Compiler-Assisted Soft Error Correction by Duplicating Instructions for VLIW Architecture

Yunrong Li[1], Jongwon Lee[1], Yohan Ko[2], Kyoungwoo Lee[2], and Yunheung Paek[1]

[1]School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-744, Republic of Korea
{wylee, jwlee, ypaek}@sor.snu.ac.kr

[2]Department of Computer Science
Yonsei University, Seoul 120-749, Republic of Korea
{yohan.ko, kyoungwoo.lee}@yonsei.ac.kr

Abstract—**Exponentially increasing with technology scaling, soft errors have become a serious design concern in the deep sub-micron era. Error detection in VLIW or embedded systems is not enough while error correction is expensive due to the recovery mechanism. In this work, we present an enhanced VLIW architecture capable of not only error detection but also error correction by duplicating instructions efficiently, by re-executing the error-detected instruction, and by adopting the voting mechanism with the help of compilation techniques. Further, we propose a scheduling algorithm to improve the instruction scheduling and reliability over the executable under the performance constraint. Our experimental results on ADL-described VLIW datapath demonstrate that our solution efficiently improves the reliability by 29% over the suite of DSPStone benchmarks without performance overhead in our compiler-scheduler-simulator framework.**

I. Introduction

System reliability is becoming the paramount concern in system design in the deep sub-micron design era [1]. With technology scaling, i.e., shrinking feature sizes, decreasing voltage level, lower noise margins, etc., microprocessors are becoming increasingly prone to transient faults [8], [23]. A transient fault results in erroneous program states and eventually incorrect outputs, but it is temporary and non-destructive, i.e., resetting the device, restores normal behavior.

While transient faults may be caused due to several reasons, radiation-induced faults are responsible for more failures than all the other causes of transient faults combined [4]. Radiation-induced faults occur when a high energy radiation particle such as an alpha particle, a neutron and a free proton, strikes the diffusion region of a CMOS transistor and produces charge, which results in toggling the logic value of the transistor. This phenomenon of change in the logic state of a transistor is called a soft error or transient fault.

Soft errors for memory systems have been widely investigated and they are protected by error detection and correction codes (EDC and ECC) as a consequence. However, soft errors in logics are becoming also critical and take up more than 50% in overall soft errors in embedded systems [14]. Thus, researchers have presented several redundancy based techniques at various levels of design space abstraction, based on dual modular redundancy (DMR), triple modular redundancy (TMR), and checkpointing. However, these redundancy based techniques without optimization incur high overheads in terms of power, performance, and area. For example, TMR typically uses three functionally equivalent replicas of a logic circuit and a majority voter, but the overheads of hardware and power for conventional TMR exceed 200% [17].

VLIW (Very Long Instruction Word) architectures are of lots of interest in multimedia embedded systems because they are able to exploit high degrees of instruction parallelism with a reasonable tradeoff in complexity and cost [3]. Further, VLIW can provide the full control on scheduling of instructions at the compile time, which allows designers to manage multiple parameters such as power, performance, and reliability. Using replicated instructions and a software methodology to detect hardware faults in VLIW datapaths have been presented in the previous work [6], [10]. However, full duplications of instructions in VLIW logic architectures can incur high overheads in terms of performance and energy consumption [11], [12] while power and performance are important as well in embedded systems.

Jie Hu et al. [11], [12] has recently proposed a compiler-directed instruction duplication for VLIW architectures. Their approach is one of the most promising soft error detection for VLIW datapath that achieves the error resilience with low power and performance overheads. The main idea behind their work is to exploit the empty slots with duplicate instructions under the performance and energy consumption constraints. However, their solution is limited to error detection, which indicates the necessity of error recovery mechanism such as checkpointing. The absence of error recovery mechanism incurs the lack of soft error solution. Further, error recovery mechanism such as checkpointing is inappropriate for real-time embedded applications.

