An Integrated Design Environment of Fault Tolerant Processors with Flexible HW/SW Solutions for Versatile Performance/Cost/Coverage Tradeoffs

Yi-Ju Ke†, Yi-Chieh Chen‡, Ing-Jer Huang†
†Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung, Taiwan
ylke@esl.cse.nsysu.edu.tw, ijhuang@cse.nsysu.edu.tw
‡Research and Development, Cadance Company, Hsinchu, Taiwan
erikac@cadence.com

***Abstract*—This paper presents an integrated design environment (IDE) for embedded fault-tolerant processor system. It takes in a processor core IP and the embedded software which is to be executed on the given processor, and turns them into a fault-tolerant system with various hardware and software mechanisms, subject to the designer's selection. The hardware options include dual redundancy for processor core, and single-error-detection/correction protections for memory. The software option is control flow error detection. A GUI is provided for the designer to select the options and the IDE automatically generates the hardware Verilog code and the modified embedded software. The IDE reports the cost, speed and power consumption of the generated hardware and the static and dynamic instruction counts of the generated software. In addition, a fault-injection tool is also provided to evaluate the fault coverage of the generated hardware and software. With this IDE, the designer could explore the tradeoffs of cost/performance/ power/fault-coverage. We have successfully demonstrated this IDE with an industrial processor core Andes N8.**

Keywords—processor; fault-tolerant; fault injection; data error; control flow error; tradeoffs

I. INTRODUCTION

As the feature sizes of IC manufacturing processes shrink continuously and the device complexity keeps increasing, IC's become more and more susceptible to cosmic rays or electromagnetic waves in hostile environments, which could cause faulty behaviors in IC's. Therefore, fault tolerance becomes an important issue in IC design and application.

There have been many fault tolerance techniques available, ranging from device, circuit, architecture to software levels. These techniques are practiced at different phases of the design process. However, there are two major problems in adopting these techniques when designing an SoC (system-on-chip). First, it requires a wide range of design, integration and verification skills to adopt these techniques. In addition, it is often necessary to modify the original designs, which is inconvenient or even impractical to the IP (intellectual property)-based design style where IP's (pre-designed hardware modules) are treated as black boxes, not allowing the modification of their internal structures. Second, there is no apparent fault tolerant solution for each SoC design since these techniques have diverse impacts on performance, cost, power, compatibility and fault coverage. For each design case, the designer has to carefully select the appropriate techniques to match the corresponding requirements.

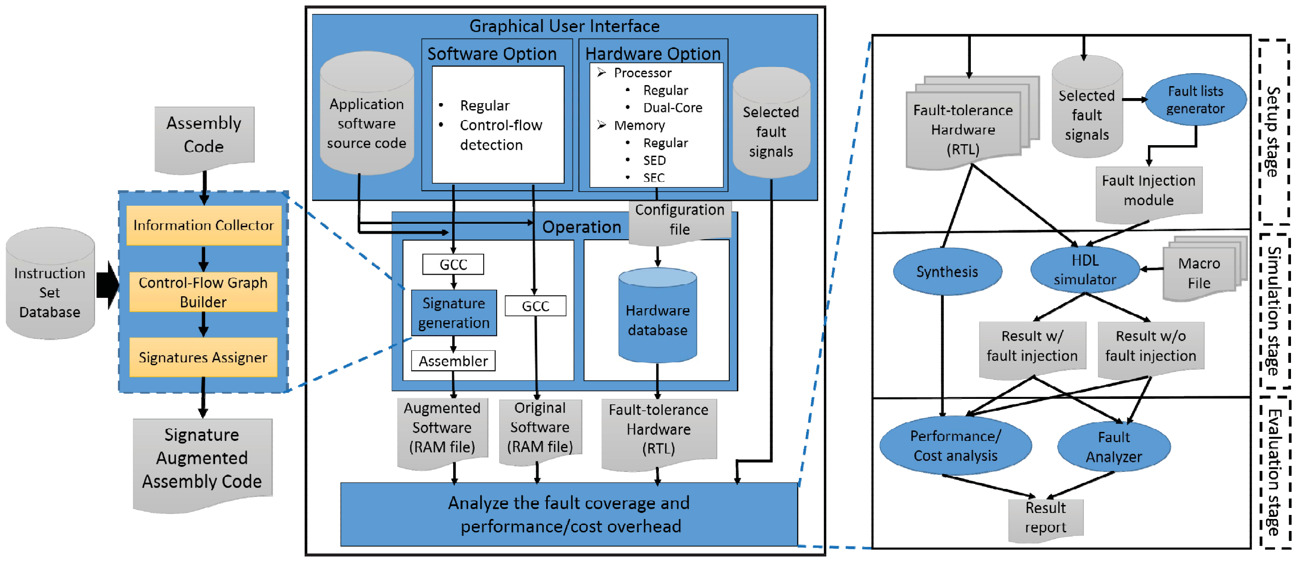
To address these problems, we propose an integrated design environment (IDE) for the development of fault tolerant processors. This IDE consists of a toolbox of hardware and software fault tolerant solutions. Currently the hardware solutions include dual redundancy for processor core, single-error-detection (SED) or single-error-correction (SEC) for memory. The software solution is control follow error detection. Given an original processor IP and its corresponding software, the IDE automatically enhances the processor, memory and software with fault tolerance mechanmisms that the designer selects from the toolbox. The enhancement is conducted in a non-intrusive way; i.e., there is no need to modify the internal structure of the processor. Therefore, it is suitable for IP-based design style. This IDE analyzes the performance, cost and power overheads and the fault coverage of the selected solutions, allowing the designers to explore the hardware/software design space. A GUI has been constructed to automate these design and analysis tasks, and is demonstrated with the IP core of an industrial processor core Andes N801S [1].

