A Low-Cost Mechanism Exploiting Narrow-Width Values for Tolerating Hard Faults in ALU

Seokin Hong and Soontae Kim, *Member, IEEE*

Abstract—Digital circuits are expected to increasingly suffer from more hard faults due to technology scaling. Especially, a single hard fault in ALU (Arithmetic Logic Unit) might lead to a total failure in processors or significantly reduce their performance. To address these increasingly important problems, we propose a novel cost-efficient fault-tolerant mechanism for the ALU, called LIZARD. LIZARD employs two half-word ALUs, instead of a single full-word ALU, to perform computations with concurrent fault detection. When a fault is detected, the two ALUs are partitioned into four quarter-word ALUs. After diagnosing and isolating a faulty quarter-word ALU, LIZARD continues its operation using the remaining ones, which can detect and isolate another fault. Even though LIZARD uses narrow ALUs for computations, it adds negligible performance overhead through exploiting predictability of the results in the arithmetic computations. We also present the architectural modifications when employing LIZARD for scalar as well as superscalar processors. Through comparative evaluation, we demonstrate that LIZARD outperforms other competitive fault-tolerant mechanisms in terms of area, energy consumption, performance and reliability.

Index Terms—Hard faults, arithmetic and logic units, fault tolerance, processor architectures

1 Introduction

Although advances in technology provide significant benefits in performance, density and energy efficiency of processors, the probabilities of hard (or permanent) faults also increase. Hard faults can be manifested during fabrications due to manufacturing defects and process variations, which reduce processor yield [1]. They can occur during system operation due to electro migration, stress migration, thermal cycling, time-dependent dielectric breakdown, and negative bias temperature instability, which reduce lifetime reliabilities of the processors [2]. All these faults make the processors more vulnerable and increase their design complexities. Especially, ALU is very susceptible to these faults in embedded processors (i.e., scalar processors) because they generally have a single ALU and a single hard fault in the ALU could make the entire processors obsolete. In case of high performance processors (i.e., superscalar processors), there are multiple ALUs for enhancing performance. Thus, even though some of the ALUs are faulty, the processor can extend its lifetime by disabling the faulty ALUs. However, the permanent loss of them may result in significant performance degradation.

The conventional way of tolerating hard faults in ALU is to employ hardware redundancy. For example, triple modular redundancy (TMR) can be applied to the ALU. TMR requires three identical ALUs and a majority voter. The ALUs operate concurrently and their outputs are voted on to produce a correct output. This approach is efficient in terms of performance but significantly increases hardware costs in terms of area and energy consumption. Another way is to use time redundancy. For example, RESO [3] performs computations with the original values first, and then with shifted values again. The result of the second computation is recovered to the actual result, and then it is compared with the result of the first computation for fault detection. However, this approach reduces performance significantly.

In this paper, we propose a fault-tolerant mechanism called **LIZARD** that can combat various types of hard faults in ALU. Our goal is to design a cost-efficient fault-tolerant ALU, which achieves both the performance benefit of hardware redundancy and the cost-efficiency of time redundancy. Key observation is that a wide range of applications use popularly narrow-width values whose upper bits are all zeros or ones [4], [5], [6], [7], [8], [9], [10], [11]. When an ALU performs operations with at least one narrow-width operand (value), the upper bits of their results can be obtained without computations. This is because the upper bits are also all zeros or all ones, or identical with one of their operands in most cases. By exploiting this observation, LIZARD employs two half-word ALUs to generate two copies of lower half-word of results, which are compared to detect faults, while predicting the upper half-word of results. When a fault is detected, the half-word ALUs are partitioned into four quarter-word ALUs, and then they perform a same operation simultaneously. After that, by voting their results, we can identify a faulty quarter-word ALU. LIZARD isolates the faulty quarter-word ALU and continues its operation with the remaining three fault-free quarter-word ALUs. These fault-free ALUs can survive another fault in a similar way.

We also present architectural modifications for both scalar and superscalar processors when employing LIZARD. Even if LIZARD performs most operations in a single cycle, it consumes multiple cycles when the upper bits of their

• *The authors are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea. E-mail: {seokin, kims}@kaist.ac.kr.*

Manuscript received 23 July 2013; revised 27 Sept. 2014; accepted 8 Oct. 2014. Date of publication 3 Nov. 2014; date of current version 12 Aug. 2015. Recommended for acceptance by E. Rotenberg.
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.2014.2366743

results are not predictable. Thus, the execution latency of ALU becomes variable with LIZARD, which should be addressed to minimize its performance impact. We use the inherent pipeline stall logic to support this variable latency execution for scalar processors. For superscalar processors, we introduce an efficient technique predicting execution latency and extend the instruction issue logic.

Experimental results show that our fault-tolerant mechanism significantly improves the reliability of the ALU with low hardware costs. Moreover, our mechanism adds trivial performance and energy overheads for both scalar and superscalar processors with minimal architectural modifications.

The rest of this paper is organized as follows. Section 2 presents our motivational results. Section 3 explains the proposed fault-tolerant mechanism and Section 4 presents its detailed implementation. Section 5 presents processor architecture design using the proposed mechanism. Section 6 discusses experimental results and we present related work in Section 7. Finally, the conclusions are given in Section 8.

2 Predictable Instructions

Our mechanism is based on the key observation that a large number of ALU operands are narrow-width values of which upper bits are all zeros or ones [4], [5], [6], [7], [12]. In modern processors, the ALU is used to execute arithmetic and logical instructions. It is also used to calculate the effective addresses for memory and branch instructions. When the ALU executes these instructions with at least one narrow-width operand (values), the upper bits of their results are all zeros, all ones, or identical with that of the other operand in most cases. If both operands are narrow, the upper bits of results are mostly all zeros or all ones (e.g., 0x000000f0+0x00001100=0x**0000**11f0). If one of the operands are narrow, the upper bits of results are mostly identical with that of the other operand (e.g.,0x000000f0 +0x**ffeeff**00=0x**ffeeff**f0).

Fig. 1 shows the distributions of the upper bits of results produced by ALU instructions, which is defined as instructions using the ALU, in the Mibench [13] and SPEC CPU2006 benchmarks. Since these benchmarks are compiled respectively with 32-bit ARM and 64-bit ALPHA gcc compiler, the word width are 32 and 64 bits wide, respectively; the experimental methodology is presented in Section 6 in more detail. In case of the Mibench benchmarks, the upper half-word (16 bits) of results is all zeros or ones, or identical with that of the first or second operand for 94 percent of the ALU instructions. For 74 percent of the ALU instructions, even upper three-quarter-word (24 bits) of results has this characteristic. We observed similar results for SPEC CPU2006 benchmarks. The upper half-words and three-quarter-word of results are all zeros or ones, or identical with that of the first or second operand, respectively for 98 and 91 percent of the ALU instructions.

From this observation, we can conclude that most ALU instructions can predict the upper bits of their results by a simple heuristic, but not by a computation. We call these instruction **Predictable instruction (P-inst)**. Predictable instructions are further classified into **Moderately Predictable instructions (MP-inst)** and **Highly Predictable instructions (HP-inst)**, depending on the number of the predictable portion of results. For MP-insts, at most upper half-word of their results can be predictable while at least upper three-quarter-word can be predictable for HP-insts.

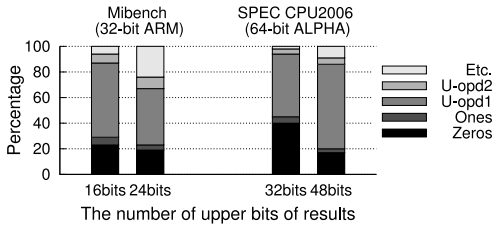


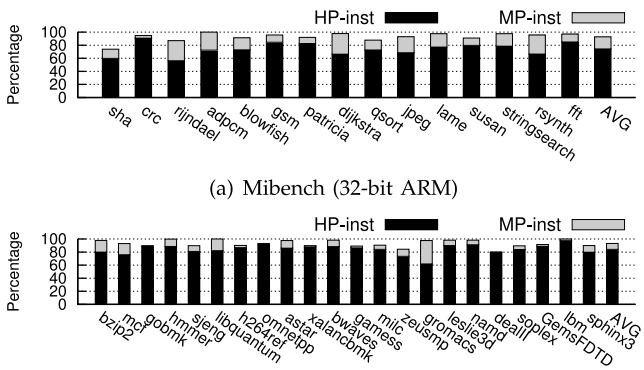
Fig. 1. Distribution of upper bits values of results. U-opd1 denotes upper bits of the first operand while U-opd2 denotes those of the second operand.

In order to determine whether a given instruction is a P-inst (either HP- or MP-inst) and to predict the upper bits of its result, three attributes of instructions are considered. First one is the upper bits of operands (i.e., whether the upper half or three-quarter-word of operands are all zeros or ones). Second one is the operation types of the instructions: ADD (Addition), SUB (Subtraction), AND, etc. Finally, for instructions performing arithmetic operations (ADD and SUB), a carry signal, generated in a computation with lower half or quarter-word of operands, are further considered. Table 1 summarizes the predicted upper bits of results for each case of P-insts which are denoted by gray color. For example, when an instruction performs the 'AND' operation, the upper bits of its result are predicted as all zeros whenever those of at least one operand are all zeros. In another case, when an instruction performs the 'ADD' operation and the upper bits of its first operand are neither all zeros nor all ones and those of second operand are all ones, the upper bits of its result are predicted as those of the first operand on presence of the carry signal. On absence of the carry signal, however, the instruction is considered as ***non-predictable instruction (NP-inst)***. In binary arithmetic, 'SUB' operation is typically performed using two's complement 'ADD' operation where the second operand is inverted and added to the first operand, and then '1'

TABLE 1
The Predicted Upper Bits of Results for Each Case of Predictable Instructions

InstOP	C	Upper bits of operands								
		1/1	0/0	1/0	0/1	*/1	*/0	1/*	0/*	*/*
ADD	0	·	0	1	1	·	a	·	b	·
	1	1	·	0	0	a	·	b	·	·
SUB	0	1	1	·	0	a	·	·	·	·
	1	0	0	1	·	·	a	·	·	·
AND	·	1	0	0	0	a	0	b	0	·
OR	·	1	0	1	1	1	a	1	b	·
XOR	·	0	0	1	1	·	a	·	b	·

***InstOP**: operation of the instructions, **C**: carry generated in a computation with lower bits of operands, **a/b** : a - first operand, b - second operand ('0' and '1' indicate all zeros and ones while '*' indicates other values), **Gray-colored cells**: Predictable instructions, **Value in a cell**: predicted upper bits of result ('0' and '1' indicate respectively all zeros and ones while 'a' and 'b' indicates the upper bits of the first and second operand, respectively).*



(a) Mibench (32-bit ARM)

(b) SPEC CPU2006 (64-bit ALPHA)

Fig. 2. The percentages of predictable instructions.

is added to the sum. Thus, the upper bits of results can be predicted in a similar way to the 'ADD' operation.

