Fine-Grained Checkpoint Recovery for Application-Specific Instruction-Set Processors

Tuo Li, *Member, IEEE*, Muhammad Shafique, *Senior Member, IEEE*, Jude Angelo Ambrose, *Member, IEEE*, Jörg Henkel, *Fellow, IEEE*, and Sri Parameswaran, *Senior Member, IEEE*

Abstract—Checkpoint recovery (CR) is a classic fault-tolerance technique, which enables computing systems to execute correctly even when affected by transient faults. Although a number of software and hardware based approaches for CR does exist, these approaches usually are either too large, too slow, or require extensive modifications to the software and the caching/memory schemes. In this paper, we propose a novel CR approach, which is based on re-engineering the instruction set of a target processor. We take the base instruction set and augment the native micro-operations, i.e., an architectural description language (ADL), with additional micro-operations to perform checkpointing at the granularity of basic blocks. The recovery mechanism is realized by three custom instructions, which can undo the corruptions caused by transient faults during instruction execution, including the values of general-purpose registers, data memory, and special-purpose registers (PC, status registers, etc.), which were incorrectly modified. Our checkpoint storage is sized according to the application program executed. The experimental results show that our approach degrades the system performance by just 0.76 percent when there is no fault, and introduces an area overhead of 44 percent on average and 79 percent in the worst case. During the fault injection test with the benchmark applications, the recovery took just 62 clock cycles (worst case).

Index Terms—ASIP, checkpoint recovery, reliability

1 Introduction

Computing systems must be protected against transient faults, so as to guarantee that the computation running on the system can be relied upon constantly [1]. In particular, for embedded systems, transient faults have been identified as one of the key reliability issues [2]. Mitigating transient faults first needs to *detect* errors (this has been intensively studied in [3], [4], [5]) and second to *recover* the system when errors have been detected.

Checkpoint recovery (CR) has been studied as a viable solution for error recovery of transient faults [6]. Given the ever-increasing impact of transient faults, in recent years, CR has been investigated for embedded systems [7]. CR recovers the current computing process (program) by using the most recent checkpoint. A checkpoint is generated periodically, which consists of data that keeps a copy of the error-free system states. Depending on the checkpointing mechanism and implementation style, the *checkpoint data size* and *checkpoint period* (interval) can be varied. CR requires additional resources, in terms of hardware and/or machine cycles, for both generating checkpoints (i.e., state capture) and performing recovery (i.e., rollback). Focusing on general-purpose and high-performance computing platforms (e.g., microprocessor and data center), recent studies have discussed CR techniques from two aspects: (1) *Software-based CR* introduces redundant program and typically has large code size or considerable fault-free performance overhead [8]; and, (2) *Hardware-based CR* introduces specific modifications to the microarchitectures (e.g., cache or memory) and redundant hardware blocks in a processor, and are non-systematic and inflexible [9].

Embedded systems usually must satisfy stringent design constraints, such as performance, area, and power. Hence, *a viable CR implementation for embedded systems have to be small, fast, and energy efficient*. Notwithstanding, adopting the existing CR techniques is very likely to worsen time, area, and energy constraints considerably. Application-specific instruction-set processor (ASIP) has been widely adopted in a variety of embedded application domains, such as communication, multimedia, and so on. ASIP design, including *instruction set customization and extension* [10], has been intensively studied and successfully practiced, with respect to the design constraints such as performance, power, and area, in both industry (e.g., `Cadence/Tensilica`[1] and `Synopsys/ARC`[2]) and academic circles [11].

Contribution. In this paper, we propose a transient-fault countermeasure called Reli, which is a fine-grained CR approach for ASIP-based embedded processors. To the best

- *T. Li and S. Parameswaran are with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia. E-mail: {tuol, sridevan}@cse.unsw.edu.au.*
- *M. Shafique is with Vienna University of Technology (TU Wien), Vienna 1040, Austria. E-mail: muhammad.shafique@tuwien.ac.at.*
- *J.A. Ambrose is with Canon Information Systems Research Australia, 5 Talavera Rd, Macquarie Park, NSW 2113, Australia. E-mail: Angelo.Ambrose@cisra.canon.com.au.*
- *J. Henkel is with the Karlsruhe Institute of Technology, Karlsruhe D-76131, Germany. E-mail: henkel@kit.edu.*

Manuscript received 22 Dec. 2015; revised 26 Aug. 2016; accepted 29 Aug. 2016. Date of publication 6 Sept. 2016; date of current version 17 Mar. 2017. Recommended for acceptance by P. Girard.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2606378

1. http://www.tensilica.com/
2. http://www.synopsys.com/IP/PROCESSORIP/ARCPROCESSORS/

of our knowledge, Reli is the first to realize CR at the basic-block level by leveraging custom instruction design. In addition, we present an ASIP design flow based on one of the existing commercial tool (`ASIPmeister`), which can generate the RTL description of the resultant processors with Reli functionality. Consequently, the cost in terms of execution time, area, and power is reduced significantly compared to existing techniques.

Paper Organization. The rest of the paper is structured as follows. Sections 2 and 3 discuss related work and assumptions. Section 4 presents the conceptual idea of the proposed CR scheme. Section 5 elaborates the detail of instruction set architecture (ISA) implementing Reli scheme. Section 6 discusses the ASIP design flow generating Reli processors. Section 7 gives experimental setup and results, followed by a further discussion in Section 8. Lastly, Section 9 concludes the paper.

2 Related Work

2.1 Software-Based Checkpoint Recovery

Software-based CR techniques do not require additional hardware. CATCH [8] requires the modification in the compiler to insert checkpoint routine code into native code. CATCH's CR is reasonably fast but induces large checkpoint data size. In addition, the static code size is increased. UIUC reliability and security engine (RSE) [12] has a thread-level CR mechanism, which needs OS support. Later in reliability microkernel (RMK) [13], a loadable kernel module to support application-level checkpointing is proposed and implemented. Different approaches to perform software-based CR for parallel programs, such as C^3 [14], the work by Dieter et al. [15], BLCR [16], and WAG-DBI [17], have been proposed for shared memory symmetric multiprocessors.

2.2 Hardware-Based Checkpoint Recovery

Hardware-based CR techniques, also referred as Backward Error Recovery (BER), use special customization and optimization in arbitrary micro-architectural components (mainly storage components which contain process state) to implement CR.

2.2.1 Cache-Based Checkpoint Recovery

Carer [18], [19] and Swich [9] are different cache-based CR techniques. These CR techniques utilize a specially designed cache as a buffer to keep the temporary results of computation, until the check is passed. In addition, the register states are duplicated as a backup. Cache-based CR techniques needs to invalidate and reload the cache to rollback the state, which consumes millions of seconds. Implementing cache-based CR requires modifying cache replacement policy and architecture.

2.2.2 Memory-Based Checkpoint Recovery

ReVive [20] is a CR technique for shared memory multiprocessor system, which has a rollback delay of 0.82 s in the worst case for 80 ms checkpoint period. ReVive needs to modify the memory directory controller for capturing (this type of checkpointing is called "logging") the checkpoint data. The checkpoint data is located in a special space in the memory. Checkpointing and recovery are controlled by timer-interrupt and the protocol is implemented in software.

2.2.3 Separate Dedicated Checkpoint Storage

`IBM S/390 G5` [21] is equipped with a full duplication of register-file (R-unit), which serves as checkpoint data. The rollback recovery in `IBM S/390 G5` is around 1,000 clock cycles. SafetyNet [22] is a coarse-grained (100,000-cycle checkpoint period) CR approach for multiprocessor systems. SafetyNet uses separate special storages for cache and memory states. However, no exact experimental data is shown for recovery time.

2.3 Checkpoint Recovery for Embedded Systems

A two-state CR system (TsCP) was proposed in [23] for use in embedded sytems, which combines two different checkpointing schemes for fault-free and faulty scenarios, respectively. TsCP creates both uniform and non-uniform checkpointing intervals so as to reduce the number of checkpoints. TsCP requires the checkpoint data size in the order of kilo-bytes, while both checkpointing and rollback takes around 44 to 885 μs. Ocean [24] optimizes CR by targeting given cost constraints such as performance, area, and energy. The checkpoint period and checkpoint data sizes are carefully configured based on the optimal results, which are obtained from Ocean's cost model. For multicore embedded systems, P1200 [25] architecture was proposed. P1200 implements task-level CR for multiple processor clusters, communicated by NoC. Execution time overhead of P1200 is around 6 to 100 percent, depending on the task size and data size. The rollback time overhead is 12 percent.

2.4 Summary

The software-based CR techniques [8], [12], [13], [14], [15], [16], [17], [23], [24], [25] checkpoint all necessary program variables. Hence, these techniques have enormous checkpoint data, which slows down recovery. Additionally, the software-based techniques need the code size to be significantly increased for CR functions. The recovery time of the software-based CR techniques is quite large, e.g., in the order of milliseconds or even seconds.

Many of the hardware-based CR techniques [9], [18], [19], [20], [21], [22] have either relied on register, cache or memory to back up checkpoint data. The cache-based CR techniques [9], [18], [19] have to modify cache replacement policy and dependent on the inclusion of cache in the system. The memory-based CR techniques [20], [22] slows down the system because the checkpointing has to be performed in memory.

In contrast to the previous methods for general systems and embedded systems, we integrate the CR functionalities into the processor by leveraging custom instructions. As a result, in each instruction, the processor can perform checkpointing automatically. In addition, checkpoints are assigned at a far finer granularity (i.e., instruction and basic-block level) than previously considered. Hence, the performance constraint can be more easily satisfied compared to existing techniques. Moreover, the cost in terms of execution time, recovery time, area, and power is reduced significantly.