In this paper, we present an enhanced architecture of VLIW datapaths to detect and correct soft errors with least overheads in terms of area and performance. The main idea is to exploit empty slots for instruction duplication, to comare outputs of duplicated instructions, and to re-execute the instruction if the comparison returns the difference. Our approach is very effective and efficient since it does not triplicate the execution every instruction while it exploits a mechanism of TMR and it does not need use checkpoint-like mechanism for error recovery. Also, we propose a novel scheduling algorithm to intelligently utilize the empty slots for increasing reliability with minimal degradation of performance and to distribute duplicate instructions considering their priority over the scheduled executable.

The contributions and results of our work are:

- We propose a VLIW architecture with least area cost to correct soft errors by duplicating instructions and re-executing the erroneous instruction.
- Our compiler has full control of instruction scheduling for the proposed VLIW architecture and schedules instructions under performance constraint.
- Our scheduler is able to consider the priority among instructions to duplicate instructions with higher priority if they compete with others.
- Our proposal can efficiently increase the reliability in terms of duplications of instructions by 29% without performance overhead in terms of the code size and the execution time at minimal area cost.

II. Related Work

The primary source of transient faults in digital CMOS circuits are cosmic radiation. The phenomenon of radiation inducing faults has been under investigation for several decades. Due to incessant technology scaling (lower supply voltage and smaller feature size), the soft error rate (SER) has exponentially increased [8], [23], and now it has reached a point, where it has become a real threat to system reliability. Solutions to reduce the failures due to soft errors have been proposed at all levels of design hierarchy.

Logic elements were considered more resilient against soft errors than memory elements but several researchers predict that the logic SER will become one of main contributions to the system reliability [4], [17], [19]. The simplest and most effective way to reduce failures due to soft errors in combinational logic is TMR [18], which uses three functionally equivalent replicas of a logic circuit and a majority 2-out-of-3 voter. But the overheads of hardware and power for conventional TMR exceed 200% [17]. Duplex modular redundancy [15], [17] is also possible but still it requires more than 100% area and power overheads without any optimization techniques. In order to reduce the high overheads in conventional redundancy techniques, Mohanram et al. in [15] presented the partial error masking by duplicating the most sensitive and critical nodes in a logic circuit based on the asymmetric susceptibility of the nodes to soft errors. Recently, Nieuwland et al. [17] proposed a structural approach analyzing the SER sensitivity of combinational logic to identify the SER critical components at circuits.

Temporal redundancy is another main approach that has been used to combat soft errors in circuits. In order to detect soft errors, [16] applied fine time-grain redundancy within the clock cycle greater than the duration of transient faults by using the temporal nature of soft errors. Krishnamohan et al. in [13] proposed the time redundancy methodology by using the timing slack available in the propagation path from the input to the output in CMOS circuits. A Razor flip-flop was presented in [7] to detect transient faults by sampling pipeline stage values with a fast clock and with a time-borrowing delayed clock.

Since compiler is able to control all the schedules of instructions in VLIW architecture, several approaches have been proposed to duplicate instructions robust against soft errors at compile-time. The idea of replicated instructions in a VLIW processor has been investigated [10]. Bolchini [5] provides introduction of additional instructions to detect hard and soft errors for a VLIW datapath and compiler controls the replicated instructions so that it can carry out redundancy only for mission critical applications. However, full duplications of instructions in VLIW logic architectures can incur high overheads in terms of performance and energy consumption [11], [12]. Jie Hu et al. [11], [12] has recently proposed a selective duplication of instructions and its scheduling algorithm for VLIW architectures. Their approach is one of the most promising soft error detection in embedded systems because it can also consider and bound power and performance overheads. However, these previous works only detect soft errors while our work focuses on not only error detection but also error correction by modifying the previously proposed VLIW architecture. Further, we present a novel scheduling algorithm to consider priority of instructions for selective protection. Thus, the main contribution of this paper is in developing techniques to utilize unused empty slots in a 4-way VLIW datapath by assigning duplicate instructions into unused empty slots to improve the reliability without performance overhead because researchers have observed that the empty slots can take up more than 50% [21].

