$ then
13         $C \leftarrow c_{remain}$
14         $S \leftarrow v$
15         $p_{total} \leftarrow p_{total} + p_v$
16     else
17         break
18 return $(S, p_{total})$
```

B. Composition and Configuration

Provided the characterization of recovery functionality primitives, the composition and configuration process is equivalent to solving a selection problem given the constraints. This problem can be simplified to a *knapsack problem* where the capacity denotes the tolerable design constraints. The objective of the solution is to maximize the reliability profit while satisfying the cost constraint. We firstly propose an instruction level selection with IRV solely, which mainly targets on execution time and resultant power consumption. For exploring the efficacy with coarser and more spatially redundant recovery techniques, we propose a heuristic for optimizing BCR for high reliability while satisfying the constraints. At last, we propose to use a hybrid technique IRV/BCR, for exploiting the benefit from the orthogonality between these two primitives. For example, replacing a number of blocks under BCR protection with a few instruction replays can trade off area or power (due to the $C_{checkpoint}$) to execution time.

One fundamental step before this process is the **Reliability Analysis**, which provides the reliability profit function. We adopt one instruction-based estimation tool [16] to obtain the basic parameter, instruction vulnerability index (IVI). To reflect the dynamic behavior of the target program, we also use profiling to get the frequency of the dynamic instructions. Therefore we have a profit function $P(instr) = IVI_{instr} \cdot Freq_{instr}$ where $Freq_{instr}$ indicates one dynamic instruction's occurrences. The value of the profit indicates the influence of the underlying instruction to the system under the pressure of soft error effect.

1) Instruction Level Selection: Instruction level selection (ILS) essentially selects a set of instructions that are to be composed of IRV statically and configured to perform IRV at run-time. This set S is a subset of the node in the program graph $G = (V, E)$, where a node v_i is a dynamic instruction (equivalent to program counter value) of instructions V that are derived from the assembly code.

Problem: Select the subset of instructions from the given set of instructions V in G, considering each instruction as basic element v_i $(v_i \in V, i = 1, \dots, n)$, each instruction having a reliability profit $P(v_i)$ in case that it performs IRV, and weight (or cost) function $W(v_i)$.

Objective: Determine the subset S so as to achieve

$$p_{total} = maximize \sum_{v \in S} P(v)$$

Constraint: The total cost of the selected instructions must satisfy

$$c_{total} = \sum_{v \in S} \Delta C \leq C$$

Algorithm 2: Block Level Selection Heuristic

```
1  Initialize $C$: set the capacity/constraint
   /* Initialize block profit table $P = \{p_1, p_2, \ldots, p_n\}$ */
2  for $b \in B$ do
3      for $v \in b$ do
4          $GetProfit()$ /* Line 3-4 in Alg. 1 */
5      $p_b \leftarrow P_{block}(b) \leftarrow \sum_{v \in b} p_v$
   /* Initialize block cost table $W = \{w_1, w_2, \ldots, w_n\}$ */
6  for $b \in B$ do
7      $w_b \leftarrow W(b) \leftarrow N_{update\_b}$
   /* Dynamic programming */
8  for $i = 1$ to $n$ do
9      for $c = 0$ to $C$ do
10         $c_i \leftarrow \Lambda(w_i)$
11         if ($c_i \leq c$) and ($p_i + K[i-1, c-c_i] > K[i-1, c]$) then
12             $K[i, c] \leftarrow p_i + K[i-1, c-c_i]$
13             $sel[i, c] \leftarrow True$
14         else
15             $K[i, c] \leftarrow K[i-1, c]$
16             $sel[i, c] \leftarrow False$
   /* Examine selection and fill $S_{block}$ */
17 $c_{remain} = C$
18 $S_{block} = \emptyset$
19 for $i = n$ downto 1 do
20     if $sel[i, c_{remain}] = True$ then
21         $S_{block} \leftarrow b_i$
22         $c_{remain} \leftarrow c_{remain} - c_i$
23 $p_{total} \leftarrow K[n, C]$
24 return ($S_{block}, p_{total}$)
```

Based on the characterization of IRV, it can be assumed that most of the instructions' costs are highly similar [5]. Hence, in order to simplify the problem (knapsack problem), which is NP-hard, we make one assumption that the cost function maps to a constant value ΔC. With this simplification, the solution of instruction level selection problem can be solved within a reasonable time (described in Alg. 1), which follows a greedy search. At first, the algorithm initializes the table that contains the profits of all the dynamic instructions (Line 3-4). This table is then used to make a sorted list of instructions for selection (Line 5-8). Then, the algorithm greedily puts the instructions one by one from the top of the list into a new set, until the capacity is met (Line 10-17).