Fig. 2 shows the distribution of P-inst in the Mibench and SPEC CPU2006 benchmark suits. P-inst accounts for more than 92 percent of total ALU instructions; on average, HP-inst and MP-inst account for 74 and 18 percent, respectively, in the Mibench benchmarks. In the SPEC CPU2006 benchmarks, the P-inst represents a greater share of total ALU instructions; HP-inst and MP-inst account for 84 and 10 percent, respectively, on average.

3 LIZARD: A Cost-Efficient Fault-Tolerant Mechanism for the ALU

3.1 Fault Detection and Diagnosis

Concurrent fault detection validates every instruction to detect various types of hard faults, not only manufacturing defects but also wearout-induced faults. When a fault is detected, it is required to find a fault location to be masked or repaired; this process is called fault diagnosis. The simplest approach for the concurrent fault detection and the fault diagnosis is tripling the original hardware and compare their outputs. However, this approach is not efficient in terms of area and energy consumption.

By exploiting the prevalence of P-inst, we propose cost-efficient fault detection and diagnosis techniques for ALU. For executing P-insts, a half-word ALU is sufficient. Thus, we employ two half-word ALUs to execute the ALU instructions while detecting faults concurrently. If an instruction is a P-inst (either MP- or HP-inst), the lower half-word of its operands is duplicated on the two half-word ALUs and their

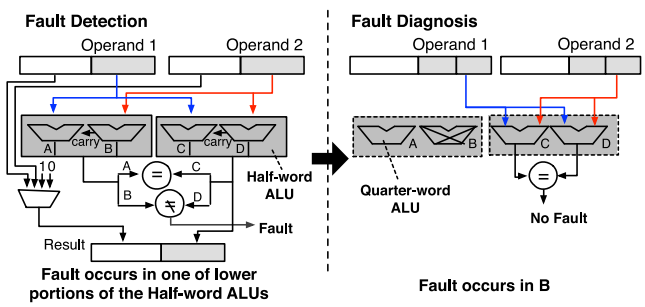


Fig. 3. Fault detection and diagnosis at a half-word duplication granularity (FD-H).

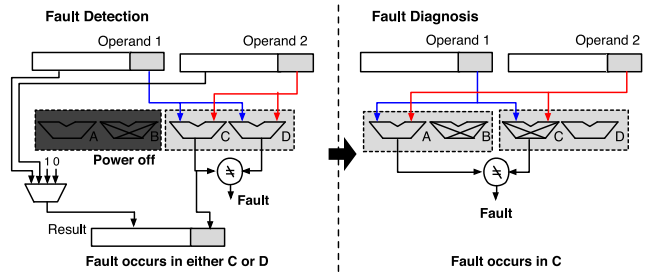


Fig. 4. Fault detection and diagnosis at a quarter-word duplication granularity (FD-Q).

outputs are compared to detect a fault. At the same time, the upper half-word of the instruction's result is predicted to be all ones, all zeros, or that of its operands. The predicted value is concatenated with one of outputs of the half-word ALUs to form a full-word result. If an instruction is a NP-inst, the half-word ALUs execute the instruction with lower half-word of operands at first and then with upper half-word of operands later; thus, it takes two cycles to execute a NP-inst.

For fault diagnosis, we utilize the existing redundancy in ALU. Due to symmetric structure of the ALU, its different portions can generate the same output values as long as the same input values are provided to them. When a fault is detected during executing an instruction, each half-word ALU is partitioned into two copies of quarter-word ALUs, and then the instruction is re-executed on fault-free quarter-word ALUs with the same input values. By comparing the output of the quarter-word ALUs, we can identify the faulty one.

Fig. 3 depicts an example scenario of fault detection and diagnosis when an instruction is P-inst. The two half-word ALUs perform a computation with the lower half-word of operands. In this scenario, lower portion of their outputs are not identical (a fault is detected). To perform fault diagnosis, each half-word ALU is partitioned into two quarter-word ALUs; thus, there are total four quarter-word ALUs (A, B, C, and D). Since a fault is detected in the outputs of quarter-word ALU B and D, a quarter-word ALU C, which is fault-free, re-executes the instruction with lower quarter-word of its operands. After comparing the outputs of quarter-word ALU C and D, we can conclude that the fault occurred in the quarter-word ALU B because their outputs are identical.

In the presence of a faulty half-word ALU, the concurrent fault detection cannot be performed for MP-inst. However, for HP-inst, it can be performed because there are still three quarter-word ALUs which can execute a HP-inst in a single cycle. Thus, we consider two variants of the fault detection and diagnosis techniques according to the duplication granularity indicating how many bits of operands are duplicated: fault detection and diagnosis at a half-word duplication granularity (FD-H), which is explained above, and at a quarter-word duplication granularity (FD-Q).

Fig. 4 illustrates an example scenario of FD-Q when an instruction belongs to HP-inst. Each of half-word ALUs operates as two separate quarter-word ALUs (A and B, C and D). In order to check whether a fault occurred, quarter-word ALU C and D perform a computation with the lower

quarter-word of operands. Inconsistency between outputs of the ALUs indicates that a fault occurred in one of these quarter-word ALUs. For fault diagnosis, quarter-word ALU A, which is fault-free, is utilized to perform the same computation. Since the outputs of quarter-word ALU A and C are not identical, we can conclude that the fault occurred in the quarter-word ALU C.

To perform the concurrent fault detection, FD-H requires a single cycle for a P-inst (either MP- or HP-inst) and double cycles for a NP-inst while FD-Q requires a single cycle for a HP-inst, double cycles for a MP-inst and quadruple cycles for a NP-inst. As shown in Fig. 2, more than 92 percent of ALU instructions belong to P-inst on average. Among them, more than 74 percent of ALU instructions belong to HP-inst on average. Thus, we can expect that FD-H will incur much low performance degradation and the performance impact of FD-Q will not be noticeable.

We note that the proposed fault detection technique can detect soft errors as well as hard faults. To identify whether the detected fault is transient (i.e., soft error) or permanent, the fault detection needs to be re-performed with a time interval. If the fault is identified as transient, the ALU continues its operation without performing fault diagnosis.

3.2 Fault Isolation

Our fault detection and diagnosis technique can identify a faulty quarter-word ALU in a half-word ALU. By disconnecting all paths from the lower quarter-word ALU to the upper one so that errors cannot be propagated through them, the fault-free quarter-word ALU can be used while isolating the faulty one. Thus, even if a half-word ALU is faulty, it can continue its operation as long as one of the quarter-word ALUs is functional. Thanks to this fine-grained fault isolation capability, our proposed fault-tolerant mechanism significantly improves the reliability of ALU, which will be discussed in more detail in Section 6.2.

After isolating the faulty quarter-word ALU of a half-word ALU, fault detection is performed using another fault-free half-word ALU partitioned into two quarter-word ALUs. At that time, the faulty half-word ALU is deactivated to save energy consumption. In the presence of another fault, the deactivated half-word ALU is turned on back and its fault-free quarter-word ALU is used for fault diagnosis. After another faulty quarter-word ALU is isolated, the remaining two quarter-word ALUs are used to execute instructions while detecting faults concurrently.

4 Implementation Details

In this section, we show a detailed implementation of a 32-bit ALU with LIZARD, called LIZARD ALU. Fig. 5 shows its organization which is composed of seven functional modules. The Operand Regenerator transforms operands into duplicated operands. By this transformation, sub-ALUs are selected to perform a computation and fault detection. The duplicated operands are consumed by two copies of 16-bit ***partitioned ALU*** which can be partitioned into two 8-bit ALUs. In parallel, a Narrow-width Value Detector checks the narrowness of operands. After an

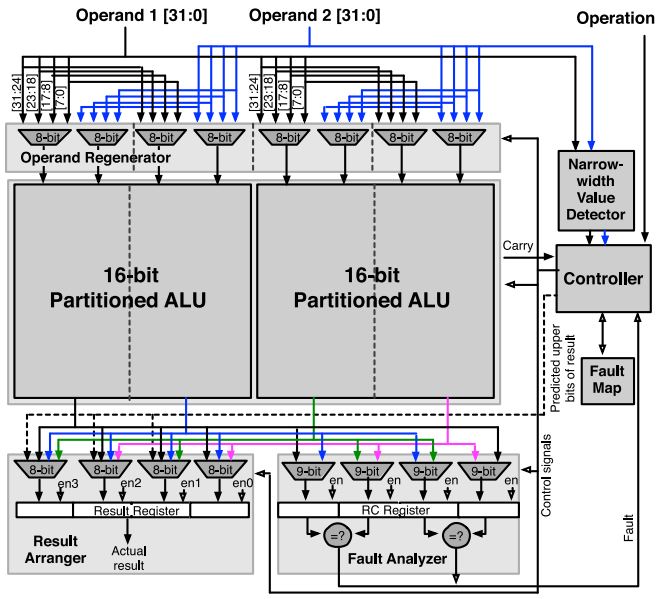


Fig. 5. Detailed organization of 32-bit LIZARD ALU.

operation is executed, the Result Arranger reconstructs an actual result with outputs of the ALU and the predicted upper bits of the result. Meanwhile, the Fault Analyzer verifies whether a fault occurred or not. These functional modules are managed by a Controller. Especially, it predicts the upper bits of the results by observing the operation type of instructions, the narrowness of their operands, and carry signals generated by the ALUs. Controller also determines the execution latency of the instructions by checking the predictability of their results and referring to the fault locations stored in a Fault Map. In the following sub-sections, we will present the detailed design of these functional modules.