III. Our Architecture for Error Correction

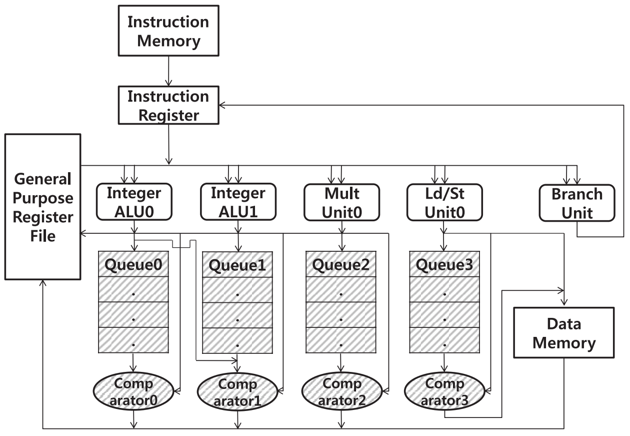
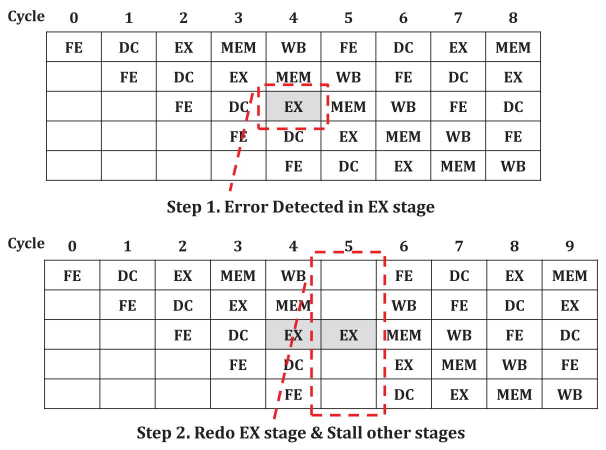


Fig. 1. VLIW Architecture for Error Detection and Correction

In this work, our VLIW datapath is composed of two integer ALUs, one integer multiplier, one load/store unit, and one branch unit as shown in Figure 1. The instruction duplication is the method to increase the reliability in our study. The main idea behind our approach for error correction is triple modular redundancy with the voting mechanism. However, this approach has been adopted to incur the third redundant instruction only when the first dual redundant instructions result in two different outputs, i.e., the original instruction and its duplicate instruction generate different outputs. By doing so, we can reduce the third redundant execution which would have been wasted if we would follow the conventional TMR approach.

To detect a soft error during the datapath, the output of an original instruction is compared to that of its duplicate instruction and if they are different, an error is detected. To correct a soft error during the datapath, the outputs of an original instruction and its corresponding duplicate instruction will be compared at the first step, then the duplicate instruction will be re-executed and its output will be compared to the original output. At this step, if they are identical, then this output and original output will be considered the correct one according to the 2 out of 3 voting mechanism as in TMR [18]. Note that address values are stored in the queue in case of load and store instructions. Figure 2 shows how pipeline works in our proposed VLIW architecture to correct an error if it is detected. This scenario shows that a soft error occurs during *EX* stage at cycle 4 (by duplicating an instruction and by comparing their outputs) and at the next cycle (cycle 5) this instruction is re-executed while other stages are stalled as shown in Figure 2. Note that soft error rate increases significantly with technology scaling but it does not occur every a few cycles. Therefore, it would be extremely expensive to execute each operation triple times to meet the TMR requirement.



Cycle	0	1	2	3	4	5	6	7	8
	FE	DC	EX	MEM	WB	FE	DC	EX	MEM
		FE	DC	EX	MEM	WB	FE	DC	EX
			FE	DC	EX	MEM	WB	FE	DC
				FE	DC	EX	MEM	WB	FE
					FE	DC	EX	MEM	WB

Step 1. Error Detected in EX stage

Cycle	0	1	2	3	4	5	6	7	8	9
	FE	DC	EX	MEM	WB		FE	DC	EX	MEM
		FE	DC	EX	MEM		WB	FE	DC	EX
			FE	DC	EX	EX	MEM	WB	FE	DC
				FE	DC		EX	MEM	WB	FE
					FE		DC	EX	MEM	WB

Step 2. Redo EX stage & Stall other stages

Fig. 2. Our pipeline configuration re-executes the detected erroneous stage and stalls other stages.