The rest of the paper is organized as follows. Section II reviews related work. Section III describes the IDE. Section IV presents experiment results. Section V concludes this paper.

II. RELATED WORK

Fault tolerance can be achieved with hardware, software or a combination of both. The hardware-based approaches add additional hardware modules, such as double module redundancy (DMR) [8], triple module redundancy (TMR) [22] or embedded debugger [9], to monitor the execution and detect error in real time with insignificant performance overhead, at the cost of less flexibility and chip area overhead.

In processor-based systems faults may affect the data flow or control flow of the software running on the system, and therefore may be detected with software-based approaches [4-7]. Data flow related faults are related to data manipulation and storage which may be detected by performing redundant



(a). Signature generation (b). Workflow of the IDE (c). Synthesis, simulation and analysis

Figure 1. Organization of the integrated development environment (IDE) for fault tolerant processors

data read and comparison operations with software instructions [23]. However, such approaches result in heavy performance/power/storage overheads due to frequent data operations, especially to the memory system. On the other hand, faults may also alter the control flow of software due to changes in the branch conditions, branch target addresses, opcodes, register index, *etc*. Such faults can be detected by instrumenting the software program with signatures and their update operations at appropriate locations at compile time. If the control flow is hampered by faults, it can be detected by checking the signatures at run time [6]. In summary, software-based approaches are flexible and require no hardware support. However, they incur overheads in the program size and software execution time. To reduce these overheads, specialized hardware modules may be used to manipulate the signatures so less instructions are required [10][11].

Fault injection plays an important role in analyzing the effectiveness of fault-tolerant mechanisms. Physical fault injection uses x-ray or laser to cause faults in the chip [12]. However, physical fault injection has some shortcoming that fault injection equipment is very expensive and hard to control how much faults and which signals are injected. Physical fault injection is more realistic, but less observable and controllable. In the software-based approach such as Fiesta [13], it injects faults by modifying the C code or assembly code. EDFI analyzes and duplicates the control flow to inject the faults [20]. GOOFI2 uses breakpoints, exceptions and modifies the assembly code to inject faults [21]. For simulation-based fault injection approaches, Kammler *et al*. uses the Verilog Programming Interface (VPI) to inject faults [14]. On the other hand, extra hardware modules can be used to generate the stuck-at faults and bit-flip faults in the circuit under test [15][16].