3 Fault Type and Assumptions

This paper targets transient faults only, and does not examine permanent faults. We assume a single bit-flip happens

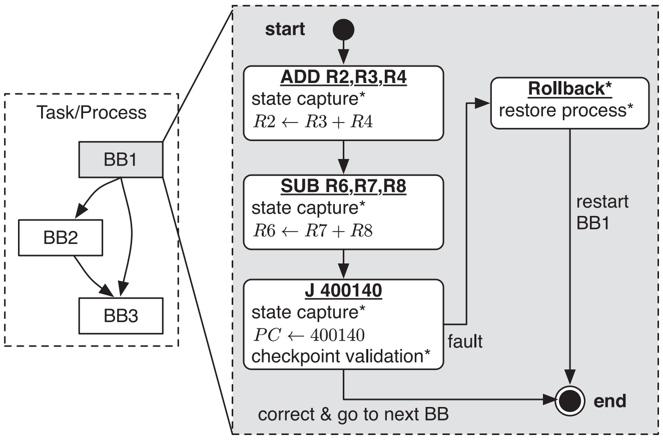


Fig. 1. Overview of RELI functionality. Symbol "*" denotes the underlying operation or state is related to RELI functionality. The grey frame on the right is a state-transition diagram showing the operations within the underlying basic block (also in grey). BB1 is assumed to have three instructions originally.

during the execution of a program. This assumption has been commonly established in the studies targeting transient faults [26]. As error-correcting codes (ECC) [27] can be applied to register-file and memory, we assume that bit-flips in register-file and memory are self-correcting. A typical CR technique can mitigate three types of faults, which are manifested at the architectural level:

- *Data fault*—when incorrect data is written to the registers or memory;
- *Address fault*—when the data is written to incorrect location of the registers or memory; and
- *Control fault*—when an incorrect operation is executed (e.g., ADD instead of SUB).

If these faults can be detected by any means, then CR method described in this paper can be activated to restore the register and memory states, which were corrupted, by rolling back the changed values in the registers (including PC and status registers) and memory to a fault free state. We assume that the CR hardware is fault-free. In this paper, the baseline processor assumed is a single-issue in-order integer unit. Checkpointing of out-of-order processors is out of the scope of this paper. In addition, since embedded systems are typically deployed with on-chip memory [28], the baseline processor is assumed to be working with on-chip memory, which can be accessed within one or two clock cycles [29].

4 RELI CHECKPOINT RECOVERY SCHEME

4.1 Definitions

We define the following terms to better explain RELI functionality.

- *Architectural states (AS)* are a set of values of the program-transparent storage such as registers and data memory. Architectural states are main objects which are captured during RELIS state capture, and restored during the rollback operations.
- *Checkpoint data (CD)* is the data required to restore the architectural states. RELI's checkpoint data covers the architectural states. Checkpoint data is generated in state capture operation. Thus CD includes AS as well as the places to where the data should be restored.
- *Checkpoint storage (CS)* is the memory that holds the checkpoint data. Checkpoint storage is written during RELI's state capture operation and if necessary read back in the rollback operation.

4.2 Overview

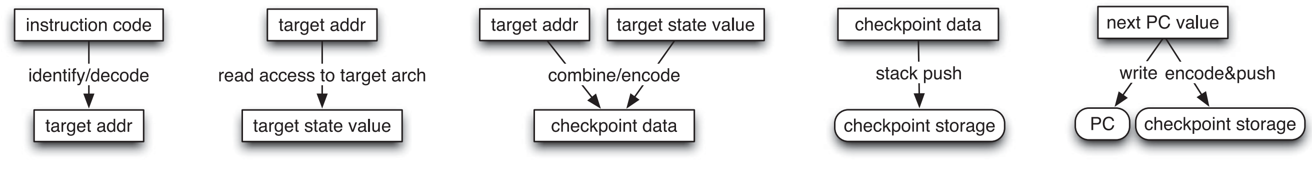
Fig. 1 shows an overview of RELI's fine-grained CR functionality, which is at the basic-block level. In each basic block, RELI's state capture operation allows every instruction to automatically back up architectural states, which are changed by that instruction (for example, When ADD R2, R3, R4, occurs, the original value of R2 will be backed up, before being updated with the new value). State capture is performed simultaneously with the original instruction. A checkpoint validation (e.g., control flow checking) is performed at the end of each basic block. If a fault occurs, the last instruction in the basic block initiates a rollback.

RELI's rollback operation restores the architectural states to the most recent correct value. After the rollback operation, the states of the process/task are identical to those before entering the basic block. Fault detection, which has been extensively covered by other bodies of research (control-flow based or data-flow based) [4], [5], [30], is assumed to be given and not a part of RELI.

4.3 Elements of State Capture Functionality

Figs. 2a, 2b, 2c, and 2d present the basic components of state capture functionality. The rule of thumb of this functionality is *making a backup before committing an update to the architectural states*. The first component, shown in Fig. 2, is *identification of target architectural states* that are to be captured or backed up. A target architectural state is defined as a member of the set of architectural states, which is the destination (being written/updated) of the underlying instruction. To this end, the specific code field in the instruction code needs to be fetched and decoded to obtain the address of target architectural states.

The second component, shown in Fig. 2b, is *fetching the value of target architecture states*. This step requires the address of the target architecture states from the first



(a) State identification (b) Fetching (c) Data generation (d) Data write (e) PC

Fig. 2. Basic components of state capture functionality. The components are ordered in sequence of occurrence during a state-capture event.

TABLE 1
Instruction versus Checkpoint Storage

Cycle	Instruction	Operation	AS	CS
t	ADD $R1, R2, R3$	$R3 \leftarrow R1 + R2$	$R3$	i
$t+1$	SUB $R4, R5, R6$	$R6 \leftarrow R4 - R5$	$R6$	$i+1$
$t+2$	LD $R3, R6, R3$	$R3 \leftarrow M[R3 + R6]$	$R3$	$i+2$
$t+3$	AND $R3, R0, R3$	$R3 \leftarrow R3 \wedge R0$	$R3$	$i+3$

AS: address of target architectural state.
CS: address of CS. higher address are closer to stack top.

TABLE 2
Primitive versus Reduced State Capture for Registers

Cycle	Instruction	Operation	$\sum$	$\sum'$
t	ADD $R1, R2, R3$	$R3 \leftarrow R1 + R2$	1	1
$t+1$	SUB $R4, R5, R6$	$R6 \leftarrow R4 - R5$	2	2
$t+2$	LD $R3, R6, R3$	$R3 \leftarrow M[R3 + R6]$	3	2
$t+3$	AND $R3, R0, R3$	$R3 \leftarrow R3 \wedge R0$	4	2

$\sum$*: cumulative sum of the number of primitive state capture.*
$\sum'$*: cumulative sum of the number of reduced state capture.*
Note*: highlighting denotes redundant state capture.*

component. This component also adds one read access to the corresponding resource, for example, register file read access and memory read access.

The third component, shown in Fig. 2c, is *generating checkpoint data*. In order to guarantee that checkpoint data is sufficient for rollback, both the address and value of target architectural state are required. These two informations are provided by previous components. To combine the informations together, an encoding method is needed. Here, we adopt a simple encoding method, which concatenates the two inputs in such form as $\mathsf{CD} = \mathsf{target\_addr} \,||\, \mathsf{target\_state\_value}$ where the bits of address are more significant bits followed by the bits of architectural state value. Hence, the total width of checkpoint data is equal to the sum of the numbers of address bits and value bits. For one target architecture state i, this relation is given as

$$\mathsf{size\_CD}_i = \mathsf{width\_addr}_i + \mathsf{width\_value}_i, \tag{1}$$

which are architecture dependent.

The fourth component, shown in Fig. 2d, is *pushing checkpoint data into checkpoint storage*. Checkpoint storage is generally a stack, meaning a last-in-first-out (LIFO) structure. As shown in Table 1, LIFO structure is optimal as it guarantees the most recent state is first written back for an architecture state during rollback. To decreasing memory overhead, checkpoint storage is separated from the main memory. In addition, as suggested by Equation (1), $\mathsf{size\_CD}$ can vary amongst different architectural states. Therefore, CS includes separate stacks, each of which is associated to one type of architectural states (e.g., registers and memory). After checkpoint data is generated, it is written into checkpoint storage in the fashion of stack push. Given a basic block j, the size of checkpoint storage is determined by the number of target architectural states ($\mathsf{target\_AS}$), shown in the following equation

$$\mathsf{size\_CS}_j = \sum_{i \in \mathsf{target\_AS}} \mathsf{size\_CD}_{i,j}, \exists\, \mathsf{target\_AS} \subset \mathsf{AS}. \tag{2}$$

Therefore, for an application consisting of k number of basic blocks, the size of checkpoint storage is bounded by the maximum size of checkpoint storage amongst the basic blocks. This relation is given as

$$\mathsf{size\_CS} = \lceil \mathsf{size\_CS}_j \rceil, \forall\, j \in \mathbb{N}^+, j \leq k. \tag{3}$$

4.4 Reduced State Capture for Registers

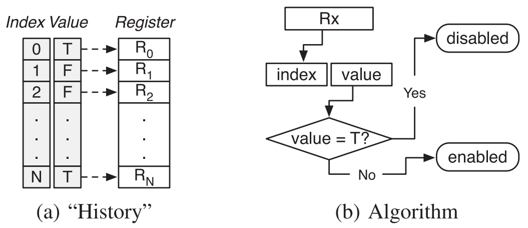
Reduced state capture is motivated by the property that capturing a subset of target architectural states, in a basic block, might satisfy the requirement of rollback or restoring the status of process to the end of the most recent basic block. Therefore, not all the occurrences of register write/update in a basic block is necessary for state capture, and therefore state capture for those states is redundant. Avoiding redundant state capture is a leverage to reduce the cost in terms of CD and CS size. Table 2 provides a comparison to primitive state capture with the same instructions in Table 1. Without the functionality of reduced state capture, redundant state capture occurs in instructions at $t+2$ and $t+3$ (highlighted), which accommodate write events to the same address $\mathsf{R3}$.