On the other hand, if both outputs from an original instruction and from its duplicate one are identical at the first step, they will be considered the correct one, i.e., there was no error in both units in the VLIW datapath. In order to support this error correction mechanism, the queue entries and comparators are introduced as shown in Figure 1.

Our redundancy technique to increase the reliability is to duplicate instructions. Thus, the more duplicate instructions cause the higher reliability in our VLIW datapath. To indicate whether this instruction is the original one or the duplicate one, one bit (*B1*) is introduced in VLIW architecture. If *B1* is set to 1, it indicates the original instruction. Otherwise (*B1* = 0), it indicates the duplicate instruction. In our VLIW architecture described in Figure 1, two integer ALUs are included while one integer multiplier and one load/store unit are considered. Thus, two integer ALU operations can be scheduled at the same time. Then, the results of two integer ALU operations such as addition can be compared immediately without the result of original one being stored in the queue. On the other hand, if the original instruction is scheduled before the duplicated one, the output of the original instruction should be stored in the queue and be compared to that of its duplicated one. The contents of the registers and/or the memory are not updated by the output of the original instruction. They are updated after that of the corresponding instruction is compared and confirmed if there was no error in both instructions. Note that the inconsistency between contents of register files and queues are avoided by our scheduling algorithm that considers dependencies among scheduled instructions. Therefore, another bit (*B2*) is introduced to indicate whether this output of instruction needs to be stored in the queue and to be compared to the entry in the queue. *B2* is reset to 0, if both an original and its duplicate instructions are scheduled at the same time. *B2* is set to 1, if they are scheduled at different times, i.e., the original instruction needs to write its output to the queue while the duplicate instruction needs to compare its output to the corresponding output from its original instruction stored in the queue.

IV. Our Scheduling Algorithm: Compiler Assist

Our scheduling algorithm, *DIScheduler*, takes two inputs: i. a sequence of instructions scheduled using a list scheduler (*region*) and ii. allowable increase in schedule length (*f*), and searches for a scheduled set of instructions including original and duplicate instructions with the schedule length less than or equal to $(1 + f) \times C$ where C is the schedule length with original instructions only. *DIScheduler* is composed of two rounds. The first round duplicates instructions unless they incur performance overhead and marks instructions with the calculated priority which need extra cycles. The priority can be determined by a feature of applications, preferred operations like addition, or soft error susceptibility. In the second round, *DIScheduler* duplicates instructions with higher priority while not incurring more overheads than the predefined constraint on code size.

Figure 3 shows the outline of our double-round scheduling algorithm. Lines 1 to 13 presents the first round pseudo code and lines 14 to 42 presents the second round pseudo code. First off, our algorithm generates a list of duplicate instructions list_of_dupl_inst_w_priority from *make_dupl_inst_w_priority()* considering the priority (line 1). The order of instructions in list_of_dupl_inst_w_priority depends on heuristics that we have developed. In our im-

```
DIScheduler (region, f)
   /* first round */
01: $list\_of\_dupl\_inst\_w\_priority = make\_dupl\_inst\_w\_priority()$
02: for (each dupl_op from list_of_dupl_inst_w_priority)
03:     $orig\_op = get\_orig\_op\_from\_dupl\_op(dupl\_op)$
04:     $etime = get\_earliest\_sched\_time(orig\_op)$
05:     $ltime = get\_latest\_sched\_time(orig\_op)$
06:     $stime = get\_sched\_time(orig\_op)$
07:     if $(sched\_success(dupl\_op, stime))$
08:         $list\_of\_dupl\_inst\_not\_inc\_cycle.add(dupl\_op)$
09:     else
10:         $list\_of\_dupl\_inst\_inc\_cycle.add(dupl\_op)$
11:     endIf
12: endFor
13: $list\_of\_dupl\_inst = list\_of\_dupl\_inst\_not\_inc\_cycle$
                        $+list\_of\_dupl\_inst\_inc\_cycle$
   /* second round */
14: $inc\_cycle = f \times C$
15: $max\_sched\_cycle = C - 1$
16: for (each dupl_op from list_of_dupl_inst)
17:     $orig\_op = get\_orig\_op\_from\_dupl\_op(dupl\_op)$
18:     $etime = get\_earliest\_sched\_time(orig\_op)$
19:     $ltime = get\_latest\_sched\_time(orig\_op)$
20:     $stime = get\_sched\_time(orig\_op)$
21:     while $(!sched\_success(dupl\_op, stime))$
22:         $stime++$
23:         if $(stime > ltime)$
24:             if $(inc\_cyle > 0)\&\&(shift\_for\_dupl\_not\_violate\_dep())$
25:                 $shift\_point = comp\_shift\_point(orig\_op, stime)$
26:                 if $(shift\_point! = -1)$
27:                     $shift\_down\_sched\_table(shift\_point, max\_sched\_cycle)$
28:                     $inc\_cycle--$
29:                     $max\_sched\_cycle++$
30:                     if $(shift\_point < stime)$
31:                         $stime = shift\_point$
32:                     elsif $(shift\_point > stime)$
33:                         $stime--$
34:                     endIf
35:                 endIf
36:             else
37:                 $delete\ dupl\_op$
38:                 $break$
39:             endIf
40:         endIf
41:     endWhile
42: endFor
```