III. IDE FOR FAULT-TOLERANT PROCESSORS

The workflow of our IDE is shown in Fig. 1(b). The IDE has a GUI interface to select software or hardware options for fault tolerant capability. In order to make our IDE non-intrusive and cost-effective, we select DMR and SED/SEC, as the hardware options, to protect the processor and memory respectively. As for software options, we choose CEDA (Control flow Detection with Assertions) [6] to detect the control-flow error. Data-flow approaches are not selected due to their high performance and memory overheads in software. The regular options in the figure indicate the non-fault-tolerant design. This IDE adopt a simulation approach for fault injection due to its advantages in controllability, repeatability and observability.

After selecting those configurations, this IDE generates the processor and memory system with fault tolerant capability, as the fault-tolerance hardware (RTL) in the figure and analyze their overheads. On the other hand, the IDE compiles the application software from its source code to assembly code and instrument it with signatures if the CEDA software option is selected. The IDE also launches a fault simulation to analyze the fault coverage of the generated fault-tolerant system and software. The report is then shown in our IDE GUI.

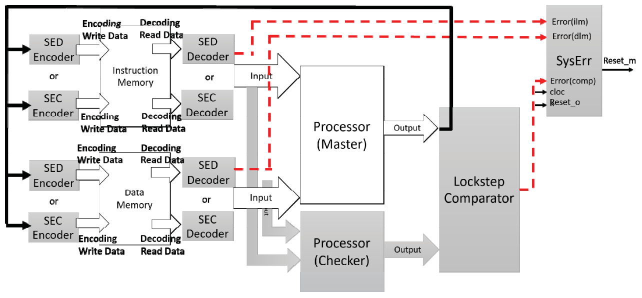


Figure 2. Non-intrusive hardware-based fault tolerance techniques for the processor and memory system

A. *The hardware-based fault-tolerance mechanisms*

Fig. 2 shows the block diagram of hardware fault-tolerant mechanisms. The grey blocks are the extra hardware to detect processor or memory faults.

(1). DMR Architecture for processor protection

We adopt a double core locked-step architecture as our DRM implementation to protect the processor. The two cores are identical, one as the master and the other as the checker. Both the master processor and the checker processor are connected to the same instruction and data memories. They run at the same clock frequency but only the master processor can modify the memory, I/O and control the system.

(2). SED and SEC for memory protection

Error detection/correction codes are effective ways to protect memory systems [18]. To be cost effective, we consider single-bit error detection (SED) and correction (SEC) with the parity code and the hamming code respectively. The parity code requires one extra bit for a multiple-bit data whereas the hamming code requires seven extra bits for a 32-bit data.

In Fig. 2, the instruction/data memory is protected by connecting an SED/SEC encoder and decoder to the memory's write/read port respectively. If the SEC detects a single bit error, it corrects the error and sends the corrected value to the processor. If the SEC detects multiple-bit error or the SED detects any parity error, they send the error signal to notify the system.

(3). Error detection modules

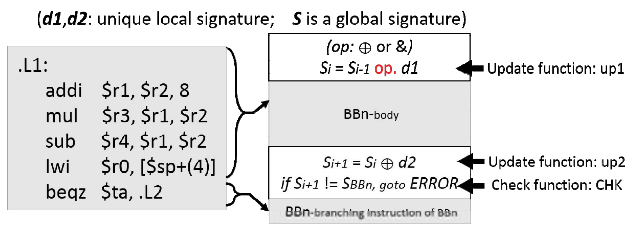
The address bus, data bus, memory write and request signals of the master and checker processors are checked by the *Lockstep Comparator* to ensure the correctness of the processor in a cycle-by-cycle fashion.

There is a module *SysErr*, which consolidates the original reset signal *Reset_o*, the error signals from the *Lockstep Comparator* and the SED/SEC decoders to generate the system reset signal *Reset_m*. Inside of *SysErr*, there is a reset-duration counter to hold *Reset_m* for a fixed cycle, which is specified by the designer, to reset the whole system.

B. *The software-based fault-tolerance mechanisms*

After evaluating appropriate software-based techniques, we choose CEDA [6] in our IDE due to its simplicity and moderate performance and storage overheads. The CEDA technique is performed at the assembly level[1]. Our IDE automatically analyze the assembly code of a given application program and then instrument it with signatures and their update/check operations to detect control flow error.