Fig. 3 shows the functionality of reduced state capture for registers. The rule of thumb is to ensure redundant state capture is disabled. To this end, reduced state capture for registers keeps a table of "history" for the state capture events in every basic block. Fig. 3a depicts the mapping between the table (highlighted in grey) and registers. An index $i \in [0, N]$ in the table is associated to 1-bit binary value h_i ($h_i \in \{\mathsf{True}, \mathsf{False}\}$), and one register address R_i. History table is a special storage. The size of table, $\mathsf{size\_HT} = 1 \cdot (N+1)$, is determined by the total number of registers ($N+1$), since the association of index to registers is a direct one-to-one mapping.

Fig. 3b depicts the algorithm of using history table to avoid redundant state capture for registers. Upon state capture, $h_i = False$ indicates that the state of R_i has not been captured during the previous executions in the current basic block. In this case, state capture is not redundant and allowed on R_i. On the contrary, $h_i = \mathsf{True}$ means that the state of R_i has been captured, and thus state capture is redundant and avoided. At the end of basic block, the values in history table are reset to False. In fault-free situation, history table reset occurs after checkpoint validation and before entering the next basic block. Otherwise, it occurs after rollback and before restarting the current basic block.

4.5 Special State Capture for Program Counter (PC)

Program counter is a special architectural state. State capture for PC is different from that for other states. PC is deterministically written/updated in every instruction,



(a) "History" (b) Algorithm

Fig. 3. Functionality of reduced state capture for registers.

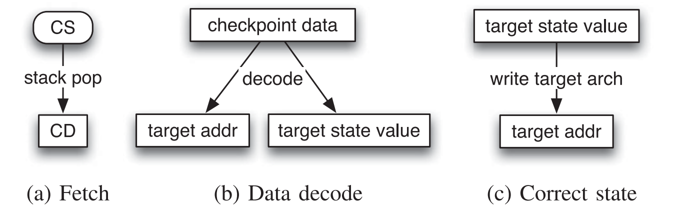


Fig. 4. Basic components of rollback functionality. The components are ordered in sequence of occurrence during a rollback event.

albeit state capture for PC happens only at the last instruction of every basic block. The reason is that rollback aims to restore the status of the process to the entry point of the underlying basic block and therefore the subsequent changes on PC are negligible except the last update on PC in the previous basic block before entering the underlying basic block.

The mechanism of special state capture for PC is shown in Fig. 2e. State capture is activated only on condition that checkpoint validation is successful, meaning there has been no faults during the underlying basic block. State capture for PC generates checkpoint data on next PC value and push checkpoint data into CS, along with the original PC write operation. CS for PC can either be a part of CS for registers (with need for encoding) or a unique separate storage (without need for encoding). If using a separate stack, $\mathsf{size\_CS}_{PC}$ is equal to $\mathsf{width\_value}_{PC}$.

4.6 Rollback Functionality

Fig. 4 depicts the basic components in rollback functionality, which are essentially in opposite order to those components in state capture. The rule of thumb of this functionality is *undoing the update on the architectural states that is committed by an original operation* of a faulty basic block.

The first component, shown in Fig. 4a, is *fetching checkpoint data* from checkpoint storage. As checkpoint storage is LIFO structure (stack), checkpoint data is popped out from the stack top. The second component, shown in Fig. 4b is *decoding checkpoint data* to get address and value of target architectural state. Corresponding to the encoding scheme in state capture, the more significant bits are taken as address, and the less significant bits are taken as value. The third component, shown in Fig. 4c is *correcting the architectural state* by writing the decoded value back to the decoded address. The number of write events is equal to the number of state capture events in the underlying basic block.

As rollback functionality is not performed (all the time) along with the original operations of the application (as opposed to state capture functionality). Therefore, this functionality is integrated into the original application, in form of an additional routine/function. Thus, control transfer is needed at two time points, i.e., upon starting and finishing a series of basic steps in rollback. For a rollback event, the first control transfer moves the original operating state (the program of the underlying application) to rollback operating state, while the second one moves the operating state in the opposite direction.

Rollback routine mainly consists of three while-loops. The first loop handles register-file rollback. The second loop manages data memory rollback. The third loop implements special-register rollback. In each loop, the three basic steps (i.e., fetch, decode, and correct) are executed. Before control transfer back to the original program, resetting checkpoint storage is done.

5 Reli Instruction Set Architecture

This section presents the implementation of Reli functionality on top of original (baseline) instruction set architecture. The implementation includes two major parts corresponding to state capture and rollback respectively. State capture is implemented by *customizing original instructions*, while rollback is implemented by *extending original instruction set (creating custom instructions)*.

Table 3 provides the overview of Reli ISA. For the sake of brevity, this table shows the selected representative instructions in subset α and three Reli custom instructions (i.e., subset β). The following sections will elaborate implementation of Reli instruction set architecture along with this table.

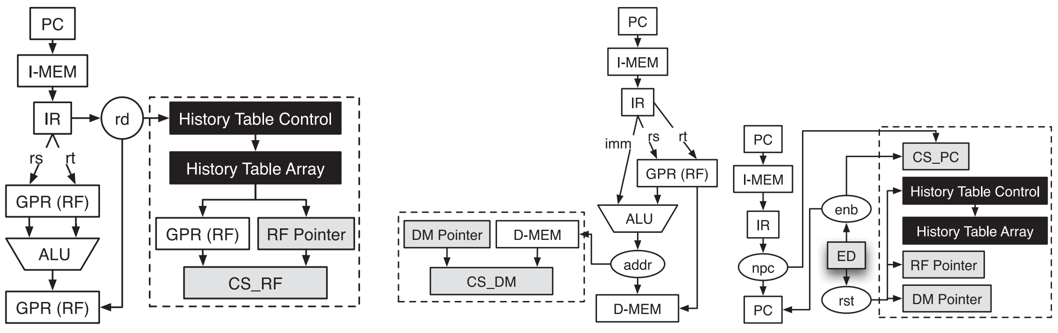
5.1 Base Instruction Set Architecture

The base instruction set architecture is a single-issue in-order RISC architecture, which is typical for embedded processors, with integer unit (i.e., excluding floating-point unit). At the current form, cache structure is not considered in the baseline architecture. The base instruction set is portable instruction set architecture (PISA) from `SimpleScalar` tool suite [31], which is a close derivative of `MIPS-IV` [32].

The base instruction set (excluding floating-point instructions) has 72 instructions, in which seven instructions are selected as representative instructions in Table 3. These instructions are addition (ADD), multiplication (MULT),

TABLE 3
Reli Instruction Set (with Selected Subset α Instructions)

Subset	Instruction	Type	Original Operations	Reli Operations
	ADD	ARI	GPR(rt) $\leftarrow$ GPR(rs) + GPR(rd)	reg_capture(GPR(rt))
	MULT	ARI	HI,LO $\leftarrow$ GPR(rs) $\times$ GPR(rd)	reg_capture(GPR(rt))
α	LW	L/S	GPR(rt) $\leftarrow$ MEM[GPR(rs) + GPR(rd)]	reg_capture(GPR(rt))
	SW	L/S	MEM[GPR(rt)] $\leftarrow$ GPR(rs) + GPR(rd)	mem_capture(MEM[GPR(rt)])
	J	CT	PC $\leftarrow$ target	validate(), pc_capture(target)
	JALR	CT	GPR(rd) $\leftarrow$ PC + 8, PC $\leftarrow$ GPR(rs)	validate(), reg_capture(GPR(rd)), pc_capture(GPR(rs))
	BEQ	CT	if GPR(rs) = GPR(rt) then PC $\leftarrow$ PC + offset	validate(), pc_capture(PC + offset)
	RFRB	n/a	n/a	reg_rollback(i), $\forall\, i \in$ GPR $\cap$ CS
β	SRRB	n/a	n/a	reg_rollback(i), $\forall\, i \in$ SR $\cap$ CS
	DMRB	n/a	n/a	mem_rollback(i), $\forall\, i \in$ MEM $\cap$ CS



(a) ADD (b) SW (c) J

Fig. 5. Datapath of R-type ADD, I-type SW, and J instructions in Reli ISA.

load-word (LW), store-word (SW), jump (J), jump-and-link-register (JALR), and branch-if-equal (BEQ). The instructions can be categorized into three major types: arithmetic (ARI), load/store (L/S), and control transfer (CT). ARI includes two representative instructions: ADD and MULT. L/S includes two: LW and SW. CT includes three: J, JALR, and BEQ.

5.2 Reli Checkpointing Instructions

To demonstrate the design of Reli checkpointing instructions (Subset α), we elaborate the implementation of three representative instructions, addition, store-word, and jump. The rule of thumb in designing Subset α instructions is ensuring the original path is included in the resultant new datapath. This rule allows the Reli instructions to be able to perform original operations defined by the given baseline instruction set architecture.

5.2.1 ADD Instruction

Fig. 5a presents Reli's implementation of ADD instruction (R type). The datapath includes two parts: the original one on the left and the augmented one in the dashed frame on the right. The original part implements the operations for addition that are equivalent to the counterpart in baseline architecture. The augmented part in ADD instruction implements the functionality for register-file state capture, which are described in Sections 4.3 and 4.4. The augmented part essentially is a separate path, of which the data source is rd field[3] from the decoded instruction code. In ADD instruction, rd is the address of the target AS. The black blocks represent the hardware components realizing reduced state capture (depicted in Fig. 3). Both black and gray colors denote the hardware blocks that are implemented for Reli functionality.

In specific, History Table Control is hardwired implementation of the algorithm in Fig. 3b, while History Table Array the index-value pair in Fig. 3a. Using rd as the input, History Table Control reads the corresponding value in History Table and makes decision on avoiding the state capture. The decision signal is the input to RF and RF pointer. If the state capture is allowed, the corresponding RF read access is operated. The value read from RF is encoded with rd to make CD. RF Pointer and CS_RF are combined together to work as a stack. RF Pointer's value is the address of next empty entry/slot in the stack. If the state capture is allowed, RF Pointer's current value is read and write access to CS_RF at the address with CD is enabled.