4.1 Partitioned ALU

LIZARD requires two copies of half-word ***partitioned ALU*** which can be partitioned into two separate sub-ALUs (quarter-word ALUs). The logical units, such as AND, OR, XOR units, are naturally partitioned. Thus, the challenge is to implement partitioned adder on top of the conventional adders, such as Carry Look-ahead Adder (CLA), Kogge-Stone Adder (KSA), Brent-Kung Adder (BKA), and Sklansky Adder (SKA), etc. In this section, we give a general method to implement the partitioned adder.

4.1.1 Binary Addition as Form of Prefix Computation

The prefix computation is to compute N outputs from N inputs using an associative binary operation ($Y_1 = X_1$ and $Y_n = Y_{n-1} \circ X_n$, where $1 < n \leq N$). Binary addition is one of the most fundamental prefix computations in computer arithmetic. Most prefix computation consists of three phase: pre-computation, prefix network, and post-computation. Fig. 6 illustrates the binary addition as a form of the prefix computation. In pre-computation phase, two inputs (A and B) are used to produce initial generate (G_i) and propagate (P_i) signals. And then they are used to produce intermediate generate ($G_{i:j}$) and propagate ($P_{i:j}$) signals, consequently carry signals ($C_i = G_{i:0}$), in prefix network phase. Finally, in

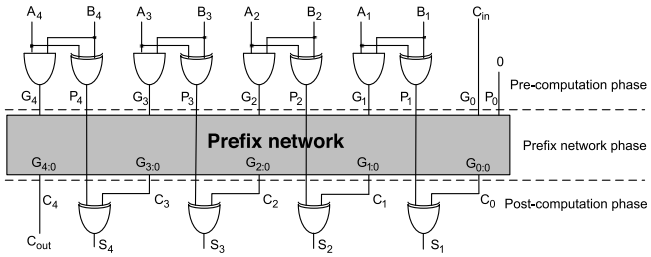


Fig. 6. 4-bit binary addition.

post-computation phase, final results (S_i) are produced using the initial propagate signals and the carry signals. Following equations represent the computations performed in each phase.

Pre-computation phase:

$$G_i = A_i \cdot B_i, \qquad 1 \le i \le N \tag{1}$$

$$P_i = A_i \oplus B_i, \qquad 1 \le i \le N \tag{2}$$

$$(G_0, P_0) = (C_{in}, 0). \tag{3}$$

Prefix network phase:

$$(G_{i:i}, P_{i:i}) = (G_i, P_i), \qquad 0 \le i \le N \tag{4}$$

$$\begin{aligned}(G_{i:j}, P_{i:j}) &= (G_{i:k}, P_{i:k}) \circ (G_{k-1:j}, P_{k-1,j}) \\ &= (G_{i:k} + P_{i:k} \cdot G_{k-1:j}, P_{i:k} \cdot P_{k-1:j}), \\ &\qquad 0 \le j < k \le i \le N\end{aligned} \tag{5}$$

$$C_i = G_{i:0}, \qquad 0 \le i \le N. \tag{6}$$

Post-computation phase:

$$S_i = P_i \oplus C_{i-1}, \qquad 1 \le i \le N. \tag{7}$$

4.1.2 Partitioning the Conventional Adder

As shown in the Fig. 6, the pre-computation and the post-computation phases are naturally partitioned, which means that there are no dependency between each bit position (e.g., G_4 and P_4 are produced using only A_4 and B_4). On the other hand, in the prefix network phase, results of the lower bit positions are used to produce those of the higher bit positions (e.g., $G_{2:0}$ is used to produce $G_{3:0}$). Thus, we need to partition the prefix network to implement the partitioned adder.

Fig. 7 (left) shows the prefix networks of the four 8-bit conventional adder designs (N=8). The numbers shown in upper box of each prefix network indicate the initial generate and propagate signals (G_i and P_i) while the numbers shown in lower box indicate the computed carry signals ($G_{i:0}$). The logic-level implementations of basic cells of the prefix network are shown in Fig. 8. A black

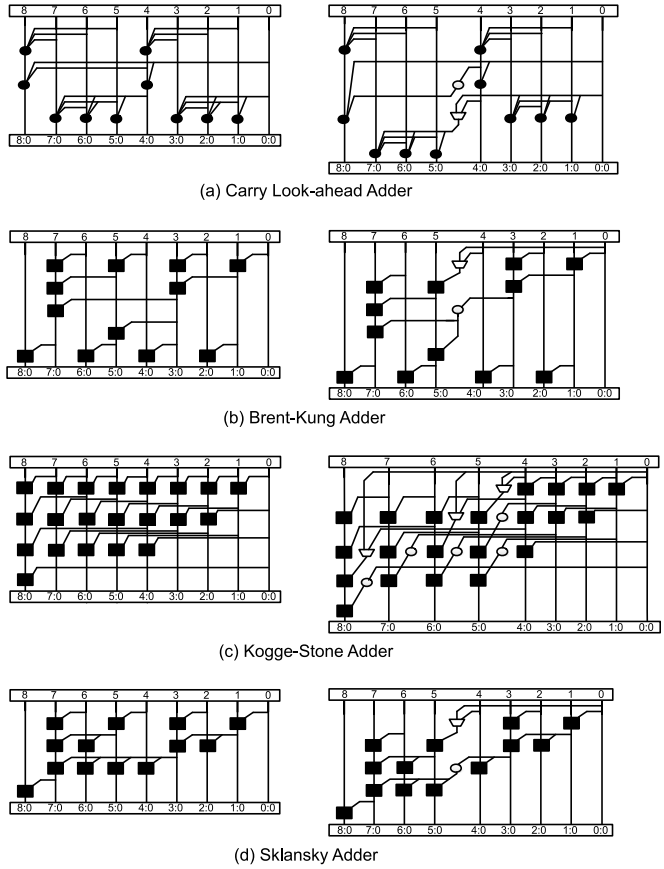


Fig. 7. 8-bit prefix network (left: conventional prefix network, right: partitioned prefix network).

square cell represents the prefix computation ∘ producing intermediate generate and propagate signals. A black circle cell is a higher valency design of the black square cell, which also represents the prefix computation, for the CLA.

As shown in Fig. 7 (left), if we vertically divide the prefix network into two parts (left half and right half), the two parts are almost identical. Thus, the prefix network can be easily partitioned through removing the dependency between the left and right parts. To achieve this, we employ white circle cells and multiplexers as shown in Fig. 7 (right). The white circle cell, illustrated in the Fig. 8, removes the dependency between the left and right parts of the prefix networks through masking the signals generated by the right part; If the value of partitioning signal of a white circle cell is one, the value of G_i' and P_i' become zero and one, respectively, regardless of the value of G_i and P_i. Then, black square cells, which consume outputs of the white

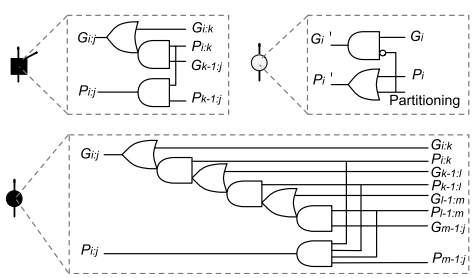


Fig. 8. Logic-level implementation of basic cells of the prefix network.

circle cell, bypass the intermediate generate and propagate signals generated by their upper-row black square (or circle) cell as follows:

$$\begin{aligned}(G_{i:j}, P_{i:j}) &= (G_{i:k}, P_{i:k}) \circ (G_{k-1:j}{}', P_{k-1,j}{}') \\ &= (G_{i:k}, P_{i:k}) \circ (0, 1) \\ &= (G_{i:k} + P_{i:k} \cdot 0, P_{i:k} \cdot 1) \\ &= (G_{i:k}, P_{i:k}).\end{aligned} \tag{8}$$

The multiplexer selects between (G_0, P_0) and $(G_{i:j}, P_{i:j})$, and then forwards the selected signals to the right-most generate and propagate signals of the left part. With the white circle cell and multiplexer, the prefix network of each adder design is completely partitioned into two separate half-size prefix networks.

The additional cost to partition the prefix network is negligible for most of the adder designs; it remains constant irrespective of the bit-width except the KSA. For n-bit CLA, BKA, and SKA, a single multiplexer and a single white circle cell are required to partition their prefix networks as shown in Fig. 7a, 7b, and 7d (right). On the other hand, partitioning the prefix network of the KSA requires more additional costs due to its heavy wiring congestion; $n - log_2 n$ of the white circle cells and $log_2 n$ of the multiplexers are required as shown in Fig. 7c (right). In Section 6.1, the additional costs are discussed in more detail.

4.2 Operand Regenerator/Result Arranger

Operand Regenerator, which is located before the partitioned ALU, performs operand duplication. It consists of eight 4:1 8-bit MUXs. The MUXs divide each operand into four parts and send them to the fault-free portions of the partitioned ALUs, which are selected by Controller.