Fig. 3 illustrates CEDA technique at the assembly level. Each basic block is assigned with a unique signature. When entering a basic block *BBn*, its signature is updated with an appropriate value, denoted as the *UP1* operation in the figure. The operator in UP1 could be XOR or AND, depending on the type of the basic block, which will be discussed shortly. Upon leaving the basic block, its signature is updated again, the *UP2* operation in the figure, and is then checked, the *CHK* operation in the figure, to ensure if this basic block is entered from a legitimate source and is executed healthily. If the last instruction of the basic block is a control flow related instruction such as branches, jumps, calls, software interrupts, *etc*., denoted as *BBn-branching instruction*, it is split from the rest of the basic block, denoted as *BBn-body*, and is executed after the signature update and check operations as shown in the figure.


(d1,d2: unique local signature; S is a global signature)
(op: ⊕ or &)
$S_i = S_{i-1}$ op. $d1$
Update function: up1
BBn-body
$S_{i+1} = S_i \oplus d2$
if S_{i+1} != S_{BBn}, goto ERROR
Update function: up2
Check function: CHK
BBn-branching instruction of BBn

```
.L1:
    addi   $r1, $r2, 8
    mul    $r3, $r1, $r2
    sub    $r4, $r1, $r2
    lwi    $r0, [$sp+(4)]
    beqz   $ta, .L2
```

Figure 3. Update operations and check operation of CEDA technique for a basic block BBn

1) *Signature Generation*

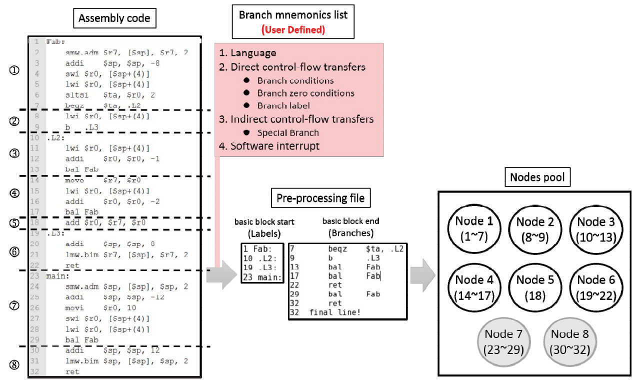
The signature can be stored in a memory location or a register. In our implementation, we store the signature in the register instead of memory since frequent memory accesses result in significant performance overhead. To achieve this goal, we first compile the source code of the application software using the *-ffixed-reg* option in GCC compiler to reserve a register so the application software never uses this register. It is solely used to manipulate signatures at run time. On the other hand, the update values of the signature and the predestinated signature value that is determined at the compile time, as *d1*, *d2* and S_{bbn} in Fig. 3, are embedded in the constant field of the instruction format and thus avoiding storing these values in memory in order to save the memory storage capacity and access time [4].

Once the source code is compiled into the assembly code, the *Signature Generation* process, depicted in Fig. 1(a) then automates the CEDA process. It divides the assembly code into basic blocks with *Information Collector*, builds the control flow graphs with *Control Flow Graph Builder* and inserts the operations and signatures with *Signature Assigner*. An *Instruction Set Database* is provided in order to make the *Signature Generation* process retargetable to different instruction sets. This automation process is not available in [6] and thus is considered as our contribution.

Fig. 4 presents the details of *Information Collector*. The given assembly code is processed to extract the line numbers in which branch instructions, labels, function names, return from function instructions (*ret*) and the end of function. The collected information is stored in a pre-processing file. Based on this file, a node structure is constructed with information such as *Label, Branch_Instruction, Begin_Line, End_Line* and *Node_ID.* In order to be retargetable to different instruction sets, information such as instruction formats, branch/jump/call/interrupt mnemonics are kept in the Instruction Set Database, as highlighted with a colored box in

[1] Alternatively, signatures may be inserted at higher levels such as at the high level language or intermediate code. However, compiler optimization could significantly modify the software structure and thus weaken the signature protection mechanism.