3. Defined as bits $[15:8]$ in R-type instruction code in `SimpleScalar PISA`.

The RF state-capture operations are all scheduled in pipeline stage 2, i.e., instruction decoding (ID) stage. Based on the functionality discussed in Section 4.4, the operations are scheduled: (1) history table is read at first using rd as input; (2) based on the result from history table read, the RF and its pointer (RFP) are read to obtain both checkpoint data and the next empty CS location; and, (3) Checkpoint data is written into the the next empty CS location, while the RFP is incremented.

5.2.2 SW Instruction

Fig. 5b presents Reli's implementation of SW instruction (I type). This instruction is an example of using immediate field imm to calculate address of target AS and doing state capture for data memory. Similarly, the augmented part is in dashed frame. In comparison to the Reli instructions with RF target AS (e.g., ADD), Reli instructions with DM target AS (e.g., SW) have relatively simple augmented part. As there is no reduced state capture functionality in Reli SW, the augmented part only implements elemental state capture functionality. The data source of the augmented part is an intermediate variable addr, which is the address of the data memory for the original memory write operation. This variable is the output of ALU's addition operation with imm and corresponding register value in RF. DM Pointer works similar to RF Pointer in ADD. Provided addr, data memory read access is operated to fetch the corresponding value and to encode CD. At last, CD is written into CS_DM at the address pointed by DM Pointer.

DM state-capture operations are scheduled at two pipeline stages, MEM1 and MEM2. In SW instruction, there is only one memory write happening at Stage 4. Hence, DM state capture operations are also scheduled at Stage 4. The sequence of DM state-capture operations is as same as that of RF state-capture operations, except there is no operation related to history table any more.

5.2.3 J Instruction

Fig. 5c presents Reli's implementation of J instruction. J and other control transfer instructions act as the exit point of current checkpoint period. Thus, there are two sets of Reli operations implemented: checkpoint validation and state capture for PC. The augmented datapath has two data sources. One is the variable npc denoting the next value for PC.

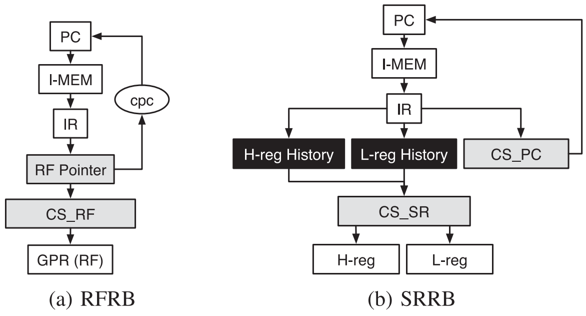


Fig. 6. Datapath of rollback instructions in Reli ISA.

npc is an intermediate variable in original datapath and its value is from target field[4] in J-type instruction code. The other data source is output of error detection logic (ED), which is assumed[5] in this work and illustrated with shadow in Fig. 5c. The output from ED affects: enb (enable) signal to PC and CS_PC, and rst (reset) signal to the hardware components (i.e., pointer and table logics) that implement RF and DM state capture.

In specific, writing npc to PC and CS_PC, as well as resetting state capture components, are allowed, if ED's output indicates error-free. Enabling PC and CS_PC allows the processor to execute the next basic block, while resetting pointers and table logics refreshes the relevant logics for state capture operations dedicated to the next basic block.

5.3 Reli Rollback Instructions

Rollback (Subset β) instructions realizes the rollback function specified Section 4.6. There are three rollback instructions dedicated to three loops for rolling back RF, data memory, and special registers, respectively. These instructions are register file rollback (RFRB), data memory rollback (DMRB), and special register rollback (SRRB). Fig. 6 presents the datapath of RFRB in (Fig. 6a) and SRRB (Fig. 6b) instructions. Since the datapath of DMRB is very similar to RFRB, we will use Fig. 6a to discuss DMRB as well.

5.3.1 RFRB Instruction

RFRB, shown in Fig. 6a, implements the loop for RF rollback. After the instruction code is decoded, the RF pointer is read to obtain the current RF pointer value (`val_rfp`). If val_rfp $\neq 0$, the instruction will pop out one element (checkpoint data) from CS_RF, and decrement RF pointer. In the next step, the address bits and value bits are obtained from the checkpoint data to restore the corresponding register in RF. At last, the instruction lock the PC value to the current PC value (cpc), so as to continue executing RFRB in iteration. If val_rfp $= 0$, RFRB will disable rollback operations (equivalent to a NOP instruction). Without any change to the PC value, RFRB naturally pass the control to DMRB, which is at the next PC address.

5.3.2 DMRB Instruction

DMRB implements the loop for data memory rollback. The datapath of DMRB is almost same as RFRB, except that DMRB uses DM Pointer and CS_DM. After the instruction code is decoded, the DM pointer is read to obtain the current DM pointer value (`val_dmp`). DMRB will be iteratively executed until val_dmp $= 0$. When val_dmp $= 0$, DMRB also performs as a NOP instruction, and passes the control to the last instruction in rollback routine, which is SRRB.

5.3.3 SRRB Instruction

SRRB, shown in Fig. 6b, implements the last parts of rollback functionality. Because PISA only has two special registers (high and low registers for the instructions related to multiplication and division), the loop implementation is simplified by using two history registers corresponding to high and low registers. In addition, CS_SR is implemented as two normal registers instead of a stack. After the instruction is decoded, the instruction checks the history bits. If one history bit is true, the instruction moves the corresponding checkpoint data from CS_SR back to the special register associated to the history bit. As the last instruction in the rollback routine, SRRB also writes back the PC value stored in CS_PC, in order to change the control flow of the processor to the beginning of the current checkpoint interval. At last, SRRB resets all the history bits, including the history bits for RF and high/low registers.

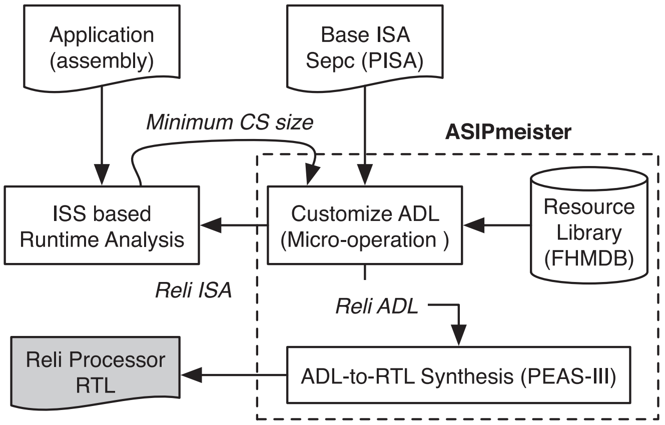


Fig. 7. Reli ASIP design flow.

6 Reli ASIP Design Flow

Fig. 7 depicts the flowchart of our Reli ASIP design flow. Reli ASIP design flow is implemented based on a commercial tool called `ASIPmeister`, which includes an architectural description language (ADL) specification, a resource library, and a ADL-to-RTL synthesis engine. The ADL specification, called micro-operation, describes the data transfer and operations in the instructions. Fig. 8 depicts the example ADL codes of the R-type ADD instruction, from the base ISA and Reli ISA. For brevity, the variable (wire) declaration is omitted. The base ISA on the left only has four operations, for fetching operand registers. In comparison, the Reli ISA on the right has more operations. The first part (grey color background) of micro-operations are about history table access and history checking. In this part, the last micro-operation generates the control signals (i.e., "cond1") for register state capture. The last part (black color background) of micro-operations are mainly register state capture, including reading RF (GPR), concatenating register address and value bits, and writing

4. Bits [25:0] of J-type instruction code in `SimpleScalar PISA`.
5. We assume ED as a block-level control-flow based detection mechanism, such as CFCSS [33] and IMPRES [30].

```
clk(2){
tmp_source0 = GPR.read0(rs);
tmp_source1 = GPR.read1(rt);
source0 = FWU0.forward(rs,tmp_source0);
source1 = FWU1.forward(rt,tmp_source1);
}
```

```
clk(2){
tmp_source0 = GPR.read0(rs);
tmp_source1 = GPR.read1(rt);
source0 = FWU0.forward(rs,tmp_source0);
source1 = FWU1.forward(rt,tmp_source1);
flag_sel = rd;
pre_flag = bufflag.read();
a0 = flag_sel == "00000";
...
a31 = flag_sel == "11111";
var_flag = <a31,a30,...,a0>;
tmp_flag = var_falg | pre_flag;
cond0 = COMP32.cmp(tmp_flag.pref_flag);
cond1 = ~cond0;
reg00 = [cond1] RFC.read();
null = [cond1] RFC.inc();
reg01 = [cond1] GPR.read4(rd);
reg02 = [cond1] FWU4.forward(rd,reg01);
data = <rd,reg02>;
null = [cond1]RFRAMreq.write(one1b);
null = [cond1]RFRAMrw.write(one1b);
null = [cond1]RFRAMaddr.write(one1b);
null = [cond1]RFRAMdout.write(data);
null = [cond1]bufflag.write(tmp_flag);
}
```

Fig. 8. Comparison of microoperations of ID stage in ADD between the base (left) and Reli (right) ISA.

checkpoint data to checkpoint storage. The last micro-operation updates the history table.

The resource library consists of the parameterized VHDL/Verilog descriptions of the functional units, such as ALU, adder, multiplier, etc. The ADL-to-RTL synthesis engine, called `PEAS-III` [34], converts the high-level description of the ISA to the RTL description of the corresponding processor.

In Reli ASIP design flow, there are two inputs. First, the base ISA is `SimpleScalar PISA`, which is discussed in Section 5.1. Second, the target application(s) described in assembly language. Given the two inputs, the major steps of Reli ASIP design flow are as follows.