Result Arranger reconstructs the actual result. This unit consists of three 4:1 8-bit MUXs, a single 3:1 8-bit MUX, and a result register. In case of P-inst, the MUXs select the upper bits of results among predicted values (i.e., all zeros, all ones, or upper bits of operands). For the lower bits, the MUXs select outputs of the partitioned ALUs. In case of NP-insts, the four parts of their results are filled with the outputs of the partitioned ALUs across consecutive cycles (two cycles for FD-H and four cycles for FD-Q); therefore, the result register consists of four 8-bit registers having individual enable signals (en0 $\sim$ en3). In the processor pipeline, we can utilize an existing pipeline register as the result register.

4.3 Fault Analyzer

Fault Analyzer checks whether a fault occurred or not in the partitioned ALU and identifies their faulty portions. In case of adders, the carry signals as well as results need to be verified. Hence, 9-bit MUXs and comparators are employed. The MUXs choose inputs of comparators between outputs (carry signals and results) of the partitioned ALUs depending on their configurations. The comparators are used to detect and diagnose faults by comparing the given inputs.

In order to minimize the critical path delay, Fault Analyzer need to be out of the critical path, because the latency

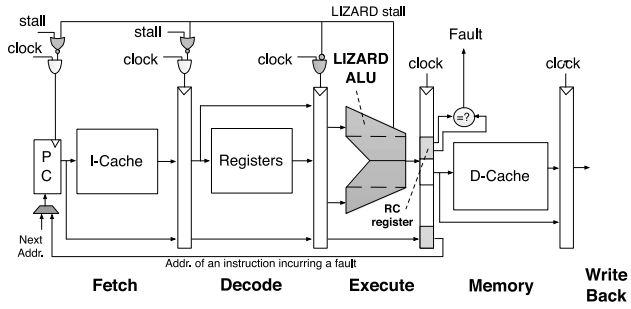


Fig. 9. The processor pipeline of a scalar processor using LIZARD ALU.

of Fault Analyzer is relatively long due to comparators. For this reason, we incorporate another register, called RC (Result and Carry) register, to store the outputs of the partitioned ALUs. Fault Analysis is performed in background using the stored value in the RC register when a successor instruction is executed in the partitioned ALUs.

5 Processor Architecture Design Using LIZARD ALU

In this section, we present issues in the processor architecture design when employing LIZARD ALU. First we will present architectural modifications in scalar processors and then in superscalar processors.

5.1 Scalar Processor

The scalar processors use a single ALU and its pipeline structure is simple. Thus, minor modifications are required to employ LIZARD ALU instead of the conventional ALU. Fig. 9 illustrates the overall processor pipeline of a scalar processor using a LIZARD ALU. It consists of five-stage pipeline: fetch, decode, execute, memory, and write back. In execute stage, LIZARD ALU produces computation results of the ALU instructions by performing computations while predicting the upper bits of the results. At this time, the instruction types (HP-inst, MP-inst, or NP-inst) of the instructions are known and some of them require multiple cycles to be executed as presented in Section 3.1. In this case, it is necessary to stall the processor pipeline for the multi-cycle executions. To achieve this, a few logic gates (gray-colored) are added in the pipeline stall logic which is inherently used to address the pipeline hazards. Fault detection is performed in the memory stage in order to minimize the increase in the critical path delay of the execute stage as we explained in Section 4.3. Thus, RC register of the LIZARD ALU is located between the execute and memory stages. When a fault is detected, an instruction incurring the fault is re-fetched from the instruction cache and re-executed on the ALU in fault diagnosis mode. This is achieved by forwarding an address of the instruction in the memory stage to the program counter (PC) while discarding instructions in all pipeline stages except the write back stage. To implement this process, the instruction address, which is originally used to calculate the branch target address in the execute stage, is stored in an additional register located between the execute and memory stages. And an instruction flush mechanism, inherently employed for recovering the branch miss-prediction, are utilized.

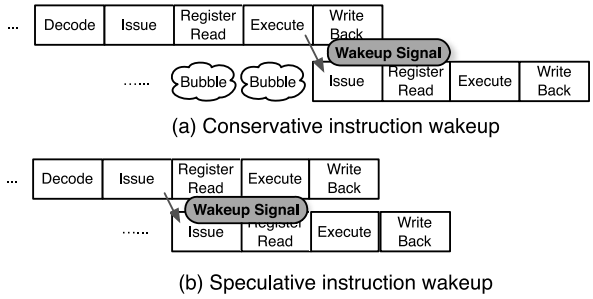


Fig. 10. Examples of dependent instruction wakeup.

5.2 Superscalar Processor

The out-of-order superscalar processors employ multiple ALUs so that multiple instructions can be executed per cycle. To maximize the computing throughput, instructions are executed in an order that different from their original order in a program. For these processors, directly employing the LIZARD ALU might harm their performance due to its variable execution latency aforementioned in Section 3.1. We introduce architectural techniques for minimizing the performance impact of the LIZARD ALUs in the superscalar processors even when some of the ALUs are faulty.

5.2.1 Speculative Instruction Wakeup

The instruction issue logic is a key component that allows the out-of-order execution of instructions by issuing them to the functional units as soon as their source operands are available. There are many implementation methods of the issue logic. In our baseline superscalar processor, a unified issue queue is used for the issue logic. It stores the information of all instructions waiting to be issued. These instructions become ready to be executed by "wakeup" process. When an instruction finishes its execution, the issue logic broadcasts "wakeup signal" to each entry of the issue queue for notifying that one of the source operands of its dependent instructions has been produced. Through this process, the instructions get to know whether their source operands are available.

Fig. 10 shows example scenarios of the wakeup process. There are two common ways for implementing the wakeup process. First one is a conservative wakeup mechanism which generates the wakeup signal after an operand value is available. In this mechanism, instructions should wait until their operands are available, which incurs bubble cycles. Another way is a speculative wakeup mechanism which generates the wake-up signal before an operand value is actually available. Since an instruction does not need its source operands until it reaches the execute stage, the wakeup signal can be generated in advance. After a source operand is generated, it is forwarded to the input of a functional unit through the bypass network. Most high performance processors use the speculative wakeup mechanism to enhance their performance [14], [15].

In order to maximize the performance gain of the speculative wakeup mechanism, the execution latencies of instructions need to be constant. If they are variable and known only when the instructions are executed, we need to predict their execution latencies to wakeup their dependent instructions speculatively. When a latency prediction fails (i.e., miss-speculative instruction wakeup), the dependent instructions, which were speculatively issued, have to be flushed from the pipeline and rescheduled to issue again [16]. This process is called instruction replay.

5.2.2 Latency Prediction and Instruction Replay for Speculative Instruction Wakeup

The speculative wakeup mechanism is always beneficial for the ALU instructions because their execution latencies are constant. However, when we employ the LIZARD ALU, the execution latencies of the ALU instructions become variable as we presented in Section 3.1. In addition, the latencies are known only after the execute stage. Therefore, we need to predict the execution latencies of the ALU instructions when they are dispatched into the issue queue.

To this end, we introduce a P-inst predictor and extend the issue logic (issue queue and arbiter). Fig. 11 (left) shows the pipeline of the superscalar processor employing three LIZARD ALUs. P-inst predictor, which is located in the decode stage, predicts the instruction type of the ALU instructions using historical information and the predicted instruction type is stored in an additional field (P) of a corresponding entry of the issue queue. When an arbiter (also called "scheduler") selects one of ready instructions and allocates an ALU, it determines the execution latency of the selected instruction by referring to both the P field and the status (either faulty or fault-free) of the ALU. For example, if the P field of the selected instruction indicates that the corresponding instruction is predicted as NP-inst, its execution latency is determined to be two cycles when a fault-free ALU is allocated. When a faulty ALU is allocated, the execution latency is determined to be four cycles.

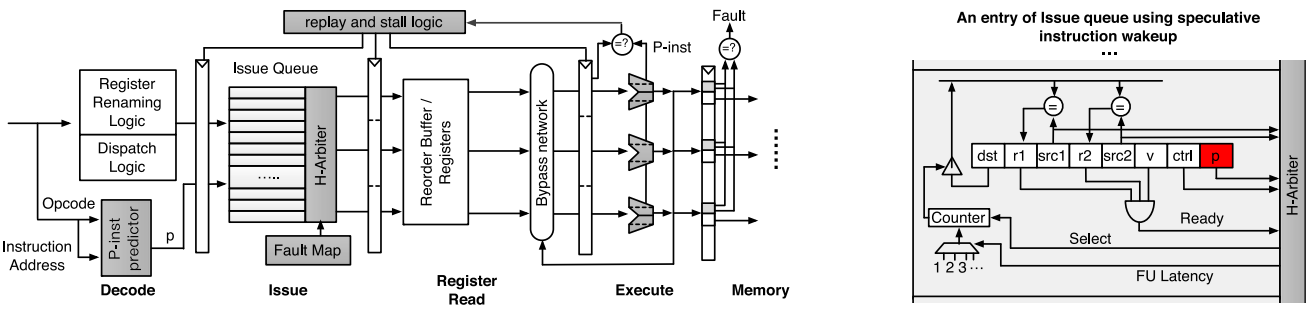


Fig. 11. The processor pipeline of a superscalar processor using LIZARD ALU

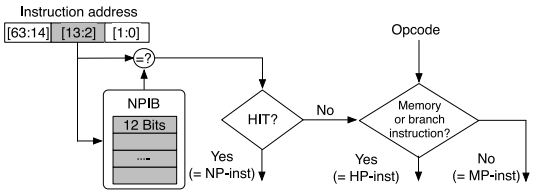


Fig. 12. P-inst predictor.