the figure. *Information Collector* is implemented as a Linux shell script because of its powerful string processing capability.



```
1  Fab:
2      smw.adm $r7, [$sp], $r7, 2
3      addi    $sp, $sp, -8
4      swi $r0, [$sp+(4)]
5      lwi $r0, [$sp+(4)]
6      sltsi   $ta, $r0, 2
7      beqz    $ta, .L2
8      lwi $r0, [$sp+(4)]
9      b    .L3
10 .L2:
11     lwi $r0, [$sp+(4)]
12     addi    $r0, $r0, -1
13     bal Fab
14     move    $r7, $r0
15     lwi $r0, [$sp+(4)]
16     addi    $r0, $r0, -2
17     bal Fab
18     add $r0, $r7, $r0
19 .L3:
20     addi    $sp, $sp, 8
21     lmw.bim $r7, [$sp], $r7, 2
22     ret
23 main:
24     smw.adm $sp, [$sp], $sp, 2
25     addi    $sp, $sp, -12
26     movi    $r0, 10
27     swi $r0, [$sp+(4)]
28     lwi $r0, [$sp+(4)]
29     bal Fab
30     addi    $sp, $sp, 12
31     lmw.bim $sp, [$sp], $sp, 2
32     ret
```

```
1 Fab:
10 .L2:
19 .L3:
23 main:
```

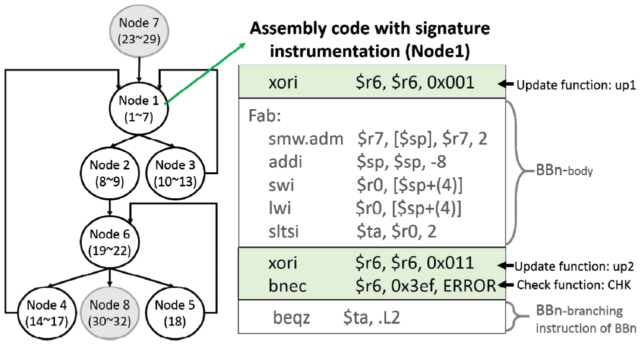
```
7       beqz    $ta, .L2
9       b       .L3
13      bal     Fab
17      bal     Fab
22      ret
29      bal     Fab
32      ret
32  final line!
```

Figure 4. CEDA process: basic block formation (Information Collector)

In the next step, *Control Flow Graph Builder* builds the control flow graph from the node structure by starting from the entry node in the node pool, as Node 7 of the example in Fig. 5. During processing, function call instructions, such as Nodes 3, 4, 7 in the figure. When processing those nodes, the return positions (Node 4, 5, 8) are also recorded by the Builder.

The nodes in the control-flow graph are classified into two types based on their reaching scenarios: X type and A type [6]. If a node can be reached from more than one predecessor and at least one predecessor has more than one successor, then this node is classified as A type; otherwise X-type.

Once the nodes are classified, we are now ready to generate the instrumented assembly code. For each basic block, the corresponding instructions for *UP1*, *UP2* and *CHK* operations are inserted to the appropriate locations, following the template in Figure 3. The *UP1* operation is an XOR operation and an AND operation for X type node and A type node respectively. The signature related values *d1/d2/*S_{bbn} are determined by *Signature Assigner*. The control flow graph constructed from the assembly code in Figure 4 is illustrated in Figure 5. In this example, all nodes are of X type. The instrumented assembly code for the basic block Node 1 is also shown in the figure.



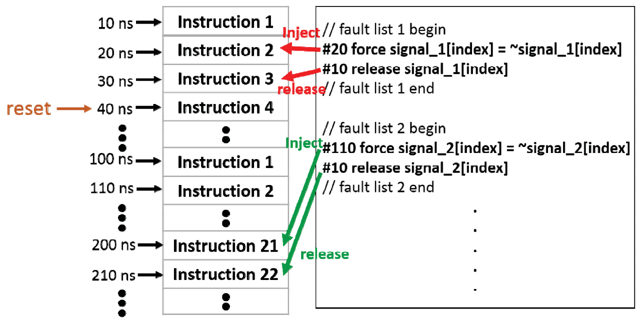
```
xori      $r6, $r6, 0x001
Fab:
   smw.adm   $r7, [$sp], $r7, 2
   addi      $sp, $sp, -8
   swi       $r0, [$sp+(4)]
   lwi       $r0, [$sp+(4)]
   sltsi     $ta, $r0, 2
   xori      $r6, $r6, 0x011
   bnec      $r6, 0x3ef, ERROR
   beqz      $ta, .L2
```