ADL Customization. Based on the ADL of base ISA, the ADL model of Reli ISA is manually created. The major functional units in Reli instruction's datapath (discussed in Section 5) are allocated and sequenced (scheduled) during this step. In order to resolve the size of checkpoint storage, instruction-set simulator (ISS) based profiling is adopted as well.

ISS Based Profiling. Based on the ADL of Reli ISA, Reli ISS is built on top of SimpleScalar simulator [31]. Reli ISS has runtime analysis functions, which profile the program's runtime events and calculate the application-specific minimum size of checkpoint storage. The minimum checkpoint storage size is the worst-case size amongst all the checkpoint intervals during program's execution. These runtime functions mainly implement Equations (1), (2), and (3). The profiling result is passed back to ADL customization to determine the final Reli ASIP ADL.

In addition, for running RTL simulation with Reli processors, the methodology in [35] is adapted to generate the resultant simulation model of memory. This process also inserts the Reli rollback instructions (i.e., rollback routine) into the instruction code in the instruction memory. The starting PC address of the rollback routine is passed to Reli ASIP design flow as well.

7 Experiment and Results

7.1 Experimental Setup

Experiments were conducted on a computer with the following configuration: `Intel Xeon X7560` CPU (2.27 GHz),

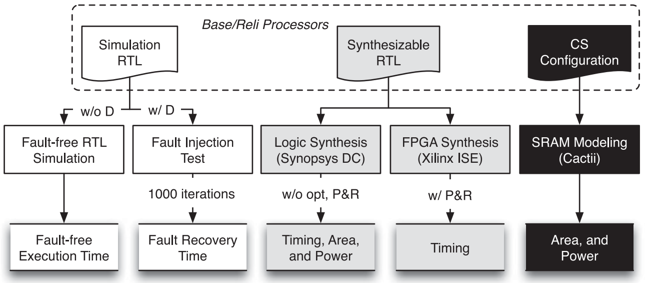


Fig. 9. Experimental methodology.

24 MB cache, and 256 GB main memory. As shown in Fig. 9, our experimental methodology includes RTL simulation, synthesis (shown in grey), and SRAM for checkpoint storage (CS) models (shown in black). Specifically, the experiment consists of five flows. *Fault-free RTL simulation* to provide the cycle-accurate fault-free execution time. We also evaluated the reduction in the number of state capture by the use of the history table in Reli using this simulation model. The results from the RTL simulation model for the fault free scenario are discussed in Section 7.2. *Fault injection test* examines fault recovery time, i.e., rollback time. The recovery times are discussed in Section 7.3. *Logic synthesis* shows the hardware cost, which is bound to a real-world ASIC fabrication technology. Hardware costs are discussed in Section 7.4. *FPGA synthesis* presents the timing impact of Reli after place-and-route (P&R), targeting a contemporary commercial FPGA device and technology. The FPGA timing result is discussed in Section 7.5. *SRAM modeling* is used to study the impact of different implementation for checkpoint storage, including D-flip/flop (DFF) and SRAM. These memory results are discussed in Section 7.7.

The RTL simulation environment used is `Mentor Graphics ModelSim` (a HDL simulator).[6] The logic synthesis tool used is `Synopsys Design Compiler`.[7] The FPGA synthesis tool used is `Xilinx ISE`.[8] The SRAM modeling tool is `Cacti`.[9] The details of the tool configuration are introduced in the corresponding sections.

The benchmark applications used for RTL simulation are from `MiBench` suite [36] and represent typical applications for embedded processors. For simulation flows, we only tested selected MiBench applications. The reason is twofold: (1) the code and data size of `MiBench` is suitably small for RTL simulation; and, (2) our RTL simulation environment (e.g., boot-up and system-call implementations) does not support `SPECINT` applications. For synthesis flows, we targeted both six `MiBench` and six `SPECINT`[10] applications, so as to find the worst-case scenario for hardware overhead. The application binaries are generated using the `SimpleScalar PISA` compiler [37].

In the experiment, we tested three different types of processors: *Base processor* is defined as the processor that shares the base pipeline (no CR hardware) with a Reli processor

6. http://www.model.com/
7. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/
8. http://www.xilinx.com/products/design-tools/ise-design-suite.html
9. http://www.hpl.hp.com/research/cacti/
10. https://www.spec.org/cpu2006/CINT2006/

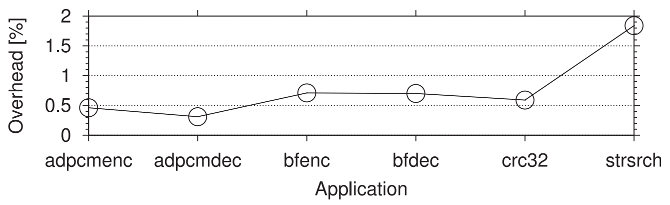


Fig. 10. Fault-free execution time results.

and only executes the native instruction set. RELI *general processor (GP)* is the processor with Reli functionalities, which is directly augmented on top of a base processor with *full integer instruction set*. Reli GPs are synthesized to observe the worst-case overhead in terms of area, power, and timing. RELI *application-specific instruction-set processor* is the processor with Reli functionalities, which is augmented on top of a base processor with *a special instruction set*. The special instruction set is tailored for the target application. In this paper, we only applied instruction-set pruning for tailoring one instruction set. An instruction is pruned, if the instruction is never used in the target application's execution. The corresponding hardware of a pruned instruction is not implemented. These ASIPs are tested in both simulation and synthesis. Before evaluation, the baseline processors are generated, and their functionality is verified.

7.2 Fault-Free Simulation Results

Fig. 10 presents the fault-free execution time results. Fault-free execution time represents the processor performance while there is no fault occurrence in the system. Base processor (without any recovery mechanism installed) and the RELI ASIPs are compared for fault-free execution time across the six applications. The y-axis is the fault-free execution time overhead in percentage The main reason of the fault-free execution time overhead is the pipeline stalls, caused by pipeline resource hazards. The worst-case execution time overhead is 1.84 percent in `strsrch`, which has the most number of pipeline stalls when RELI processor executes the program, while the least overhead is 0.31 percent in `adpcmdec`. The average execution time overhead across the six applications is 0.76 percent.

Fig. 11 demonstrates the effect of using history table for reducing the number of register state capture. The y-axis is the percentage of reduction for register state capture, in comparison to the naive scheme, which performs same CR without history table. Without history table, there are much more register state captures during program runtime. As a result, we observed significant reduction of register state captures among the six applications. The maximum reduction is 57 percent in `strsrch`, while the minimum reduction is 44.3 percent in `bfdec`. The reason for the greater reduction in `strsrch` is that `strsrch` has the greatest number of register writes with the same destination registers (equivalent to the number of redundant register state captures defined in Section 4.4). The average reduction is as high as 50.8 percent. This result indicates the efficacy of history table in RELI CR scheme.

7.3 Fault Injection Test

7.3.1 Test Methodology

The fault-injection test environment is implemented using Python and bash scripts. Bit-flips are injected into the

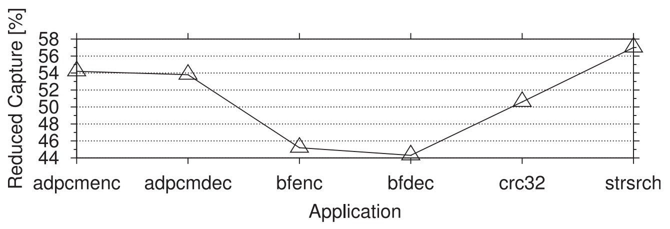


Fig. 11. Reduced register state capture by using history table.

system at the instruction level, which is more abstract than gate level. Instruction-level fault injection, also known as software-implemented fault injection (SWIFI) [38], [39], is selected because:

i. The focus of this paper is about checkpointing and recovery, and not detection. As such, we only implemented an instruction level fault injection test procedure. Performing fault injection test at gate level for a very large system, such as a processor, requires significant implementation effort and time, including implementation and simulation time.
ii. All checkpoint/recovery techniques only react to fault occurrence after the fault is detected. We have existing detection method implemented for this processor, which can detect the bit-flips injected at the instruction level. Gate-level fault injection will require a compatible detection mechanism to be implemented as well (which is beyond the scope of this paper).
iii. Instruction-level injection can result in sufficient manifestation in processor architecture for the purposes of testing checkpoint and recovery. Such manifestation includes incorrect writes to registers and memory, as well as incorrect operations.

In order to inject three types of bit-flips in data, address, and control bits, at the instruction level, we directly inject bit-flips into the instruction code. To inject a bit flip, a random location of the instruction memory is chosen. Then a random bit of the instruction code at the location is flipped to the opposite binary value. For one instruction, depending on the exact bit, which is flipped, the manifestation of the bit flip varies. The bit flip in *opcode field* results incorrect operation, which represents the *control fault*. The bit flip in other fields, e.g., *operand name* and *immediate data fields*, represents *datapath fault*.

The number of fault injections is 1,000, in order to obtain a small error margin (4 percent with 99 percent confidence level) according to [40], for each application. We inject one fault for each iteration. The test consists of three procedures: (1) injecting a fault at compile time to the instruction memory data file; (2) invoking the HDL simulator to run the application; and, (3) collecting the run-time behavior from the simulation transcript.

To make the simulation as close as possible to a realistic one, we implement a detection technique similar to IMPRES [30] to work with RELI. The detection mechanism monitors bit-flips of instructions via a control-flow based mechanism, and communicates to RELI at the end of every basic block. Since the library code is difficult to modify, the faults in library code are excluded in this test.