The P-inst predictor is implemented with two level prediction scheme as shown in Fig. 12. At the first level, P-inst predictor filters out the NP-insts using a simple buffer called non-predictable instruction buffer (NPIB). NPIB stores the instruction addresses of the recently executed NP-insts. When an instruction is dispatched, its address is compared with all entries of NPIB. If the instruction hits on NPIB, it is predicted as an NP-inst. If the instruction misses on NPIB, it is predicted as a P-inst. Actual instruction type is known after the instruction is executed in the execute stage. If the actual instruction type is NP-Inst, its address is stored in NPIB. In our implementation, we store small portion (12 bits) of the instruction addresses (64 bits) to minimize the implementation cost. Since the NP-insts account for less than 6 percent and small instruction sets (i.e., basic blocks) are repeatedly executed, NPIB with a few entries can efficiently filter out the NP-insts. At the second level, the P-inst predictor distinguishes the HP-insts from the MP-insts with a simple heuristics. In our experiments, we observed that most memory and branch instructions belong to the HP-inst (more than 97 percent on average). This is because these instructions use the ALU to obtain the effective address, which is calculated by adding the base addresses and small offset values in most cases. Moreover, memory and branch instructions account for more than 60 percent, on average, of total ALU instructions. By exploiting this observation, we predict the memory and branch instructions as HP-insts.

Fig. 11 (right) illustrates one entry of a typical issue queue with one issue port [16], which includes the P field. In order to support long latency instructions such as load, store, and multiply, broadcasting the wakeup signal need to be delayed until the instructions' results are produced. This is typically implemented with a small counter. When a ready instruction is selected to be issued, its counter value is set to its execution latency determined by the arbiter. After the counter expires, the wakeup signal is broadcasted to the dependent instructions. In the issue queue, required modification to support variable execution latency of ALU instructions is only including the P fields (two bits per each entry of the issue queue), which is negligible.

In the execute stage, the actual instruction type is known and it is compared to the predicted one. On a mismatch between them, the instruction replay process is performed in some cases. In addition, for MP- and NP-insts, a path of the processor pipeline need to be stalled when they require multiple cycles in the execute stage. Fig. 13 illustrates the decision diagram involved in a replay and stall logic. In order to determine actions of the logic, three factors are considered: actual instruction type, predicted instruction type and status of the allocated ALU. The instruction replay process, which is more crucial for performance, is performed

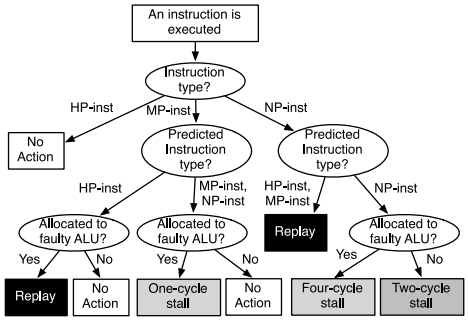


Fig. 13. Steps involved in replay and stall logic.

for only two cases. First case is when a MP-inst is predicted as HP-Inst and allocated to a faulty ALU. If a MP-inst is predicted as HP-Inst, the counter value of its issue queue entry is set to one and thus the wakeup signal is broadcasted to its dependent instructions in the following cycle. However, in the case when the MP-inst is executed with a faulty ALU, it is executed in two cycles and thus it cannot provide its result value to the dependent instructions on time. Therefore, we need to replay the dependent instructions. Second case is when an NP-inst is predicted as P-Inst (either HP- or MP-inst). When an instruction belongs to the NP-Inst, it is executed in multiple cycles irrespective of the status of the allocated ALU. Thus, if an NP-inst is predicted as P-Inst, it also cannot provide its result value on time.

5.2.3 Heterogeneity-Aware ALU Allocation

In our superscalar processor, the ALUs can be heterogeneous in the presence of faults because LIZARD ALU can isolate its faulty portions and continue its operation. In order to maximize the performance even when faults occur on some of LIZARD ALUs, we leverage the heterogeneity when allocating the ALUs to the instructions. In the issue stage, the arbiter chooses the subset of ready ALU instructions whose operands are available and allocates an ALU to each of them. In this process, by considering the status of the ALUs, we adaptively allocate them to the instructions. To this end, we extend the conventional arbiter to heterogeneity-aware arbiter (H-Arbiter). H-Arbiter obtains the fault information of each ALU by observing the fault map. If an instruction is HP-Inst, it is allocated to a faulty ALU whenever it is available. This policy gives more chance for MP-Insts or NP-Insts to be executed on a fault-free ALU.

6 Evaluation

We evaluate LIZARD and competitive fault-tolerant mechanisms (Baseline, TMR and DMR_S). Baseline indicates no fault-tolerance. DMR_S (Dual Modular Redundancy with Spare) uses two ALUs to detect a fault and employs a spare ALU to replace a faulty ALU. The two ALUs of DMR_S execute the same operation and detect faults by comparing their outputs while the spare ALU is deactivated. When a fault occurs in one of the two ALUs, the spare ALU is used to identify a faulty one. After that, a reconfiguration unit replaces the faulty ALU with the spare ALU.

To estimate the hardware costs (area, power and critical path delay) of the fault-tolerant mechanisms, we implemented them in verilog HDL and then synthesized in

TABLE 2
Configurations of Simulated Systems

Parameter	Configuration	
	Superscalar Processor	Scalar Processor
Clock frequency	1.88 ∼ 2 GHz	760 ∼ 800 MHz
Fetch/issue/commit	4/4/4	1/1/1
Window size	Issue queue (32), ROB (96), LSQ (32)	-
Function units	Int Add /Mult (3/1), FP Add/Mult (3/1)	Int Add/Mult (1/1), FP Add/Mult (1/1)
L1 I-cache&D-cache	32KB, 4-way, 64B line, 3 cycles	32KB, 4-way, 32B line, 1 cycle
Unified L2 cache	2MB, 8-way, 64B line, 12 cycles	256KB, 8-way, 32B line, 6 cycles
BTB/Branch Predictor	2048-entry 4-way BTB / Comb. of bimodal and 2-level global	128-entry direct-mapped BTB / Bimodal
Memory access latency	141 ∼ 150 cycles	76 ∼ 80 cycles
ITLB&DTLB	32-entry, 4-way, 4KB page size	

** Clock frequency and memory access latency vary depending on the fault tolerant mechanism employed in ALU.*

Synopsys Design Compiler using TSMC 45 nm standard cell library. Since we assume that scalar processors employ a 32-bit ALU and superscalar processors employ 64-bit ALUs, we estimated the hardware costs for both of them. Performance and energy evaluations were performed using the XTREM simulator [17] which models the scalar processor and the MASE simulator [18] which models the superscalar processor. For energy evaluation, we integrated McPAT [19], which estimates the energy consumption of processors, into these simulators. Architectural configurations of the simulated systems are shown in Table 2. For typical applications of scalar and superscalar processors, we used 16 benchmarks of MiBench [13] and 22 benchmarks of SPEC CPU2006, respectively, which are executable in our simulators. These sets evenly include integer, floating point, memory-intensive and cpu-intensive benchmarks.

6.1 Hardware Cost Estimation

Fig. 14 shows the hardware costs (area, power and critical path delay) of the fault-tolerant mechanisms, which are normalized to that of baseline. As shown in Fig. 14a, the area overhead of LIZARD is 65 and 63 percent of the baseline in case of 32-bit and 64-bit ALUs, respectively, on average (AVG). This overhead is much lower than those of TMR and DMR_S (on average, more than 208 and 223 percent, respectively). We note that the area overhead of LIZARD is smaller in case of 64-bit ALU than 32-bit ALU. This is because the additional costs for partitioning the conventional adders remain constant irrespective of the bit-width except the KSA as we discussed in Section 4.1. Even if the KSA requires more additional costs to be partitioned, they are also not noticeable because area of the KSA is much larger than the additional circuits. Since ALU is a small component, it takes up relatively small area in a core (5.6 and 1.0 percent in scalar and superscalar processors, respectively, which are estimated by McPAT [19]). Thus, the overhead of LIZARD is trivial in terms of total core area (less than 3.5 and 0.6 percent in the scalar and superscalar processors, respectively). Both TMR and DMR_S also require small area overheads in the superscalar processor (3.02 and 2.4 percent, respectively). However, their area overheads are not ignorable in the scalar processor (15.7 and 12.5 percent, respectively). In order to estimate the area overhead of P-inst predictor, we use a CAM structure model of McPAT [19]. Since this component consists of a small memory array (up to sixteen 12-bit entries) and a few logic gates, the area overhead is trivial (up to 0.06 percent in the superscalar processor).