2) Block Level Selection: Block level selection (BLS) considers composing and configuring BCR, which is more complex than that with IVR. The reason is in two aspects: 1) the grains are more different to each other (blocks have very different constituent instructions and behave very differently), thus they introduce fairly distinctive recovery operations; 2) the additional HW structure cost is strongly dependent to the block's behavior (i.e. the number of registerfile and memory write references), due to the feature of BCR.

Based on the instruction model that is used in ILS (Sec. III-B1), we further derive the block model. For each block in the program, it has two important parameters:
Block Profit is defined as the sum of the profits for instructions in the block and given by the profit function $P_{block}(b) = \sum_{i \in b} P_{instr}(i)$.
Block Cost is dominated by the number of the registerfile and memory write references that would require checkpoint operations and buffer space (i.e. $C_{variant}$ and $C_{dynamic}$), which is reflected by the cost function $W(b) \equiv N_{update\_b} = N_{reg\_b} + N_{mem\_b}$ and is elaborated in Sec. III-A2.

Problem: Select a subset of basic blocks from the given set B in the program $G = (B, D)$, where a node is a block $b \in B$ and D is the set of dependencies between blocks, considering the capacity C, the reliability profit function $P(b)$ and the cost function $W(b)$.

Objective: Determine a subset S_{block} so as to satisfy

$$p_{total} = maximize \sum_{b \in S_{block}} P(b)$$

[5]Exceptions can happen in some architectures, where the multiplication and division instructions take longer cycles in execution stage. In this case, the algorithm for BCR can be applied.

Algorithm 3: Hybrid Level Selection Heuristic

```
1  Initialize $C_1$: set the capacity/constraint
2  Initialize $C_2$: set the capacity/constraint
3  Initialize $\Delta C_{IRV}$: set the capacity modifier constant for IRV
   /* Initialize block profit and cost tables */
4  $InitBLS()$ /* Line 2-7 in Alg. 2 */
   /* Block Level Selection */
5  $(S_{BLS}, p_{BLS}) \leftarrow GetBLS()$ /* Line 8-24 in Alg. 2 */
   /* Reduced Instruction Level Selection */
   /* Initialize instruction profit table */
6  $InitILS()$ /* Line 3-4 in Alg. 1 */
7  $G' = G - S_{BLS}$
8  $(S_{ILS}, p_{ILS}) \leftarrow GetILS()$ /* Line 5-18 in Alg. 1 */
9  $p_{total} \leftarrow p_{BLS} + p_{ILS}$
10 return ($S_{ILS}, S_{BLS}, p_{total}$)
```

Constraint: The total cost must be bounded by

$$c_{total} = \sum_{b \in S_{block}} W(b) \leq C$$

This selection problem can be solved by using *dynamic programming*. Our heuristic is described in Alg. 2. For the sake of brevity, we assume the total number of blocks in the program is n, and they are enumerated from b_1 to b_n. We also assume the cost mapping $\Lambda : w_i \to c_i$, where w_i is the value of $W(b_i)$. Precisely, this mapping abstracts out the detail of deriving the total cost c_i while including b_i that has a weight w_i. We use a common dynamic programming data structure which is a memoization table $K[i, c]$ recording the maximum profit achievable with capacity c and blocks $1, \ldots, i$. In addition, we use an auxiliary boolean table $sel[i, c]$ whose value is True if b_i is selected in $K[i, c]$ and False otherwise. This table is used to derive the S_{block} after dynamic programming is finished. The algorithm firstly initialize the profit table and cost table via the profit and cost function respectively (Line 2-7). Then, these two tables are used to select the optimal subset for each subproblem from the base cases, until reaching the ultimate case, i.e. $c = C$ and $i = n$ (Line 8-23).

3) Hybrid Level Selection: The main motivation behind using hybrid level selection (HLS), i.e. composing and configuring IRV/BCR, lies in the benefit from orthogonality. One typical exemplar scenario is that, the execution time constraint solely prohibits the composition and configuration from achieving broader coverage of IRV, i.e., more reliability profit, while there is still a rich quota on area constraint. Apparently, using hybrid recovery promises further gain in reliability, exploiting unused area quota, within the multi-dimensional constraints.

Problem: Given design constraints $\{C_1, C_2\}$, two reliability profit functions $P_{instr}(v)$ and $P_{block}(b)$, and the cost functions $W_{instr}(v)$ and $W_{block}(b)$, the problem is to determine the selections of the subsets S_{ILS} and S_{BLS}.