Figure 5. CEDA process : signature instrumentation (Control Flow Graph Builder and Signature Assigner)

C. Synthesis, simulation and analysis

Fig. 1(c) illustrates the *Synthesis, simulation and analysis* process, which consists of three phases. In the setup phase, based on the target signals and the number of faults specified by the designer, the IDE generates a fault injection module that applies faults to randomly selected signals at randomly selected time slots. For efficiency, we choose to inject only single bit error since double (or more)-bit error is reported to be no significant difference to single bit error (only with 4~5%) [19].

After that, in the simulation phase, the fault-tolerance hardware (RTL code) is synthesized to analyze its performance, power and cost. On the other hand, it is simulated in two versions: without the fault injection module to generate the golden result and with the fault injection module to generate the faulty result. For both results, information such as the program counter, corresponding instructions, the register values and memory read/write operations are collected with a monitor module.



```
// fault list 1 begin
#20 force signal_1[index] = ~signal_1[index]
#10 release signal_1[index]
// fault list 1 end

// fault list 2 begin
#110 force signal_2[index] = ~signal_2[index]
#10 release signal_2[index]
// fault list 2 end
```

Figure 6. Example of fault injection and simulation

The fault injection simulation is carried out by using *force*, *release* and # operators in Verilog to simulate single bit transient faults. The operator *force* aims to enforce a single bit flip to the signal, the operator *release* returns the handler to original signal and the operator # denotes the injection time. This simulation approach does not require any modification to the original HDL code and thus contribute to the retargetability of our IDE.

Fig. 6 provides an example of the fault simulation. At the left side is the instruction execution sequence, assuming each instruction takes 10ns for execution. At the right-hand side is the fault list generated by the fault generator. The first fault is injected at time 20ns and released at time 30ns which correspond to instruction 2 and instruction 3 respectively. The fault is detected and system is reset at time 40ns and therefore the execution restarts from instruction 1. The IDE removes the first fault from the fault list and applies the second fault at time 200ns. On the other hand, if the current fault is not detected by the fault-tolerance mechanisms in the system, the simulation continues without reset, and there is no need to delete the current fault from the fault list; the simulation goes directly to next faults at their designated times.

To speed up the fault simulation, it is possible to simply intercept the reset signal generated by the *SysErr* module and print them out to the GUI, and then continue the hardware execution without actually resetting the system to execute from the initial condition. This method could test all the target faults in a single simulation run, as opposed to repeating the simulation from the beginning for many times, and waiting a long time to reach target faults scheduled at much latter cycles.

At the last phase, the evaluation phase, we consolidate the synthesis results, the golden/fault simulation results to generate the analyses of hardware performance/power/cost, application software execution time and fault coverage. All the information in report is shown in GUI. It allows the user to trade off performance/cost/power and fault coverage with different hardware/software configuration.

IV. Experiments and Results

We demonstrate our IDE with a commercial 32-bit processor N801S [1], provided by Andes Technology [2] as a Verilog-based IP core. It has a proprietary 32-bit instruction set and an efficient three-stage pipeline targeted for embedded applications.

First, Table 1 shows the chip area and power consumption of the processor, memory and optional fault tolerance modules available in our toolbox. These hardware modules are synthesized under Synopsys SAED_EDK 90nm process. The basic modules include the processor core and the on-chip instruction and data memory, with total area of 306556um^2 and power of 3903uW. The rest of the modules are the optional fault tolerance modules.