LIZARD slightly increases the critical path delay as shown in Fig. 14b. This is because it employs two copies of the half-word partitioned ALU which is faster than the full-word ALU. LIZARD increases the critical path delay by only 1.8 and 1.2 percent on average for 32-bit and 64-bit ALUs, respectively, even if the additional circuits such as Operand Regenerator and Result Arranger are on the critical path. On the other hand, DMR_S increases the critical path delay by more than 5.1 and 4.3 percent for 32-bit and 64-bit ALUs, respectively, due to the reconfiguration unit; we assume that DMR_S performs fault detection in the successor pipeline stage (e.g., memory stage in the scalar

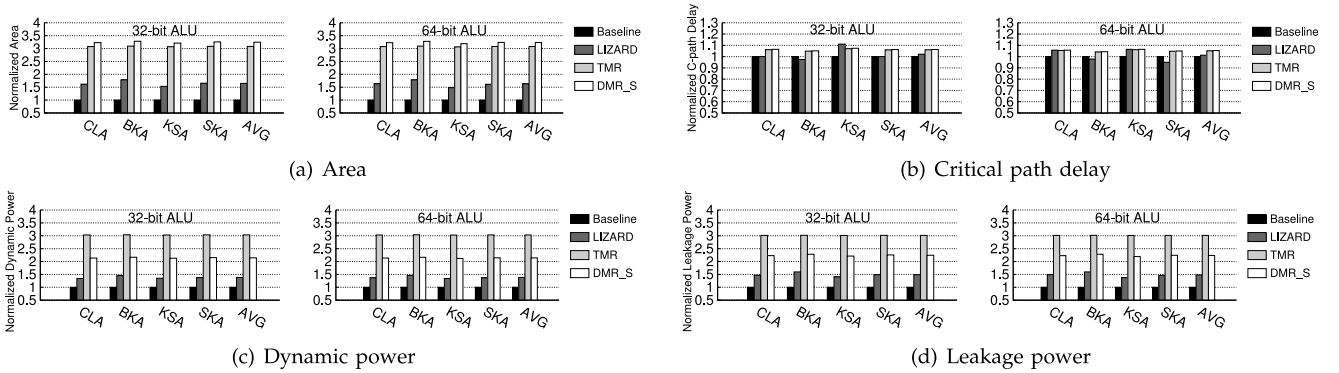


Fig. 14. Area, power, and critical path delay (*x*-axis indicates the ALUs using each adder design).

processor) as LIZARD. TMR increases the critical path delay by more than 4.8 and 4.1 percent for 32-bit and 64-bit ALUs, respectively, due to its majority voter. These impacts on the critical path delay can be minimized when optimizing the proposed mechanisms in the processor pipeline. For example, in case of LIZARD, Operand Regenerator and Result Arranger can be integrated into an input and output select MUXs, respectively, in the execution pipeline stage. DMR_S also can reduce its critical path delay by integrating its reconfigurable units into the output select MUXs. However, we do not consider this optimization for fair comparison, because the effectiveness of the optimizations can vary depending on the pipeline datapath design.

Fig. 14c and 14d show the dynamic and leakage powers. For 32-bit ALUs, LIZARD increases the dynamic and leakage powers by about 38 and 49 percent, respectively, on average. On the other hand, TMR dramatically increases the dynamic and leakage powers (by about 203 and 201 percent, respectively, on average) due to its redundant ALUs. Even if DMR_S uses two ALUs for fault detection while deactivating the spare one, it also suffers from significant dynamic and leakage power overheads (about 114 and 124 percent, respectively, on average). The power overheads of LIZARD for 64-bit ALU are similar to those for 32-bit ALU. We use these power overheads to estimate the impact of the fault tolerant mechanisms on the energy consumption of the processors in Section 6.4.

6.2 Reliability

Compared with the competitive mechanisms, LIZARD provides better fault tolerance capability [20], which indicates how many faults can be tolerated, with low cost. As we presented in Section 3, LIZARD can generate correct results as long as two of the four quarter-word ALUs are functional. Thus, it can tolerate up to two faults which occur in different quarter-word ALUs at the same time or at a distance of time, whereas TMR and DMR_S can tolerate a single fault.

To quantify the benefit of the outstanding faulty tolerance capability of LIZARD, we analyze the reliability of LIZARD ALU and its competitors. The reliability of a component is conventionally defined as the probability that it will survive at least until time t, denoted by R(t), which can be calculated as follows [21], [22]:

$$R(t) = Prob\{T > t\} = e^{-\lambda t^{\beta}}, \tag{9}$$

where T and λ denote the lifetime and the initial failure rate (t=0) of a component, respectively. Depending on the type of fault model, the aspect of the failure rate is different. For example, defect-induced faults typically occur in the early stage of the lifetime (i.e., infant mortality failures). Thus, their failure rate decrease as time goes on. On the other hand, the failure rate of wearout-induced faults increase. In above equation, β is used for reflecting these aspects of the failure rates. For the former case, β is less than 1 while it is greater than 1 for latter case. The initial failure rate is computed with the following empirical failure rate formula [21].

$$\lambda = (c_1\pi_T\pi_V + c_2\pi_E)\pi_Q\pi_{L,} \tag{10}$$

where C_1, C_2 are complexity factors, $\pi_T, \pi_V, \pi_E, \pi_Q$, and π_L are temperature, voltage stress, environment, quality, and learning factor, respectively. For simplicity, we conservatively assume that the failure rate depends on only complexity factors which are functions of the number of gates of the circuits.

Using this reliability formula, we derive reliability models of the fault-tolerant mechanisms. Their reliabilities are calculated with the reliabilities of their components. For instance, the reliability of 32-bit LIZARD ALU is expressed as the following formula, denoted by $R_{LIZARD}(t)$:

$$R_{LIZARD}(t) = R_{RA}(t)R_{FA}(t)R_N(t)R_C(t)\sum_{i=2}^{4}\binom{4}{i} (R_{ALU8}(t)R_{OR8}(t))^i \times (1 - R_{ALU8}(t)R_{OR8}(t))^{4-i}, \tag{11}$$

where R_{RA}, R_{FA}, R_C, $R_N(t)$, R_{ALU8}, R_{ALU8} and R_{OR8} are the reliabilities of Result Arranger, Fault Analyzer, Narrow-width Value Detector, Controller, 8-bit ALU and 8-bit Operand Regenerator, respectively. This formula implies that LIZARD ALU can continue its operation as long as two of its four quarter-word ALUs are functional because the quarter-word ALUs can operate independently by disconnecting paths between them as presented in Section 3.2. The reliabilities of 32-bit TMR and DMR_S ALUs are expressed as follow:

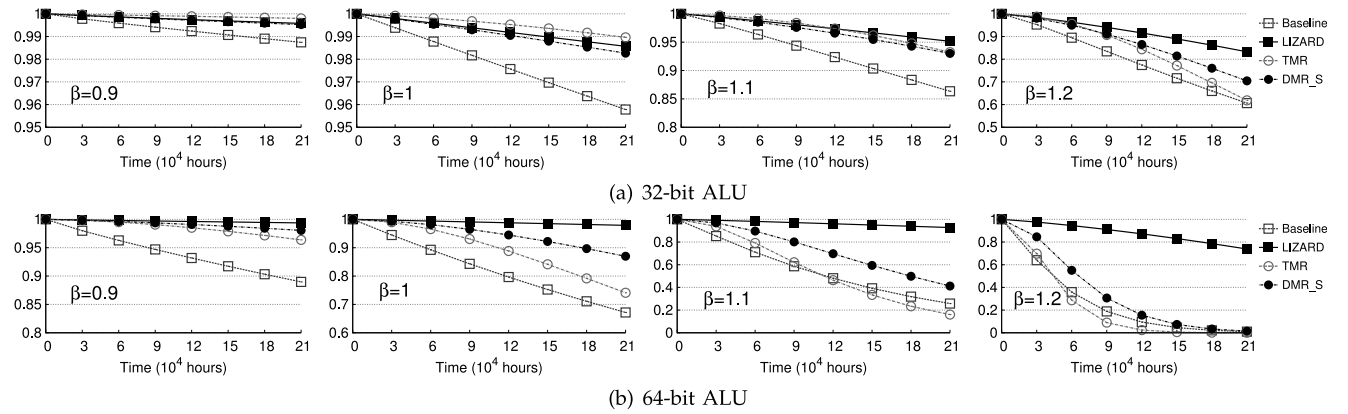
$$R_{TMR}(t) = R_{voter}\sum_{i=2}^{3}\binom{3}{i}(R_{ALU32}(t))^i(R_{ALU32}(t))^{3-i} \tag{12}$$

$$R_{DMR\_S}(t) = R_{comp}R_{RU}\sum_{i=1}^{2}\binom{2}{i}(R_{ALU32}(t))^i(R_{ALU32}(t))^{2-i}, \tag{13}$$

where R_{voter}, R_{ALU32}, R_{comp} and R_{RU} are the reliabilities of voter, 32-bit ALU, comparator and reconfiguration unit.

By using these reliability models, we plot the reliabilities of the 32-bit and 64-bit fault-tolerant ALUs. In order to show their reliabilities over various aspects of the failure rate, we use different values of β. In all cases, LIZARD achieves higher reliability than both TMR and DMR_S across all lifetimes as shown in Fig. 15. When the failure rate is constant or decrease ($\beta \leq 1$), the reliabilities of all 32-bit fault-tolerant ALUs are similar. However, TMR and DMR_S ALUs are less reliable than LIZARD ALU for $\beta = 1.2$. Moreover, in case of 64-bit TMR and DMR_S ALUs, their reliabilities are significantly reduced as time goes on for $\beta \geq 1.1$. This is because the failure rate of each component in the ALUs is pretty high due to their large area. Interestingly, the reliability of TMR ALU drops even below that of the baseline ALU when the failure rates become high. This is because redundancy becomes a disadvantage if the reliability of a single component of TMR is less than 0.5 [21].

Even if ALU is a small component in a processor core, it is very susceptible to hard faults, especially for the scalar processors. This is because a single ALU is employed in these processors and the ALU is one of the most frequently used components. Thus, a single hard fault in the ALU



(a) 32-bit ALU

(b) 64-bit ALU

Fig. 15. Reliability trends over time.

could make the entire processors obsolete. In case of other large components such as cache memory, the hard fault can be tolerated by disabling faulty cache blocks because there are a lot of identical blocks. Even though the superscalar processors can continue its operation by disabling one of their ALUs, they may suffer from significant performance degradation as will be shown in Section 6.3.2.

6.3 Performance Overhead

In performance evaluation, we assume that the clock cycle time is determined by the latency of the ALU. Thus, we increase the clock cycle time by the average of critical path delay overheads of LIZARD, TMR and DMR_S (1.8, 6 and 6.3 percent, respectively for scalar processors and 1.2, 5 and 5.3 percent, respectively, for superscalar processors).