Objective: The selected subsets must satisfy

$$p_{total} = maximize(\sum_{v \in S_{ILS}} P_{instr}(v) + \sum_{b \in S_{BLS}} P_{block}(b))$$

Constraint: The total cost should be restricted by

$$\forall i \in \{1, 2\} : c_{total\_i} = c_{ILS\_i} + c_{BLS\_i} \leq C_i$$

Alg. 3 describes the solution to this problem, which essentially is optimizing ILS-BLS partition across the dynamic instructions G. We use a good (not necessarily optimal, but with significantly reduced complexity) partitioning heuristic that can be summarized in two sequential steps:
Partitioning Coarse Grains: select the set S_{BLS} of blocks of instructions protected by BCR from graph $G(V, E)$ with the constraint C (follow Alg. 2).
Partitioning Fine Grains: select the set of instructions S_{ILS} protected by IRV from sub-graph $G' = G - S_{BLS}$ with the restrained constraint $C' = C - C_{BLS}$ (follow Alg. 1).

The finer-grained step (i.e. ILS) is sequenced at last. The reason is twofold: it has a higher chance than the coarser one (i.e. BLS) to avoid including "bad grains", which does not have a desirable

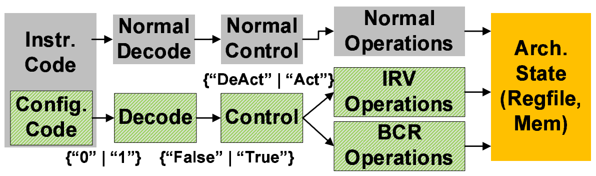
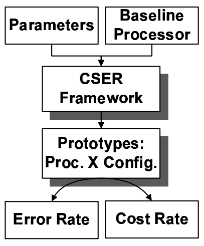


Fig. 4. CSER Architecture (Green: CSER, Grey: Template/Normal)



(a)

Parameter	Value
Template	Sparc-V8 16-Window Integer
Benchmark	ADPCM,CRC,SHA,SUSAN [6]
Constraint	3%,5%,10%,15%,20%
Clock Period	10 ns
Fault Model	Single-Bit Flip
Repetition	1000 (1/trial)
Technology	TSMC 65 nm
Reference	Baseline,SWIFT [17],Reli [10]
HDL Simulator	ModelSim SE6.6d
RTL Synthesis	Synopsys DC&PT

(b)

Fig. 5. Experiment: (a) Flowchart; (b) Parameter Value

profit/cost ratio; and the proportion of "bad grains" increases after a number of "good grains" are picked at the previous step. Note that on the ground of CSER framework, more optimal hybrid selections can be achieved if provided more complex algorithms, e.g. simulated annealing (typically used for HW-SW partitioning [4]); However due to the space limit, our discussion here only covers a simpler version.

C. *CSER Architecture*

CSER architecture (shown in Fig. 4) is generated by modifying the template architecture with the consideration on the result of CSER approach. This modification generally includes: 1) additional operations from the corresponding error-recovery functionality primitives are merged into the relevant instruction ADL model, which is equivalent to control data flow graph (CDFG); 2) the relevant instruction (including the error recovery operations) is then enhanced with the functionality of configuration.

Merging the operations into template instructions considers pipeline stages and conflicts (w.r.t data and structure). The detail of this part can be seen in other literatures that follow the same implementation style (ADL-to-HDL) [14]. For the sake of brevity, we mainly discuss the functionality of configuration here. The main components in this functionality are described:

Configuration Code: One bit for each primitive is added into the code of the program at compile stage. This bit value denotes the configuration or the scheme for the error recovery operations in the instructions. The unused instruction code bits are potential configuration bit.

Decoding: The configuration bit is decoded at the decoding stage[6], consistent to normal instruction code. The configuration information ("1" for "True" and "0" for "False") is thus brought into the pipeline execution (control unit).

Control: The control unit adapts to the configuration information and send out corresponding control signals to activate (if "True") and/or deactivate (if "False") the underlying operations.

IV. Experiment Setup

Fig. 5(a) depicts the experiment flow. Given the parameters and baseline system, CSER processors are generated in processor vs. configuration pairs. Each pair realizes a recovery scheme for an benchmark application (e.g. ILS targeting 10% cost for ADPCM). The complete experiment consists of two procedures: 1) fault injection test to obtain the error rate, and 2) logic synthesis and fault-free simulation to yield the cost in design constraints.