Fig. 3. DIScheduler – Duplicate_Instruction Scheduling Algorithm

plementation, the instructions of list_of_dupl_inst_w_priority is sorted by the height of each instruction in data dependence graph. Note that *make_dupl_inst_w_priority()* ignores branch and move operations since they are not considered for instruction duplication in our work due to their feasibility in our architecture. For each duplicate instruction in list_of_dupl_inst_w_priority (lines 2-12), it gets original instruction from *get_orig_op_from_dupl_op (dupl_op)* and calculates the earliest schedule time (*etime*), the latest schedule time (*ltime*), and the initial schedule time (*stime*) (lines 3-6). If dupl_op can be scheduled without performance overhead, it is included in the list_of_dupl_inst_not_inc_cycle (lines 7-8). Otherwise, it is in the list_of_dupl_inst_inc_cycle (lines 9-10). For the further scheduling in the second round, both lists are combined into list_of_dupl_inst (line 13). inc_cycle and max_sched_cycle are initialized (lines 14-15) in the beginning of the second round. For each duplicate instruction in list_of_dupl_inst (lines 16-42), it gets original instruction and calculates the earliest schedule time (*etime*), the latest schedule time (*ltime*), and the initial schedule time (*stime*) (lines 17-20). During the while loop in lines 21-41, when dupl_op is not schedulable within stime, stime is increased (line 22). If stime is less than or equal to ltime, it does nothing during this while loop. If stime is larger than ltime and if inc_cycle is larger than zero and shifting duplicate instruction does not violate dependence, shift_point is computed with the inputs of orig_op and stime (lines 24-25). Otherwise, dupl_op is deleted and it breaks (lines 36-39). if the calculated shift_point (line 25) is not equal to -1, it shifts down the instruction in the schedule table, decreases inc_cycle by 1, and increases max_sched_cycle by 1 (lines 26-29). Further, if shift_point is less than stime, it sets shift_down to stime (lines 30-31). If shift_point is larger than stime, it decreases stime by 1 (lines 32-33).

TABLE I
CODE SEGMENTS

A:	Add	r2, r1, r3
B:	Load	r4, r2
C:	Mul	r6, r5, 2
D:	Add	r2, r1, 4
E:	Store	r2, r6

To describe the processes for instruction duplication, we have chosen a set of code segments including integer ALU operation (Add), integer multiplication (Mul), and Load (Load) and Store (Store) operations with some dependencies among operations as shown in Table I. r*N* indicates a *N* numbered register in register files. And the first operand is the destination and the second and third operands the sources. For example, operation *A: Add r2, r1, r3* executes addition of contents in r1 and r3, and stores the result in r2. Note that Table I shows the code segments in assembly before applying our VLIW scheduling algorithm for them.