6.3.1 Scalar Processor

Fig. 16 shows the impact of the fault-tolerant ALUs on execution times which are normalized to the baseline. With TMR and DMR_S ALUs, the execution times increase by average 5.8 and 6.7 percent, respectively. The performance overhead of LIZARD ALU ranges from 2 to 20.1 percent, with average 4.1 percent. The overhead depends on the percentage of P-insts. On one hand, in case of the sha benchmark, 26 percent of ALU instructions are NP-insts as shown in Fig. 2. Since LIZARD ALU executes these instructions across two cycles, it suffers from large execution time overhead. On the other hand, for the adpcm benchmark, LIZARD ALU incurs lowest execution time overhead because almost instructions belong to P-insts (about 99 percent) in this benchmark.

After a faulty portion is isolated, LIZARD ALU incurs 22 percent performance loss, on average, as shown in Fig. 16. This is because it performs FD-Q after isolating the faulty portion. Since FD-Q requires a single cycle for HP-insts, double cycles for MP-insts and quad cycles for NP-insts, LIZARD ALU with a fault causes more performance loss than without it (ranges from 8 to 71 percent). We note that LIZARD is still fault-tolerable in the presence of a fault, even though the performance loss is non-negligible. In contrast, both TMR and DMR_S ALU are not fault-tolerable after a fault occurs even though they incurs same performance loss regardless of the presence of a fault. This is because they cannot tolerate more than a single fault as we discussed in Section 6.2.

6.3.2 Superscalar Processor

Fig. 17 compares the performance impacts of LIZARD ALU with those of TMR and DMR_S ALU for a superscalar processor. We evaluate the performance impacts of LIZARD with various configurations: Ideal, Simple, NPIB4, NPIB8 and NPIB16. 'Ideal' indicates a scenario that the latency prediction is perfect so that there is no penalty caused by the miss-speculative instruction wakeup presented in Section 5.2.1. 'Simple' indicates a scenario that ALU instructions are always predicted as P-insts while 'NPIB' indicates a scenario that a P-inst predictor is employed with various numbers of NPIB entries.

The performance overhead of LIZARD ALU is ignorable for most benchmarks. This is because most ALU instructions are P-insts in SPEC CPU2006 benchmarks as shown in Fig. 2. However, for some benchmarks such as h264ref and omnetpp, LIZARD ALU with 'simple' configuration sacrifices performance significantly. This is because these benchmarks show high IPC (Instruction Per Cycle) and there are relatively large number of NP-insts. With the P-inst predictor, the performance loss is reduced significantly. LIZARD ALU with the P-inst predictor having 8- and 16-entry NPIB incur 3.6 and 3.1 percent performance loss, respectively, on average while TMR and DMR_S ALU reduce performance by 3.8 and 4.0 percent, respectively.

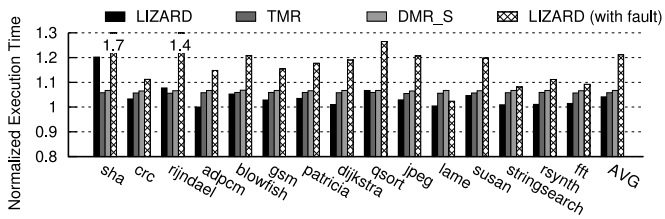


Fig. 16. Performance overheads for a scalar processor.

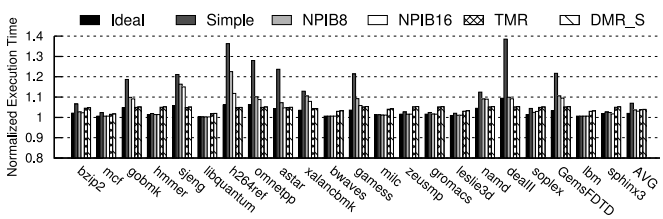
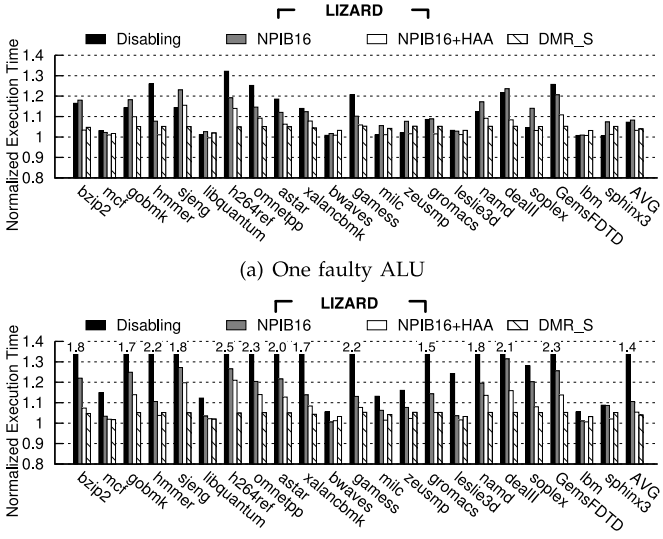


Fig. 17. Performance overheads for a superscalar processor.



(a) One faulty ALU

(b) Two faulty ALU

Fig. 18. Performance overheads for a superscalar processor in the presence of faulty ALUs.

For superscalar processors, the simplest approach to mitigate hard faults in the ALU is disabling the faulty ALU without employing any fault tolerant mechanisms since other ALUs are still functional. However, this approach incurs significant performance degradation as we can see in Fig. 18 (on average, 7.2 percent after disabling one ALU and 40.4 percent after disabling two out of three ALUs). On the other hand, a processor using LIZARD ALU with the P-inst predictor having 16-entry NPIB experiences 10.5 percent performance loss even when two ALUs are faulty. Moreover, in conjunction with the HAA (Heterogeneity-aware ALU Allocation) scheme presented in Section 5.2.3, the performance overhead of LIZARD ALU approaches that of DMR_S ALU.

6.4 Energy Efficiency

6.4.1 Scalar Processor

The fault-tolerant mechanisms increase the dynamic and leakage powers of the ALU. Moreover, the increase in the execution time results in more leakage energy consumption of other components such as the L2 cache which is a major contributor to the total energy consumption.

Fig. 19 shows the normalized energy consumption for a scalar processor using the fault-tolerant ALUs. As can be seen in the figure, LIZARD ALU is more energy efficient than TMR and DMR_S ALUs. It slightly increases energy consumption (by 6.2 percent on average) while TMR and

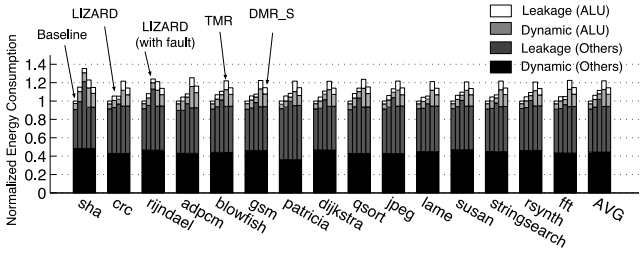


Fig. 19. Energy consumption breakdown for a scalar processor.

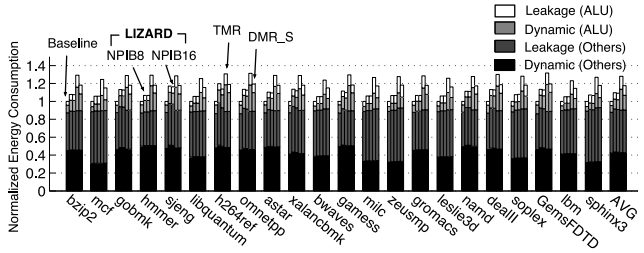


Fig. 20. Energy consumption breakdown for a superscalar processor.

DMR_S ALUs suffer from significant energy consumption overheads (22 and 14.3 percent, on average, respectively). This is because dynamic and leakage power of LIZARD ALU are much smaller than those of TMR and DMR_S, and its execution time overhead is also small for most benchmarks as we have seen in Section 6.3. Even if LIZARD ALU suffers from considerable performance degradation for the sha benchmark, it is more energy-efficient than TMR ALU and comparable to DMR_S ALU even for this benchmark.

In the presence of faults, LIZARD ALU increases energy consumption by 11.5 percent, on average. This is mainly caused by the additional leakage energy consumed by the other components due to increased execution time. LIZARD ALU deactivates its faulty half-word ALU and performs fault detection using its fault-free half-word ALU. This functionality minimizes the energy consumption overhead. For example, for the crc benchmark, the energy consumption overheads of LIZARD ALU with and without faults are almost the same. This is because the energy consumption of the ALU is reduced by deactivating its unused portion even though the other components consume more leakage energy.

6.4.2 Superscalar Processor

The energy efficiency of LIZARD ALU is also observed for a superscalar processor. On average, LIZARD ALU with the P-inst predictor having 8- and 16-entry NPIB consume respectively 7.5 and 6.8 percent more energy than the baseline as shown in Fig. 20. On the contrary, TMR and DMR_S ALU consume average 27.1 and 18 percent more energy, respectively. For some benchmarks, the portion of dynamic energy consumed by other components is slight increased. This is due to the P-inst predictor and the extended issue logic. However, their contribution to the total energy consumption is ignorable. As we have observed in a scalar processor, even if LIZARD ALU experiences non-negligible energy consumption overhead due to its execution time overhead for some benchmarks such as the sjeng and h264ref, it is still more energy efficient than its competitors for these benchmarks.