Fig. 5(b) describes the parameters used in the experiment. The template architecture (i.e. baseline reference) resembles 16-window integer SPARC-V8 instruction set. The benchmark consists of four

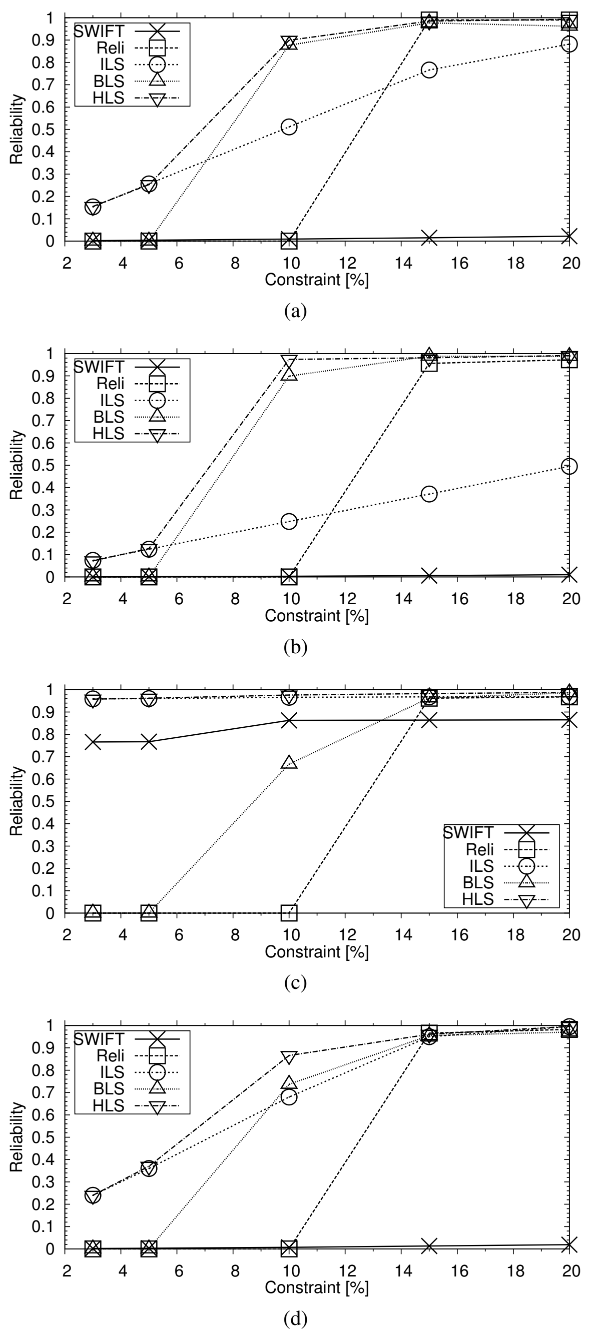


Fig. 6. Result: (a) ADPCM;(b) CRC32; (c) SHA; (d) SUSAN

applications from MiBench suite [6]. We use a set of constraints ranging from 3% to 20% to represent quantification of design constraints against the baseline. HDL simulation is conducted with ModelSim SE simulator[7], while RTL synthesis (RTL to Netlist) is performed with Synopsys Design Tools (DC and PT)[8].

Fault injection test and fault-free simulation is conducted in HDL simulation at RTL level. The fault model is single-bit flip since it represents the most common soft error occurrence, also named as single event upset (SEU). The test has a series of repetitions of injections (namely Monte Carlo Simulation). In each repetition (the number of repetitions can be acquired using the method in [9]), one fault is injected in to the system at an instruction at first (similar to

[6]Usually the second stage in single-issue architectures.

[7]http://model.com/

[8]http://synopsys.com/

the method used in [10]). The faulty instruction is randomly chosen and the probability is affected by the area and time of the instruction (follows the assumption in [15]).

The prototypes are measured in the forms of gate-netlist that is yielded by RTL Synthesis with TSMC 65nm technology library. In order to fully envision the feature of CSER, we use full-scope IRV and BCR, which always perform the respective recovery operations, as two reference techniques. They represent two state-of-art error recovery techniques (full-scope IRV as the HW/SW equivalent of SWIFT-R [17], and BCR as the resemblance of Reli [10]).

V. Result and Discussion

Fig 6 shows the results of reliability against cost rates w.r.t. the benchmark ADPCM (a) to SUSAN (d), with CSER and reference prototypes. In order to effectively express the application's characteristics, the kernel functions are mainly exercised in the fault injection trials.