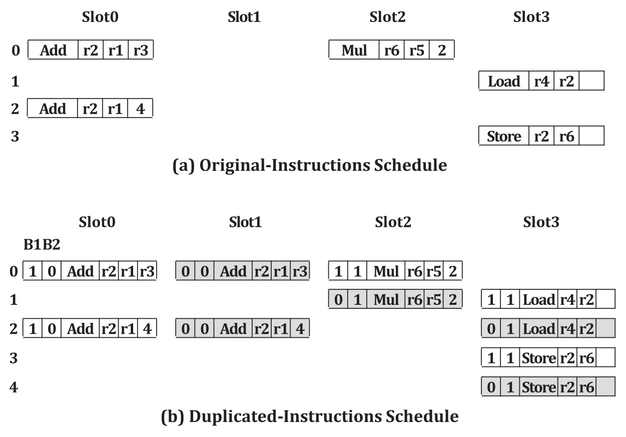


Fig. 4. Original Instructions Schedule and Duplicate Instructions Schedule

Our original compiler schedules the code segment from Table I and generates the scheduled instruction sequences for our VLIW architecture as shown in Figure 4. The top scheduling in Figure 4 presents several empty slots in 4-way VLIW datapath that are unused each cycle, i.e., NOPs.

Our objective is to insert duplicate instructions as many as possible to increase the reliability without incurring performance overhead. Our double-round scheduling algorithm finds out an efficient schedule including original and duplicate instructions under performance constraint by also being able to consider the priority of instructions and to balance the duplicate instructions among the executable. Finally, we can generate the schedule code including the duplicate instructions (shaded ones in Figure 4) and original instructions without breaking the dependency among instructions. Note that our scheduling algorithm takes dependency and VLIW configuration into account without breaking the consistency among register contents between register files and temporary data queue entries. Figure 4 also shows the *B1B2* indication in shaded duplicate instructions as described in Section III.

V. Experiments

A. Experimental Setup

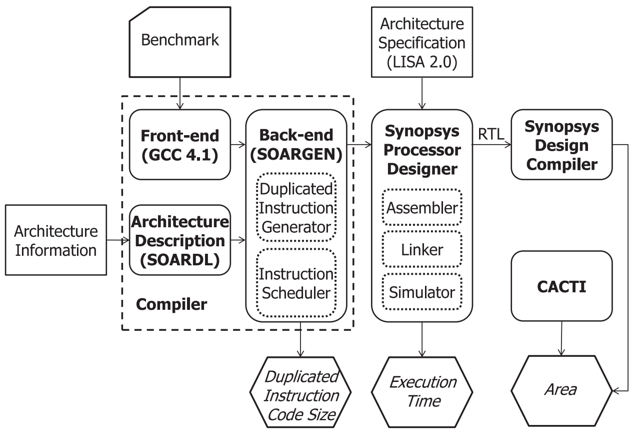


Fig. 5. Experimental Setup – Compiler-Scheduler-Simulator Framework

Our experiments have been performed in compiler-scheduler-simulator framework that we have developed as shown in Figure 5. We propose soft error correctable 32 bit embedded 4-way VLIW processor, and its datapath is composed of two integer ALUs, one integer multiplier, and one load/store unit with 16 entries of queues and comparators for soft error correction. Our proposed architecture has been written and implemented in LISA 2.0 language [20] to generate the target assembler, linker, and simulator.

Our compiler uses GNU gcc 4.1 c-compiler as front end and SoarGen compiler, a retargetable compiler platform [2], as back end. We describe the ISA of our proposed VLIW architecture using an architecture description language (ADL), SoarDL [2] to generate the target assembler. At this step, our framework returns the duplicated instruction code size as the first output and they are used to estimate the performance overhead in the code size and the reliability in the amount of the duplicate instructions as shown in Figure 5. All benchmarks from the DSPstone [22] have been simulated on the Synopsys Processor Designer [20]. At last, Synopsys simulator runs the executables of benchmarks and returns the run-time performance in the estimated execution time as shown in Figure 5. Also, Synopsys design compiler with CACTI [9] estimates and returns the area of our proposed VLIW architecture.