Table 1. Chip area and power analyses of processor, memory and fault tolerance modules under Synopsys SAED_EDK 90nm process

HW Components		Area (um^2)	Power (uW)
Basic Modules	Andes N801s Processor	157056.14	3867.70
	Inst./Data SRAM (4KB each)	149499.39	35.80
Fault Tolerance Modules	SED: Parity Encoder	1971.55	341.15
	SED: Parity Decoder	4088.37	280.21
	SEC: Hamming Encoder	3065.68	604.77
	SEC: Hamming Decoder	12311.33	945.35
	Comparator	3328.59	110.73
	SysErr	845.65	5.11

We use our IDE to generate the fault tolerant processor system with three typical configurations of hardware and software options, as shown in Table 2: Configuration *A* selects SEC (single error correction) to protect the memory; Configuration *B* selects SED (single error detection) to protect the memory and CEDA to protect control flow execution in software; Configuration *C* selects DMR architecture and SED to protect the processor and memory respectively. The IDE analyzes hardware overheads of chip area and power consumption, and software overhead of execution cycles. The application software used in this Table is the bubble sort algorithm. For the analysis of fault coverage, 1000 faults are injected, with 45% to the processor and 55% to the memory, proportional to their area ratio. For processor, the faults are injected into the fetch, issue and ALU-related modules in the pipeline stages.

Table 2. Experiment results with various fault-tolerance configurations

Configuration					Overheads			Fault Injection
Case	HW: DMR Architecture	HW:SED	HW:SEC	SW: CEDA	Hardware		Software	Fault Coverage
					Area	Power	Execution Cycles	
A			✓		6.7 %	78.7 %		98.0 %
B		✓		✓	2.8 %	31.7 %	37.2 %	98.4 %
C	✓	✓			38.0 %	133 %		99.9 %

According to the experiment in Table 2, Configuration *A* uses only one hardware mechanism, SEC, to protect the memory with moderate 6.7% area overhead and 78.9% power overhead due to relative complicated Hamming encoding/ decoding computation. In total, this configuration achieves 98% fault coverage. On the other hand, Configuration *B* adopts a simpler memory protection mechanism, SED, which has much smaller area (2.8%) and power (31.7%) overhead. To compensate for the limited protection capability of SED, this configuration also adopts CEDA software instrumentation, which slows down the software execution by 37.2%. In total, Configuration *B* achieves slightly better fault coverage. Finally, Configuration *C* adopts the DMR architecture to protect the processor and SED to protect the memory, achieving the highest fault coverage of 99.9% at the cost of much higher area (39.4%) and power (133%) overheads.

We further investigate performance overheads of the CEDA software instrumentation in Configuration *B* with more application programs chosen from Mibench [17] as listed in Table 3. The overhead ranges between 27.6% and 43.1%.

The above analyses allow us to conduct versatile tradeoffs: if very high fault coverage is the major objective, Configuration *C* which protects both processor and memory with hardware options is the best selection. On the other hand, if low hardware cost is the major concern and moderate performance overhead is acceptable, Configuration *B* gives the best tradeoff. However, if performance overhead is to be avoided, Configuration *A* is a good compromise with slightly higher hardware cost and power consumption.

Table 3. Performance overhead of CEDA instrumentation in different benchmarks

Application Programs	Execution Time (cycles)		Performance Overhead
	Original	Signature instrumentation	
Bubble Sort	29168	37032	37.2%
Quick Sort	32311	40860	27.6%
Fibonacci	28082	34907	43.1%
Bit Count	46732	63025	34.2%
Basic Math	39431	47235	29.8%

V. Conclusion

We have presented an IDE for embedded fault-tolerant processor system. It takes in a processor core IP and the embedded software which is to be executed on the given processor, and turns them into a fault-tolerant system with various hardware and software mechanisms, subject to the designer's selection. A GUI is provided for the designer to select the options and the IDE automatically generates the hardware Verilog code and the modified embedded software. The IDE reports the cost, speed and power consumption of the generated hardware and the static and dynamic instruction counts of the generated software. In addition, a fault-injection tool is also provided to evaluate the fault coverage of the generated hardware and software. With this IDE, the designer could explore versatile tradeoffs of cost/performance/power/ fault-coverage. We have successfully demonstrated this IDE with a commercial processor core Andes N801S.

In the future, more options will be added to our IDE such as hybrid fault detection and recovery mechanisms. We will also demonstrate the IDE's retargetability with other instruction set processors. Furthermore, we will improve the CEDA technique by reducing the instruction overhead.

Acknowledgement

We would like to thank Andes Technology [1] for the discussion of the initial direction of this work and providing its N801S processor core IP, tool chain and technical support.