The reference SWIFT here is our HW/SW version of SWIFT-R [17], which is a full-scope IRV notwithstanding CSER approach. We assume, SWIFT would protect the instructions from beginning until the constraints are used up, if the constraints are not sufficient for the full-coverage protection. Reli is SPARC implementation of Reli [10], which is a full-scope BCR notwithstanding CSER approach. We assume Reli would not be equipped to the processor, in case the constraints cannot suffice the cost. ILS, BLS, and HLS are CSER processors using three respective heuristics in our approach. The x-axis is the choice of design constraints or cost rate (20% means 20% more cost against baseline). The y-axis is the relative value of reliability index $R_i = (1/E_i) - R_{baseline}$ against the baseline ($R_{baseline}$), where E_i is the error rate, reflected by the number of effective errors that are not recovered.

In comparison to SWIFT (full-scope IRV), ILS demonstrates a very gradual increasing profile in CRC32 (e.g., from about 0.1 to 0.5 in Fig. 6(b)) as the constraints rise. In other three cases, the reliability of ILS approach saturates more quickly than SWIFT at 15% constraint, which shows the effect of fine-grained composition and configuration. In contrast, SWIFT mostly spends the limited constraints on low-profit instructions. One exception can be seen in FIg. 6(c) where both approaches achieve more than 0.8 reliability at all the constraint choices. The reason is that SHA kernel function has a small size and most of the dangerous instructions are executed at the front part. Overall ILS leads 1x to 9x against SWIFT.

In comparison to Reli (full-scope BCR), BLS shows optimized contribution to reliability (7x to 9x) in all the applications in the scenario of 10% constraints. This result is because the cost of configuring the basic blocks to perform BCR in the relevant applications are costly and the derived system costs are around 10%, where BLS manages to configure a set of low-cost blocks, while Reli is not applicable.

HLS (configuring IRV/BCR) shows the highest reliability profile (0.1 to 0.9) in all the applications. It can be seen that within 3% and 5% constraints, IRV is the principal contributor to the reliability, which compensates the inefficiency of using BLS alone in undesirable scenarios. The benefit of hybrid usage of two primitives is observed when the constraints rises to 10%, in comparison to using ILS alone. Overall, CSER approach at most can achieve approximately 9x reliability especially in the extreme constraint scenarios.

VI. Conclusion

In this paper, we have proposed CSER approach that can allow finely configured error recovery functionality targeting the given design constraints. Our methodology includes two primitive error recovery techniques, and three application-specific optimization heuristics, which generate the optimized composition and configuration based on the characteristics of primitive techniques, static reliability analysis and design constraints. The resultant processor is composed of selected primitive techniques, and configured to perform at runtime accordingly to the optimized recovery scheme determined at design time. The experiment results have shown that CSER can more strategically enhance the reliability of the system while maintaining the given constraints, especially in restricted cases (e.g., almost 10x reliability below 10% constraints).

Acknowledgment

The authors would like to thank Florian Kriebel (Karlsruhe Institute of Technology) for his effort in experiments for obtaining the instruction vulnerabilities.

References

[1] H. Asadi, M. B. Tahoori, and C. Tirumurti. Estimating error propagation probabilities with bounded variances. In *DFT*, pages 41–49, 2007.

[2] P. Bernstein. Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing. *Computer*, 21(2):37 –45, feb. 1988.

[3] A. Dixit, R. Heald, and A. Wood. Trends from ten years of soft error experimentation. In *SELSE5*, 2009.

[4] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, 1993.

[5] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 398–409, New York, NY, USA, 2011. ACM.

[6] 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.

[7] 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.

[8] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, New York, NY, USA, 2012. ACM.

[9] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 502–506, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[10] T. Li, R. Ragel, and S. Parameswaran. Reli: Hardware/software checkpoint and recovery scheme for embedded processors. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 875 –880, march 2012.

[11] B. H. Meyer, N. J. George, B. H. Calhoun, J. Lach, and K. Skadron. Reducing the cost of redundant execution in safety-critical systems using relaxed dedication. In *DATE*, pages 1249–1254, 2011.

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

[13] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 29 – 40, dec. 2003.

[14] 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.

[15] S. Rehman, M. Shafique, and J. Henkel. Instruction scheduling for reliability-aware compilation. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1288 –1296, june 2012.

[16] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 237–246, New York, NY, USA, 2011. ACM.

[17] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27:36–47, January 2007.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[19] X. She and P. Samudrala. Selective triple modular redundancy for single event upset (seu) mitigation. In *Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on*, pages 344 –350, 29 2009-aug. 1 2009.

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

[21] N. Wang and S. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188 –201, july-sept. 2006.