We have evaluated our proposed architecture and the scheduling algorithm in terms of reliability and performance. We assume that soft errors can occur in a normal distributed manner in time and place wise. Also we assume that the more duplicate instructions indicate the higher reliability of the VLIW datapath. The reliability is estimated in the ratio of the number of duplicate instructions to that of original instructions. The performance is estimated in two folds: (i) the ratio of the code size after duplication to that before duplication as the compile-time performance metric and (ii) the ratio of the execution time after duplication to that before duplication as the run-time performance metric. Note that one cycle overhead incurs if a soft error occurs in our proposed VLIW architecture and it is detected as presented in Section III.

B. Experimental Results

Minimal Area Cost Our estimation shows that area overhead for error correction in VLIW architecture is extremely small. The area overhead is estimated at just 1.69% for our proposed error correctable 4-way VLIW architecture with 8 MB of instruction and 8 MB of data memories. The area overhead is calculated by $overhead = (our\_architecture - base\_architecture)/(base\_architecture + memory)$ where architecture size is estimated by using Synopsys design compiler with 130 nm technology and memory size by scaling the result from CACTI with 90 nm to 130 nm technology. The estimated areas are 13606315 μm^2, 751540 μm^2, and 513616 μm^2 for $memory$, $base\_architecture$, and $our\_architecture$, respectively. These area overhead results show the effectiveness of our proposed architecture on the area cost.

Effectiveness on Reliability and Performance Our first set of experiments is to estimate the increased reliability without performance overhead. Figure 6 shows the effectiveness of our proposed techniques in terms of reliability. Our proposed soft error correctable VLIW architecture is able to increase duplicate instructions by 29% on average over the benchmarks without increase of code size as shown in Figure 6. Especially, it can duplicate about 50% instructions for a benchmark, iir_biquad_one_section. It is very effective because our solution exploits the empty slots, which would have been wasted, to increase the reliability without performance overhead.

Our second set of experiments is to maximize the reliability, i.e., fully duplicate all possible instructions, to see the performance overheads. Figure 7 shows the performance overheads in terms of increase of scheduled code size (compile-time performance metric) and increase of execution time (run-time performance metric) over benchmarks. Full duplication of instructions increase the code size by 46% and the execution time by 41% on average as shown in Figure 7. These results

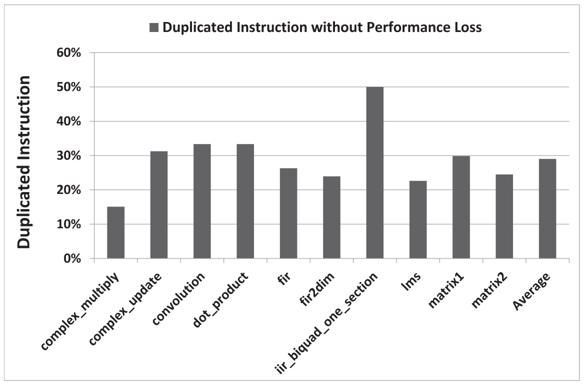


Fig. 6. Duplicated Instructions without Performance Degradation

are still effective because a conventional TMR approach can incur overheads in terms of power, performance, and area by up to 200% [17] and they would execute all instructions triple times while our solution executes some of interesting operations twice and re-executes it once again if an error is detected, which reduces in practice unnecessary executions and further expands the tradeoff space between performance and reliability significantly.