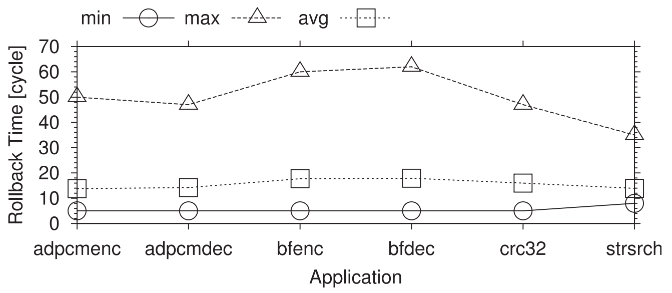


Fig. 12. Rollback time results.

7.3.2 Test Results

Fig. 12 depicts the recovery time of RELI in presence of faults. The result is obtained using Monte-Carlo simulation with RTL models of RELI processors. For each of the six applications, the minimum min, the maximum max, and the average avg of recovery time are shown. Among the six applications, `bfdec` has the largest average recovery time, i.e., 17.9 machine cycles, whereas `adpcmdec` has the smallest average recovery time (13.8 machine cycles). The worst (maximum) recovery time (62 machine cycles) is observed in `bfdec`, since `bfdec` has larger basic blocks than the other applications, and `bfdec` has fewer redundant state captures in the basic blocks. The best (minimum) case (5 machine cycles) is found in `adpcmenc`, `adpcmdec`, `bfdec` and `crc32`.

7.4 Logic Synthesis Results

We obtained synthesis results with TSMC 65 nm library using the Synopsys Design Compiler. No specific timing and power optimizations were applied in logic synthesis. There are three metrics tested during logic synthesis from the resultant gate-level netlist. First, we tested the timing of the critical path of the processors, which determines the clock period and equivalently, the operating frequency. Second, we tested the area of the processor, which is equivalent to the number of gates. At last, we measured the power of the processors. Given that the focus of this work is front-end design, the place-and-route (P&R) is not performed. Without P&R, the dynamic power measurement can hardly be accurate. Hence, we only provide the leakage (static) power results here.

Table 4 presents critical path timing results. Column 2 shows the clock period, i.e., critical path timing, in nanoseconds. Column 3 shows the overhead, compared to the base processor. In addition to the base and RELI processors, we also tested the naive processor, which performs RELI without history table. Because the multiplier and divider from ASIPmeister's library are single-cycle and dominate the critical path timing, we removed these two components during logic synthesis (targeting 1 ns clock period) to view RELI's influence on clock period. Without multiplier and divider, the timing difference between Reli GP and Reli ASIP is negligible. Hence, we did not report the timing for each Reli-ASIP here. As a result, we observed that RELI increases the critical path timing from 1.07 to 1.26 ns. Without the history table, the critical path timing is less (1.16 ns). The timing overhead of RELI is 17.7 percent. The timing increase is due to: (1) history table operations, i.e., the difference between *Base* and *Naive*, and, (2) state-capture operations, i.e., the difference between *Naive* and RELI.

TABLE 4
Critical Path Timing Results with TSMC 65 nm

Processor	Clock Period [ns]	Overhead [%]
Base	1.07	N/A
Naive	1.16	8.4
RELI	1.26	17.7

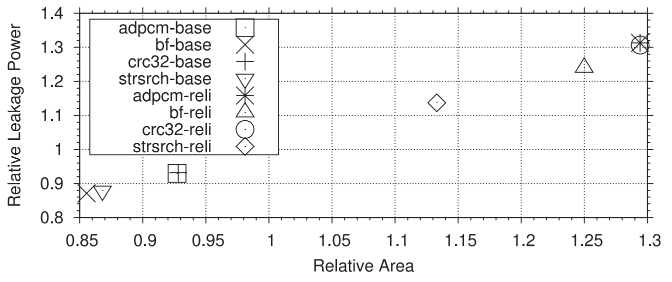


Fig. 13. Relative area and leakage power of RELI ASIPs for `Mibench` normalized to base processor.

Fig. 13 shows the relative area cost and leakage power consumption of RELI and base ASIPs, normalized to the base GP (see Section 7.1 for explanation of the processor types). For this result, we targeted 10 ns (i.e., 100 MHz) clock period, which is viable for all RELI processors with multiplier and divider. Both `adpcmenc` and `adpcmdec` ASIPs are shown as `adpcm-reli`, given that the hardware of RELI ASIP for `adpcmenc` and `adpcmdec` is same. So are `bf` (`bfenc` and `bfdec`) ASIPs. The postifx "-base" indicates the base ASIP (i.e., without Reli functionalities) for one application.

In general, due to instruction-set pruning, all base ASIPs consumes less area and leakage power than the base GP. All Reli ASIPs are larger and more power-hungry than the base GP and base ASIPs. In comparison to the *corresponding base ASIP*, the `strsrch` RELI ASIP has the lowest overhead in both area (30.5 percent) and leakage power (29.3 percent), while the `bf` RELI ASIP has the largest overhead in both area (46.1 percent) and leakage power (42.3 percent). On average, RELI ASIP costs 38.9 percent more area, and 38.3 percent more leakage power, than the corresponding base ASIP.

In order to observe RELI's worst-case hardware overhead for general use cases, we also studied the RELI GPs targeting larger `SPECINT2006` applications. Note that RELI GPs have full integer instruction set implemented, without instruction set pruning. Hence, the difference in the results for different applications comes from the difference in CS size.

Fig. 14 presents the relative area cost and leakage power consumption of RELI GPs, normalized to the base GP. Amongst the six applications, `mcf` shows the lowest overhead in area (53.8 percent) and leakage power

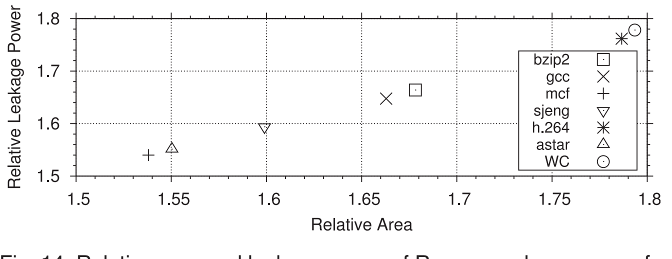


Fig. 14. Relative area and leakage power of RELI general processors for `SPECINT` normalized to base processor.

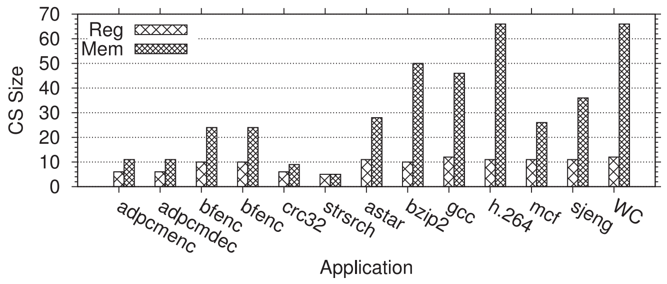


Fig. 15. CS size results.

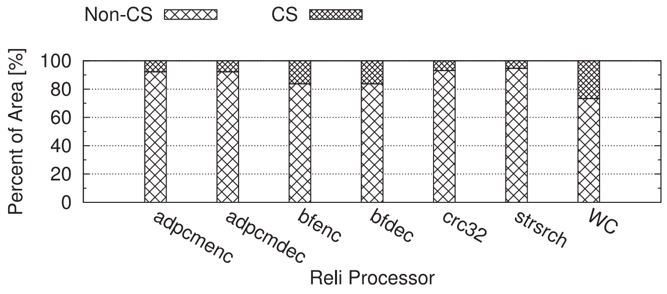


Fig. 16. Area distribution in Reli processors.

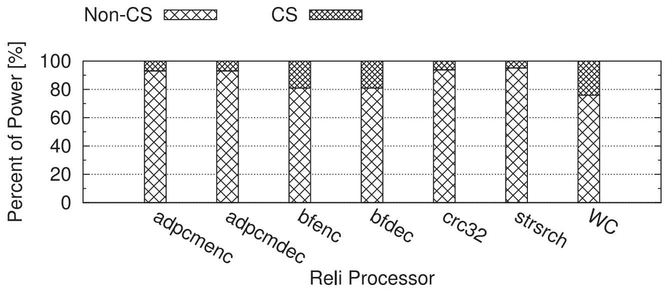


Fig. 17. Power distribution in Reli processors.

TABLE 5
Critical Path Timing Results with
Xilinx Virtex-7 After P&R

Processor	Clock Period [ns]	Overhead [%]
Base	5.986	N/A
Naive	6.096	1.84
Reli	6.277	4.86

(53.9 percent), while h.264 consumes the highest overhead in area (78.7 percent) and leakage power (76.2 percent). We also have the results for MiBench applications. However, as the hardware cost of Reli GPs for SPECINT2006 applications are much higher than MiBench, for the sake of brevity, we only report the results of Reli GPs for six SPECINT2006 applications here.

Further, we also tested a synthetic worst-case scenario, which is a Reli GP capable of running both h.264 and gcc applications from SPECINT2006 suite. This Reli GP is labeled as WC in Fig. 14. Amongst all the applications in the experiment, gcc has the largest number of register checkpointing (i.e., register CS size), while h.264 has the largest number of memory checkpointing (i.e., memory CS size). As a result, WC Reli GP has the highest overhead, i.e., 79.3 percent for area overhead and 77.8 percent for leakage power overhead. Note, the hardware overhead (area and power) is calculated by comparing the Reli processor to the baseline processor. In this comparison, for both types of processors, only the integer unit is included, while the memory, which usually is more than three times larger than an integer unit, is excluded. If we include the memory into the calculation, which is common in other studies, the hardware overhead will become very small (below 20 percent when a memory of $3\times$ area of integer unit is considered). In this experiment, the hardware overhead for error detection is 1 percent of base processor. Around 10 percent more execution time (due to 10 percent increase in code size) is also needed.