References

[1] AndesCore™ N801-S, http://www.andestech.com/product-details01.php?cls=3&id=4

[2] Andes Technology Corporation, http://www.andestech.com

[3] GCC, the GNU Compiler Collection, https://gcc.gnu.org/

[4] N. Oh, P.P. Shirvani and E.J. McCluskey, "Control-flow checking by software signatures", IEEE Transactions on Reliability, 2002.

[5] Aiguo Li and Bingrong Hong, "On-line control flow error detection using relationship signatures among basic blocks," Journal, Computers and Electrical Engineering archive, January, 2010, p.p 132-141

[6] R. Vemu and J. Abraham, "CEDA: Control-Flow Error Detection Using Assertions," IEEE Transactions on Computers, 2011., pp. 1233-1245

[7] S.A. Asghari, H. Taheri, H. Pedram and O. Kaynak, "Software-Based Control Flow Checking Against Transient Faults in Industrial Environments," IEEE Transactions on Industrial Informatics, 2013, p.p. 481-490

[8] C.El Salloum, A. Steininger, P. Tummeltshammer, W. Harter,"Recovery Mechanisms for Dual Core Architectures," International Symposium on Defect and Fault Tolerance in VLSI Systems, 2006, pp.380-388

[9] M. Grosso, M.S. Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil and L. Entrena, "An On-line Fault Detection Technique Based on Embedded Debug Features, " IEEE 16th International On-Line Testing Symposium, 2010.

[10] J.R. Azambuja, A. Lapolli, L. Rosa, and F.L. Kastensmidt, "Detecting SEEs in Microprocessors Through a Non-Intrusive Hybrid Technique," IEEE Transactions on Nuclear Science, 2011, vol.58, NO.3, JUNE

[11] L. Parra, A. Lindoso, M. Portela, L. Entrena, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martínez-Álvarez, "Efficient Mitigation of Data and Control Flow Errors in Microprocessors, " IEEE Transactions on Nuclear Science, 2014, VOL. 61, NO. 4, August.

[12] U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," International Symposium on Fault-Tolerant Computing, 1989, pp. 340-347

[13] N. Krishnamurthy, V. Jhaveri and J.A. Abraham, "A Design Methodology for Software Fault Injection in Embedded Systems," Proc. IFIP Int'l Workshop Dependable Computing and Its Applications (DCIA '98), 1998, pp. 237-248, Jan.

[14] D. Kammler, J. Guan, G. Ascheid, R. Leupers and H. Meyr, "A Fast and Flexible Platform for Fault Injection and Evaluation in Verilog-Based Simulations," Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009.

[15] M. Shokrolah-Shirazi and S.G. Miremadi, "FPGA-Based Fault Injection into Synthesizable Verilog HDL Models," Second International Conference on Secure System Integration and Reliability Improvement, 2008, pp. 143-149

[16] M.S. Shirazi, B. Morris and H. Selvaraj, "Fast FPGA-based fault injection tool for embedded processors," International Symposium on Quality Electronic Design, 2013, pp. 476-480

[17] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," IEEE International Workshop on Workload Characterization, 2001.

[18] B. Schroeder, E. Pinheiro and Wolf-Dietrich Weber, "DRAM errors in the wild: a large-scale field study," Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2009, pp. 193–204.

[19] F. Ayatolahi, B. Sangchoolie, R. Johansson and J. Karlsson, "A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution," International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2013, vol. 8153 p.p. 265-276.

[20] C. Giuffrida, A. Kuijsten and A.S. Tanenbaum, "EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments, " IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC), 2013.

[21] D. Skarin, R. Barbosa and J. Karlsson, "GOOFI-2: A Tool for Experimental Dependability Assessment, " IEEE/IIFIP International Conference on Dependable Systems & Networks (DSN), 2010.

[22] P.K. Samudrala, J. Ramos and S. Katkoori, "Selective Triple Modular Redundancy (STMR) Based Single-Event Upset (SEU) Tolerant Synthesis for FPGAs," *IEEE TRANSACTIONS ON NUCLEAR SCIENCE*, vol. 51, NO. 5, 2004

[23] A. Meixner and D.J. Sorin, "Error Detection Using Dynamic Dataflow Verification," 16th International Conference on Parallel Architecture and Compilation Techniques, 2007.