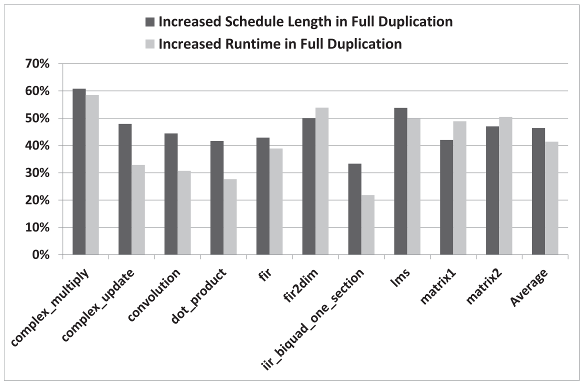


Fig. 7. Scheduled Code Length and Runtime for Full Duplicate Instructions

VI. Summary

Owing to the incessant technology scaling, soft errors, especially in datapaths are becoming a critical design concern for embedded system reliability. Detecting errors needs a recovery mechanism that is in general expensive. Thus, we focus on not only error detection but also error correction with minimal overheads and propose a compiler-assisted soft error correction architecture for VLIW where instructions can be duplicated, a soft error is detected by comparing two outputs from both duplicate instructions, and further the detected soft error can be corrected by re-executing the duplicate instruction once again and by re-comparing the stored output for the voting mechanism to correct the output. To support instruction duplication in an efficient manner, we also propose a scheduling algorithm which is composed of two steps to distribute duplications and to prioritize the candidates of instructions. We have implemented compiler-scheduler-simulator framework and our experimental results demonstrate the efficacy of our proposal for high reliability with minimal performance and area overheads. Further, our work expands the interesting design space to explore the tradeoffs between performance and reliability. Our future work includes enhanced scheduling algorithm to apply the importance of instructions for duplicate instruction selections and algorithm developments for design space exploration.

References

[1] International technology roadmap for semiconductors 2005.

[2] M. Ahn and Y. Paek. Transactions on high-performance embedded architectures and compilers ii. chapter Fast Code Generation for Embedded Processors with Aliased Heterogeneous Registers, pages 149–172. 2009.

[3] G. Ascia, V. Catania, M. Palesi, and D. Patti. A system-level framework for evaluating area/performance/power trade-offs of vliw-based embedded systems. In *ASPDAC*, pages 940–943, 2005.

[4] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, pages 258–266, 2005.

[5] C. Bolchini. A software methodology for detecting hardware faults in vliw data paths. *IEEE Trans. on Reliability*, 52:458–468, 2003.

[6] C. Bolchini and F. Salice. A software methodology for detecting hardware faults in vliw data paths. In *DFT*, pages 170–175, 2001.

[7] D. Ernst, N. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.

[8] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. on Nuclear Science*, 47(6):2586–2594, 2000.

[9] Hewlett Packard, http://www.hpl.hp.com/research/cacti/. *CACTI - An ingegrated cache and memory access time, cycle time, are, leakage, and dynamic power model*.

[10] J. G. Holm and P. Banerjee. Low cost concurrent error detection in a vliw architecture using replicated instructions. In *ICPP*, 1992.

[11] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-assisted soft error detection under performance and energy constraints in embedded systems. *ACM Transactions on Embedded Computing Systems*, 8:27:1–27:30, July 2009.

[12] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction duplication for soft error detection. In *DATE*, pages 1056–1057, 2005.

[13] S. Krishnamohan and N. R. Mahapatra. An efficient error-masking technique for improving the soft-error robustness of static cmos circuits. In *SOCC*, Sep 2004.

[14] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb 2005.

[15] K. Mohanram and N. A. Touba. Partial error masking to reduce soft error failure rate in logic circuits. In *DFT03*, pages 433–440, 2003.

[16] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *VTS'99*, 1999.

[17] A. K. Nieuwland, S. Jasarevic, and G. Jerin. Combinational logic soft error analysis and protection. In *IOLTS06*, 2006.

[18] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall, 1996. ISBN 0-1305-7887-8.

[19] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *DSN02*, 2002.

[20] Synopsys Inc., Mountain View, CA, USA. *Design Compiler Reference Manual*, 2001.

[21] D. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

[22] V. živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *ICSPAT*, 1994.

[23] F. Wrobel, J. M. Palau, M. C. Calvet, O. Bersillon, and H. Duarte. Simulation of nucleon-induced nuclear reactions in a simplified sram structure: Scaling effects on seu and mbu cross sections. *IEEE Trans. on Nuclear Science*, 48(6):1946–1952, 2001.