In order to see the hardware cost in depth, we divided the hardware cost into two categories: (1) the CS cost, and, (2) the non-CS cost, which includes the base processor and the logic circuits implementing the control and data transfer for checkpointing and rollback. Fig. 15 shows the CS size (in entires/slots) required by each application for performing Reli. This result is obtained from the profiling stage during the ASIP design flow, discussed in Section 6. Fig. 16 presents the area distribution of CS and non-CS hardware in Reli ASIPs, each targeting one of the six applications selected from MiBench suite, as well as Reli GP targeting WC. In each application, the CS size is determined by the corresponding application's runtime behavior (the number of writes in basic blocks).

We observed the correlation between the area distribution in Fig. 16 and CS size required for each application in Fig. 15. The application, which requires larger CS size, tends to have larger proportion of CS area. As a result, the WC Reli processor has the largest proportion (28 percent) of CS area, while the strsrch Reli processor has the smallest proportion (4 percent) of CS area. For leakage power shown in Fig. 17, similar results are found. The WC Reli GP leads the proportion (24 percent) of CS leakage power, while the strsrch Reli ASIP has the smallest proportion (4 percent) of CS leakage power.

7.5 FPGA Synthesis Results

In order to observe the timing impact of Reli after P&R, we synthesized Reli GP, *Naive* checkpoint/recovery processor, and *Base* processor for FPGA. The target FPGA device is Xilinx Virtex-7.[11] Table 5 presents both clock period (Column 2) and overhead (Column 3) of Reli, *Naive*, and *Base* processors. Due to the increased complexity of hardware, Reli raises the critical path timing from 5.9 ns (*Base*) to 6.3 ns when implemented on an FPGA. Hence, the overhead of Reli after P&R is about 4.86 percent. In comparison to the timing results in Table 4, the overhead is considerably lower. There are two possible reasons: (1) the FPGA LUT, limits the highest possible operating frequency of the base processor; and, (2) the wires, which are generated from P&R, dominate timing, and overwhelm the logic elements.

7.6 Wall-Clock Time Overhead

To study the overall performance overhead of Reli, we calculated the wall-clock time of Reli based on the simulation results (clock cycles for execution) and synthesis results (critical path timing). Fig. 18 depicts the wall-clock time

11. http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html

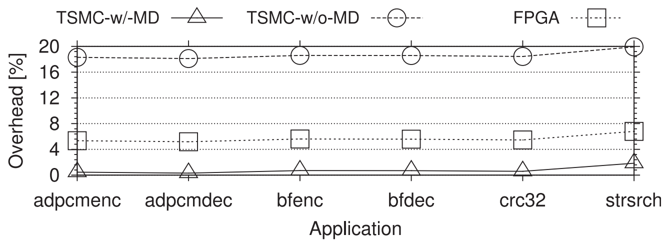


Fig. 18. Wall-clock time overhead.

overhead of six MiBench applications, based on three different timing results. TSMC-w/-MD denotes the overhead calculated using the critical path timing (i.e., 10 ns) with multiplier and divider in the processor, while TSMC-w/o-MD denotes the overhead calculated using the timing (shown in Table 4) without multiplier and divider in the processor. These two timing results are both generated from logic synthesis using TSMC65nm technology. FPGA denotes the overhead calculated using the timing (shown in Table 5) on FPGA, instead of ASIC. FPGA results are affected by P&R, while the TSMC ones are not.

Due to the larger overhead in critical path timing, Reli has the highest wall-clock time overhead (on average 18.7 percent) for the TSMC-w/o-MD processor executing the six applications. With FPGA, Reli incurs much less wall-clock time overhead (5.7 percent on average). As TSMC-w/-MD timing is constant (due to one-cycle multiplier and divider), the wall-clock time overhead is as same as clock-cycle overhead.

7.7 Checkpoint Storage Implementation Comparison

In this section, we used Cacti cache modeling tool to study the overhead of CS cost. The motivation for obtaining this result is as follows: (1) CS is a significant factor in hardware cost of Reli. In fact, the CS of WC-Reli occupies an area equivalent to 47.9 percent of the base processor. (2) In previous results, Reli's CS is implemented as an array of D-flip/flops (DFFs). Reli's CS could also be implemented as SRAM as well, similar to a L1 cache, which is more efficient for larger data array.

Fig. 19 shows the scatter plot including four different CS implementations configured for WC. The x-axis is the relative area while the y-axis is the relative leakage power. Both area and power numbers are normalized to the base processor. The result for TSMC-DFF is from logic synthesis. The results for other three implementations are from Cacti, for different SRAM cells. All the implementations are given for 65 nm technologies. TSMC-DFF has the highest area cost (around 48 percent of base processor). ITRS-HP-SRAM, is implemented with high-performance SRAM cells, with the highest cost in leakage power (2.5× base processor) amongst all implementations, and the highest overall cost amongst all the SRAM implementations. ITRS-LSTP-SRAM, implemented with low stand-by power SRAM cells, has negligible leakage power cost (0.04 percent of base processor). ITRS-LOP-SRAM, implemented with low operating power SRAM cells, shows the least cost in area (34.2 percent of base processor). In general, all the SRAM CS implementations are much smaller than DFF CS for WC.

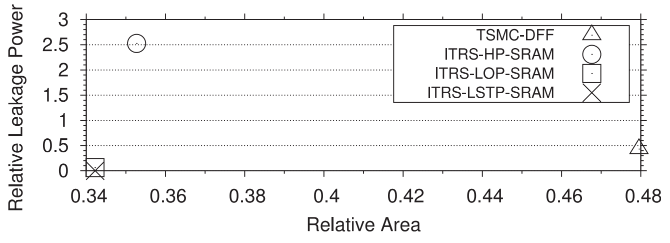


Fig. 19. Relative area and leakage power of CS implementations for WC.

TABLE 6
Checkpoint Recovery Comparison

Technique	Checkpointing	Rollback	CD
Reli	0.31 to 1.8 %	≤ 62 cycles	≤ 624 B
Ocean [24]	4 to 10 % (100 cycles)	≤ 132 cycles	≤ 512 B
P1200 [25]	10 to 29 %	million cycles	≤ 400 kB
Swich [9]	1 % (256 cycles)	≤ milliseconds	0.6 to 30 kB
ReVive [20]	1 to 22 %	0.1 to 1 s	≤ 2.5 MB

7.8 Comparison

Table 6 quantifies features of CR mechanisms of Reli and four state-of-art techniques. These are Ocean [24], P1200 [25], Swich [9], and ReVive [20]. Due to Reli's finer granularity (at instruction and basic block level), Reli shows minimum checkpointing (state capture) time overhead, rollback time overhead, and requires limited checkpoint data size, in comparison to other techniques. Reli increases the program execution time by just 1.8 percent in the worst case (0.76 percent on average). Further, Reli needs at most 62 clock cycles, and is faster than others for one individual rollback. Regarding checkpoint data (CD) size, Reli requires about 624 bytes (as the worst case for SPECINT2006) in the worst case, which is less than P1200 and Swich, and much less than ReVive. Ocean's data size is configured by the optimization algorithm based on the given cost constraints, and occupies less than 512 B in the optimal design for application FFT.

8 Further Discussion and Perspectives

Generality. In order to successfully execute applications, the Reli processor must have sufficient checkpoint storage for every checkpoint period. If there is insufficient amount of checkpoint storage, then a check is forced, by inserting a jump instruction which jumps to the next instruction.

Reliability. Considering the fault occurrence in Reli's checkpoint storage, standard coding-based error correcting techniques (e.g., ECC) can be adopted to improve the reliability of the checkpoint storage. Another possible reliability enhancement can be focused on the rollback stage, where the techniques such as two-time-recovery used in the recovery mode in IBM S/390 G5 [21] can be adopted to guarantee that the recovery is executed correctly. Permanent faults are not within the scope of the paper. If such a fault is detected, Reli processor will probably keep rolling back repeatedly in one basic block. In this case, a watchdog timer is typically used to stop the processor.

Scalability. Reli currently is studied, implemented, and tested targeting uni-processor embedded systems. However, this technique can be scaled to multi-processor embedded systems (such as an MPSoC) by taking a communication mechanism into consideration. Further exploration of MPSoC systems would be interesting.

9 Conclusion

In this paper, we have presented a novel approach for recovering embedded applications from transient faults by customizing instructions. Reli realizes CR by integrating the functionalities into native instructions of the base processor. The augmented processor, i.e., Reli processor allows CR to be executed at a finer granularity than perviously possible, such that the checkpoint data size is reduced greatly. To implement Reli processor, we have built an ASIP design flow, based on a commercial ASIP design tool, which handles ADL-to-RTL synthesis. We have simulated Reli using assembly code from `MiBench` benchmark suite, compiled using `SimpleScalar` tool set. The experimental results show that the fault-free execution time overhead is only 0.76 percent on average. From the fault injection test, we also found that in the worst case, the recovery time is only 62 cycles. Reli costs 44.4 percent area and 45.6 percent leakage power overhead on average, and 79.3 and 77.8 percent in the worst case found in `SPEC-INT2006` and `MiBench` suites.

Acknowledgments

This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500-spp1500.itec.kit.edu).