Finally, we examine the energy consumption for a superscalar processor having faulty ALUs. As shown in Fig. 21, LIZARD ALU is most energy-efficient across all benchmarks. Processor using LIZARD ALUs with the help of the P-inst predictor and HAA scheme consumes 2.9 percent more energy than baseline even if two ALUs are faulty. We note that the energy consumption overheads of LIZARD ALU with faults are lower than without faults. This is because the performance overheads of LIZARD ALU are

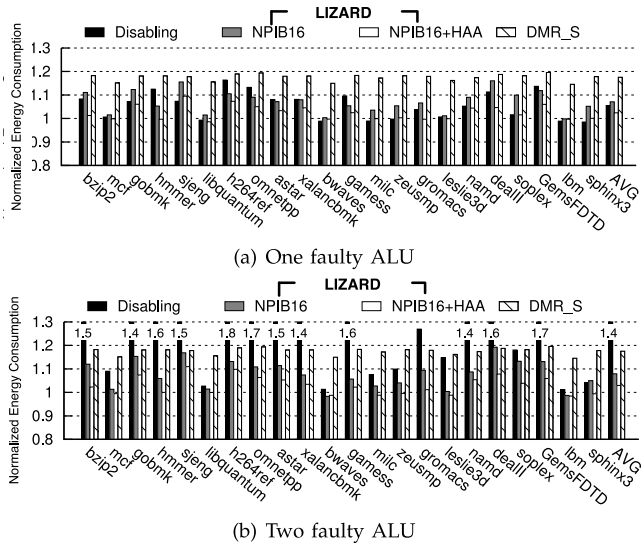


Fig. 21. Energy consumption of a superscalar processor in the presence of faulty ALUs.

small even in the presence of faults and it can save energy consumption by deactivating its faulty portions.

7 Related Work

To minimize hardware costs for fault tolerance, time redundancy techniques have been proposed. The basic idea of time redundancy is the repetition of computations on a same hardware. RESO [3] performs computation with the original values first and with modified values again. This technique suffers from performance degradation and provides only fault detection capability. REDWC [23] was proposed to reduce the performance overhead of RESO, but it also provides only fault detection.

In order to protect adders against hard faults with low cost, several techniques are proposed. Kumar et al. propose a fault detection technique for carry-select adder [24]. Ghosh et al. propose a fault isolation technique for KSA by exploiting the fact that even and odd carries are mutually exclusive in the KSA [25]. A similar technique was proposed in [26], which provides fault detection and diagnosis as well as fault isolation for parallel prefix adders. None of these techniques can be applied to various types of adder designs or provide complete fault-tolerant mechanism (fault detection, diagnosis, and isolation).

Recently, negative bias temperature instability (NBTI) and process variation (PV) are becoming growing reliability concerns in nanoscale technology. To mitigate these two sources of hard faults in the ALU, several techniques have been proposed. In [27], an elastic clocking technique is proposed to tolerate delay failures due to process variation and/or voltage scaling. A similar technique is proposed for tolerating NBTI in [28]. Even if these techniques can tolerate NBTI and PV-induced hard faults, they cannot tolerate other sources of hard faults.

Narrow-width values have been exploited to increase performance [5], [8] and to reduce energy consumption [6], [9], and to tolerate soft errors [4], [7], [10], [11]. In [7], [10], soft error-tolerant data-holding structures such as register files, issue queue, and caches are proposed. When a data is narrow, it is stored in these structures while duplicating its lower portion (non-zero) into the upper portion (zero). By comparing the two portions, the soft errors can be detected. In the similar way, operand values are duplicated in the ALU and the upper and lower portions of its output values are compared to detect the soft errors [4], [11]. In this paper, we leverage the main idea of this error detection mechanism, but with the following key differences. First, LIZARD can perform the fault diagnosis as well as the concurrent fault detection. Second, it can continue its operation even in the presence of hard faults. Third, we discussed the detailed implementation issues such as partitioned ALU. Moreover, we discussed the implications on the processor architecture when employing LIZARD.

8 Conclusions

We proposed a new versatile fault-tolerant ALU design called LIZARD that can combat various types of hard faults such as manufacturing defects, wearout-related faults, and process variation-induced faults. LIZARD employs a modular design approach to detect, diagnose and isolate hard faults. By exploiting narrow-width values, LIZARD reduces the overhead of conventional techniques such as time and hardware redundancy-based techniques. We implemented LIZARD on top of various adder designs. Moreover, we presented the architectural modifications for the scalar as well as superscalar processor employing LIZARD. Compared to competing schemes, LIZARD is shown to improve the reliability of the ALU with low cost.

Acknowledgments

This work was partly supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government [No. 2010-0029366 and No. 2014R1A2A2A01007051] and by the IT R&D program of MOTIE/KEIT [10048843, Automotive ECU SoC and Embedded SW for Multidomain Integration].

References

[1] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Jan. 2005.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The case for lifetime reliability-aware microprocessors," in *Proc. 31st Ann. Int. Symp. Comput. Architect.*, 2004, pp. 276–287.

[3] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALU's by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. 31, no. 7, pp. 589–595, Jul. 1982.

[4] S. Hong and S. Kim, "TEPS: Transient error protection utilizing sub-word parallelism," in *Proc. IEEE Comput. Soc. Ann. Symp. VLSI*, 2009, pp. 286–291.

[5] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proc. 5th Int. Symp. High Perform. Comput. Architect.*, 1999, pp. 13–22.

[6] S. Kim, "Reducing ALU and register file energy by dynamic zero detection," in *Proc. Int. Perform. Comput. Commun. Conf.*, 2007, pp. 365–371.

[7] J. Hu, S. Wang, and S. Ziavras, "In-Register duplication: Exploiting narrow-width value for improving register file reliability," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2006, pp. 281–290.

[8] T. Sato and I. Arita, "Table size reduction for data value predictors by exploiting narrow width values," in *Proc. 14th Int. Conf. Supercomput.*, May 2000, pp. 196–205.

[9] R. Canal, A. Gonzalez, and J. E. Smith, "Very low power pipelines using significance compression," in *Proc. 33rd Ann. IEEE/ACM Int. Symp. Microarchitect.*, 2000, pp. 181–190.
[10] O. Ergin, O. Unsal, X. Vera, and A. Gonzalez, "Reducing soft errors through operand width aware policies," *IEEE Trans. Dependable Secure Comput.*, vol. 6, no. 3, pp. 217–230, Jul.-Sep. 2009.
[11] R. Hyman Jr., K. Bhattacharya, and N. Ranganathan, "Redundancy mining for soft error detection in multicore processors," *IEEE Trans. Comput.*, vol. 60, no. 8, pp. 1114–1125, Aug. 2011.
[12] S. Hong and S. Kim, "Lizard: Energy-efficient hard fault detection, diagnosis and isolation in the ALU," in *Proc. 29th IEEE Int. Conf. Comput. Des.*, 2010, pp. 342–349.
[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.
[14] K. Yeager, "The Mips R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–40, Apr. 1996.
[15] R. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.
[16] D. Ernst and T. Austin, "Efficient dynamic scheduling through tag elimination," in *Proc. 29th Ann. Int. Symp. Comput. Architect.*, May 2002, pp. 37–46.
[17] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.-Y. Lueh, "XTREM: A power simulator for the Intel XScale core," in *Proc. ACM SIGPLAN/SIGBED Conf. Languages, Compilers Tools Embedded Syst.*, 2004, pp. 115–125.
[18] E. Lason, S. Chatterjee, and T. Austin, "MASE: A Novel infrastructure for detailed microarchitectural modeling," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, Nov. 2001, pp. 1–9.
[19] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, Apr. 2013.
[20] S. Dutt and F. Hanchek, "REMOD: A new methodology for designing fault-tolerant arithmetic circuits," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 5, no. 1, pp. 34–56, Mar. 1997.
[21] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. San Mateo, CA, USA: Morgan–Kaufman, 2007.
[22] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, 1962.
[23] B. Johnson, J. Aylor, and H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder," *IEEE J. Solid-State Circuits*, vol. 23, no. 1, pp. 208–215, Feb. 1988.
[24] B. K. Kumar and P. K. Lala, "On-line detection of faults in carry-select adders," in *Proc. Int. Test Conf.*, 2003, pp. 912–918.
[25] S. Ghosh, P. Ndai, and K. Roy, "A novel low overhead fault tolerant Kogge–Stone adder using adaptive clocking," in *Proc. Conf. Des., Autom. Test Eur.*, 2008, pp. 366–371.
[26] W. Rao and A. Orailoglu, "Towards fault tolerant parallel prefix adders in nanoelectronic systems," in *Proc. Conf. Des., Autom. Test Eur.*, 2008, pp. 360–365.
[27] D. Mohapatra, G. Karakonstantis, and K. Roy, "Low-power process-variation tolerant arithmetic units using input-based elastic clocking," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug. 2007, pp. 74–79.
[28] Y. Chen, H. Li, J. Li, and C.-K. Koh, "Variable-latency adder (VL-adder): New arithmetic circuit design practice to overcome NBTI," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug. 2007, pp. 195–200.

Seokin Hong received the BE degree in computer science and engineering from Sungkyunkwan University, Seoul, Korea, in 2008. Currently, he is working toward the PhD degree in the Department of Computer Science at Korea Advanced Institute of Science and Technology, Daejon, Korea. His major research experiences and interests include design of low-power, reliable, and high-performance microarchitectures and memory systems. He received best paper awards from International Conference on Computer Design (ICCD) in 2010 and Design Automation & Test in Europe (DATE) in 2013.

Soontae Kim received the PhD degree in computer science and engineering from Pennsylvania State University, State College, PA, in 2003. He has been with the Department of Computer Science, Korea Advanced Institute of Science and Technology, Daejeon, Korea, since 2007, as an Associate Professor. Previously, he was an Assistant Professor with the Department of Computer Science and Engineering, University of South Florida at Tampa, Tampa, FL, from 2004 to 2007. His research interests include embedded systems and low-power, real-time and reliable architecture/software. He received best paper awards from International Conference on Computer Design (ICCD) in 2010 and Design Automation & Test in Europe (DATE) in 2013. He is an Associate Editor of the *IEEE Transactions on Computer-Aided Design* and the *Journal of Institute of Embedded Engineering of Korea*. He is serving as a TPC Member of many international conferences, including International Symposium on Low Power Electronics and Design, Design Automation & Test in Europe. He is a member of the IEEE.