References

[1] J. M. Rabaey, "Design at the end of the silicon roadmap," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2005, pp. 1–2.
[2] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Comput.*, vol. 39, no. 1, pp. 118–120, Jan. 2006.
[3] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke, "Cost-efficient soft error protection for embedded microprocessors," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2006, pp. 421–431.
[4] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.
[5] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *ACM SIGARCH Comput. Archit. News*, vol. 28, pp. 25–36, May 2000.
[6] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed. Natick, MA, USA: A. K. Peters, 1998.
[7] J. Henkel, et al., "Reliable on-chip systems in the nano-era: Lessons learnt and future trends," in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 99:1–99:10.
[8] C.-C. Li and W. Fuchs, "Catch-compiler-assisted techniques for checkpointing," in *Proc. 20th Int. Symp. Fault-Tolerant Comput*, Jun. 1990, pp. 74–81.
[9] R. Teodorescu, J. Nakano, and J. Torrellas, "SWICH: A prototype for efficient cache-level checkpointing and rollback," *IEEE Micro*, vol. 26, no. 6, pp. 28–40, Nov./Dec. 2006.
[10] N. Cheung, S. Parameswaran, and J. Henkel, "Battery-aware instruction generation for embedded processors," in *Proc. Asia South Pacific Des. Autom. Conf.*, 2005, pp. 553–556.
[11] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. San Francisco, CA, USA: Morgan Kaufmann, 2007.
[12] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu, "An architectural framework for providing reliability and security support," in *Proc. Int. Conf. Depend. Syst. Netw.*, 2004, pp. 585–594.
[13] L. Wang, Z. Kalbarczyk, W. Gu, and R. K. Iyer, "An OS-level framework for providing application-aware reliability," in *Proc. 12th Pacific Rim Int. Symp. Depend. Comput.*, 2006, pp. 55–62.
[14] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," in *Proc. 11th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2004, pp. 235–247.
[15] W. R. Dieter and J. E. Lumpp Jr, "A user-level checkpointing library for POSIX threads programs," in *Proc. 29th Annu. Int. Symp. Fault-Tolerant Comput.*, 1999, Art. no. 224.
[16] J. Duell, P. Hargrove, and E. Roman, "The design and implementation of Berkeley Lab's linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, Dec. 2002.
[17] X. Ouyang, K. Gopalakrishnan, T. Gangadharappa, and D. Panda, "Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture," in *Proc. Int. Conf. High Performance Comput.*, Dec. 2009, pp. 99–108.
[18] R. Ahmed, R. Frazier, and P. Marinos, "Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems," in *Proc. 20th Int. Symp. Fault-Tolerant Comput.*, Jun. 1990, pp. 82–88.
[19] D. Hunt and P. Marinos, "A general purpose cache-aided rollback error recovery (CARER) technique," in *Proc. 17th Int. Symp. Fault-Tolerant Comput. Syst.*, 1987, pp. 170–175.
[20] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 111–122.
[21] T. Slegel, et al., "IBM's S/390 G5 microprocessor design," *IEEE Micro*, vol. 19, no. 2, pp. 12–23, Mar./Apr. 1999.
[22] 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*, vol. 30, pp. 123–134, May 2002.
[23] M. Salehi, M. K. Tavana, S. Rehman, M. Shafique, A. Ejlali, and J. Henkel, "Two-state checkpointing for energy-efficient fault tolerance in hard real-time systems," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 24, no. 7, pp. 2426–2437, Jul. 2016.
[24] M. M. Sabry, D. Atienza, and F. Catthoor, "OCEAN: An optimized HW/SW reliability mitigation approach for scratchpad memories in real-time SoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4s, pp. 138:1–138:26, Apr. 2014.
[25] E. Wchter, N. Ventroux, and F. G. Moraes, "A context saving fault tolerant approach for a shared memory many-core architecture," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2015, pp. 1570–1573.
[26] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36–47, Jan./Feb. 2007.
[27] T. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Microelectronics Division*, Nov. 1997.
[28] J. Henkel and S. Parameswaran, *Designing Embedded Processors: A Low Power Perspective*, 1st ed. Berlin, Germany: Springer, 2007.
[29] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip versus off-chip memory: The data partitioning problem in embedded processor-based systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 682–704, Jul. 2000.
[30] R. G. Ragel and S. Parameswaran, "IMPRES: Integrated monitoring for processor reliability and security," in *Proc. Des. Autom. Conf.*, 2006, pp. 502–505.
[31] T. M. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Comput.*, vol. 35, no. 2, pp. 59–67, Feb. 2002.
[32] C. Price, "MIPS IV Instruction Set, revision 3.2," MIPS Technologies, Inc., Mountain View, CA, Sep. 1995.
[33] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[34] M. Itoh, et al., "PEAS-III: An ASIP design environment," in *Proc. Int. Conf. Comput. Des.*, 2000, pp. 430–436.
[35] J. Peddersen, S. L. Shee, A. Janapsatya, and S. Parameswaran, "Rapid embedded hardware/software system generation," in *Proc. Int. Conf. VLSI Des.*, 2005, pp. 111–116.
[36] M. R. Guthaus, J. Ringenberg, D. Ernst, T. Mudge, R. Brown, and T. Austin, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Symp. Workload Characterization*, 2001, pp. 3–14.
[37] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[38] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Trans. Comput.*, vol. 52, no. 9, pp. 1115–1133, Sep. 2003.
[39] D. T. Stott, G. Ries, M.-C. Hsueh, and R. K. Iyer, "Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection," *IEEE Trans. Comput.*, vol. 47, no. 1, pp. 108–119, Jan. 1998.
[40] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proc. Conf. Des. Autom. Test Europe*, 2009, pp. 502–506.

Tuo Li (M'15) received the BE degree in electronic science and technology from Hefei University of Technology, Hefei, China, in 2008, and the PhD degree in computer science and engineering from the University of New South Wales, Sydney, Australia, in 2014. He is currently a postdoctoral researcher in the School of Computer Science and Engineering, University of New South Wales. His current research interests include cyber security and reliability for embedded systems. He is a member of the IEEE.

Muhammad Shafique (M'11-SM'16) received the PhD degree in computer science from Karlsruhe Institute of Technology, in January 2011. He is a full professor with Vienna University of Technology (TU Wien), Austria, where he is directing the chair of Computer Architecture and Robust, Energy-Efficient Technologies (CARE-Tech). He was a senior research group leader with Karlsruhe Institute of Technology, Germany, for more than 5 years. Before, he was with Streaming Networks Pvt. Ltd. as senior embedded systems engineer for three years. His research interests include power/energy-efficient, reliable, and adaptive computing systems covering various design abstractions of the hardware and software stacks (like micro-architecture, architecture, run-time system, and compiler), embedded systems, emerging technologies and computing paradigms, and their applications in IoT, CPS, and ICTD. He received 2015 ACM/SIGDA Outstanding New Faculty Award, six gold medals, and several best paper awards and nominations at prestigious conferences like CODES+ISSS, DATE, DAC, and ICCAD, Best Master Thesis Award, and Best Elective Lecturer Award. He is the TPC co-chair of ESTIMedia 2015 and 2016, and has served on the TPC of several IEEE/ACM conferences like ICCAD, DATE, CASES, and ASPDAC. He holds one US patent. He is a senior member of the IEEE.

Jude Angelo Ambrose received the MSc degree from the University of Northumbria, Newcastle-upon-Tyne, United Kingdom, and the PhD degree in computer engineering from the University of New South Wales. He is a senior research engineer with Canon Information Systems Research Australia, Australia, and a visiting fellow with the University of New South Wales, Australia. His research interests include automation for embedded systems design, high speed and low power multicore architectures, and secure and reliable embedded systems. He is a member of the IEEE.

Jörg Henkel (M'95-SM'01-F'15) received the PhD degree from Braunschweig University with "Summa cum Laude". He is currently with Karlsruhe Institute of Technology, Germany, where he is directing the chair of Embedded Systems CES. Before, he was a senior research staff member with NEC Laboratories, Princeton, New Jersey. He has/is organizing various embedded systems and low power ACM/IEEE conferences/symposia as general chair and program chair and was a guest editor on these topics in various Journals like the *IEEE Computer Magazine*. He was a program chair of CODES01, RSP02, ISLPED06, SIPS08, CASES09, Estimedia11, VLSI Design12, ICCAD12, PATMOS13, NOCS14 and served as a general chair of CODES02, ISLPED09, Estimedia12, ICCAD13, and ESWeek16. He is/has been a steering committee member of major conferences in the embedded systems field like at ICCAD, ESWeek, ISLPED, Codes+ISSS, CASES and is/has been an editorial board member of various journals like the *IEEE Transactions on Very Large Scale Integration*, the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, the *IEEE Transactions on Multi-Scale Computing Systems*, the *ACM Transactions on Cyber Physical Systems*, the *Journal of Low Power Electronics and Applications,* etc. In recent years, he has given around 10 keynotes at various international conferences primarily with focus on embedded systems dependability. He has given full/half-day tutorials at leading conferences like DAC, ICCAD, DATE, etc. He received the 2008 DATE Best Paper Award, the 2009 IEEE/ACM William J. McCalla ICCAD Best Paper Award, the Codes+ISSS 2015, 2014, and 2011 Best Paper Awards, and the MaXentric Technologies AHS 2011 Best Paper Award as well as the DATE 2013 Best IP Award and the DAC 2014 Designer Track Best Poster Award. He is the chairman of the IEEE Computer Society, Germany Section, and was the editor-in-chief of the *ACM Transactions on Embedded Computing Systems* for two consecutive terms. He is an initiator and the coordinator of the German Research Foundations (DFG) program on "Dependable Embedded Systems" (SPP 1500). He is the site coordinator (Karlsruhe site) of the Three-University Collaborative Research Center on Invasive Computing (DFG TR89). He is the editor-in-chief of the *IEEE Design & Test Magazine* since January 2016. He holds 10 US patent. He is a fellow of the IEEE.

Sri Parameswaran (SM'04) received the BE degree in electrical and computer systems engineering from Monash University, Melbourne, VIC, Australia, in 1986, and the PhD degree from the University of Queensland, Brisbane, QLD, Australia, in 1991. He is a professor in the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW, Australia, where he also serves as the program director of computer engineering. His current research interests include system level synthesis, low-power systems, high-level systems, and network on chips. He has served on the program committees of several international conferences, such as the Design Automation Conference, the Design Automation & Test in Europe Conference, the International Conference on Computer-Aided Design, the International Conference on Hardware/Software Codesign and System Synthesis (as the technical program committee chair), and the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. He is an associate editor of the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* and the *EURASIP Journal on Embedded Systems*. He is a senior member of the IEEE.