Self-Checking and Self-Diagnosing 32-bit Microprocessor Multiplier*

Mahmut Yilmaz, Derek R. Hower, Sule Ozev, Daniel J. Sorin
Duke University
Dept. of Electrical and Computer Engineering

Abstract

In this paper, we propose a low-cost fault tolerance technique for microprocessor multipliers, both non-pipelined (NP) and pipelined (P). Our fault tolerant multiplier designs are capable of detecting and correcting errors, diagnosing hard faults, and reconfiguring to take the faulty subunit off-line. We utilize the branch misprediction recovery mechanism in the microprocessor core to take the error detection process off the critical path. Our analysis shows that our scheme provides 99% fault security and, compared to a baseline unprotected multiplier, achieves this fault tolerance with low performance overhead (5% for NP and 2.5% for P multiplier) and reasonably low area (38% NP and 26% P) and power consumption (36% NP and 28.5% P) overheads.

1 Introduction

The common trend towards smaller feature sizes enables increased performance and transistor density for IC manufacturers. However, smaller feature sizes also increase the susceptibility of electronic circuits to manufacturing defects, process variability, and in-field wear-out [7]. Effects such as electromigration [18] and thermal scaling [20] will become more prevalent as wire sizes continue to shrink [16]. Moreover, shrinking gate-oxide thickness results in increased stress on the oxide layer, leading to gate oxide breakdowns, particularly for weak oxide layers. Since the lifetime of the oxide layer depends on its initial purity [17], it is unlikely that even costly test techniques, such as burn-in [6] or high voltage stress [9], will capture all transistors with weak oxide layers. As a result, it is increasingly likely for circuits in commodity products (such as microprocessors) to fail before they become obsolete.

For commodity microprocessors with small profit margins, hard fault tolerance schemes have traditionally been considered too costly. However, with the changing dynamics of failure mechanisms, it is increasingly inefficient to discard chips due to a small number of hard faults in the logic circuits. Therefore, designers face a new challenge of providing some degree of hard fault tolerance in their circuits while minimizing the hardware, performance, and power consumption overheads.

Tolerating failures in circuits at various levels of hierarchy, such as the system level, functional unit (e.g., adders, multipliers) (FU) level, or sub-FU level, is an intensely researched area in the literature. As a result, there is a plethora of fault tolerance techniques [8, 5, 3, 12] which we will discuss in greater detail in Section 2. However, direct application of these techniques to multipliers in commodity microprocessors is limited due to two major concerns. First, a fault tolerance technique in this domain needs to incur extremely low performance overhead, usually precluding the use of techniques that require time redundancy. Second, both area and power consumption overheads need to be reasonably small, preventing the use of techniques that replicate entire units.

In this paper, we present a recursive, fault tolerant 32-bit multiplier in the context of modern microprocessors. While not heavily utilized, multipliers are typically large, singleton functional units within the microprocessor core, the correctness of which determines the correctness of the overall system. To minimize the performance overhead, we utilize the processor's built-in branch misprediction recovery mechanism which enables us to take the error detection circuit off the critical path. In order to make our technique applicable to pipelined designs and to keep the delay overhead low, we use a modulo-3 (mod-3) checker [14, 13] for error detection. We propose a diagnosis scheme which re-uses the built-in reconfiguration capability.

We provide detailed analyses of fault tolerance overheads (i.e., hardware, power, and performance), fault detection capability, and single points of failure for the proposed scheme and compare them to prior work. Our main contributions in this work are:

- We take error detection off the critical path by re-

*This work is supported by NASA under Contract Number NNG04GQ06G and by NSF under Contract Numbers CCF-0545456, CCF-0444516, and CCR-0309164.

using existing capabilities in the microprocessor, and we thus enable extremely low performance overhead.

- We keep the hardware overhead reasonably low by re-using the existing reconfiguration capability for diagnosis.

- We evaluate the fault tolerant design in terms of hardware, power consumption, and performance overhead. We also provide a detailed analysis of fault security based on transistor level faults and the single fault-at-a-time assumption.

2 Related Work

Tolerating failures in circuits at various levels of hierarchy is an intensely researched area in the literature. In this section, we will discuss several previously proposed fault tolerance approaches for arithmetic functional units (FUs).

Some techniques provide protection at the FU-level, including triple modular redundancy (TMR). TMR is an effective technique, but it requires the triplication of the protected FU and a voter circuit. Although the delay overhead of TMR is generally very low, the hardware and power consumption overheads are over 200%.

Sub-FU reconfigurability has also been studied [8, 5, 3]. REMOD [5] has been proposed as a general framework to provide detection, diagnosis, and reconfiguration for arithmetic units with multiple identical submodules. REMOD uses a combination of time and hardware redundancy. After the functional operation is finished, a check operation is started wherein the functional operation of each submodule is re-run by another submodule and the results compared. The check operation is done by shifting the functional inputs of each module in a circular fashion through multiplexers. For diagnosis, the complete operation is shifted once again. This dual shifting effectively ensures that the functional operation of each submodule is rerun three times by disjoint hardware, thus enabling the identification of the faulty module.

Johnson et al. [8] divide a 32-bit adder into two 16-bit blocks, and use a time redundancy-based technique for fault detection. The 16-bit blocks of the adder perform each operation twice, once with functional inputs and once with the same operands rotated. The operand rotating technique was further improved by Townsend et al. [19] to reduce the area overhead. Mokrian et al. propose a hybrid multiplier architecture [12]. Four Booth-encoded 32-bit multipliers are connected as a recursive multiplier to form a 64-bit multiplier. If the multiply operation is single-precision, the operation is run on three 32-bit units, enabling detection and diagnosis. No detection is provided for the double precision operations. Chen and Chen present a serial fault tolerant multiplier [3] based on time redundancy. Fault tolerance is provided by circularly shifting the inputs and recomputing. The timing overhead is 100%, and the area overhead is not reported.

Most prior fault tolerance techniques in the context of arithmetic FUs use a combination of time and hardware redundancy. There are several drawbacks of relying on time redundancy wherein the functional units are re-used to perform the check operation. First, the check operation cannot be taken off the critical path. Thus, such techniques generally incur a high delay overhead. Second, time-redundancy based techniques cannot be easily applied to pipelined FU designs due to the re-use of FUs during the check operation. In this work, we aim at addressing these issues through two mechanisms. First,we allow the microprocessor to speculatively use the results from the multiplier and allow the multiplier to start a new operation before the check operation is complete, thereby taking this operation off the critical path of execution. Second, we provide disjoint hardware to perform the check operation to make our technique applicable to pipelined multiplier designs.

3 Fault Tolerant Multiplier

Multiplication is a complex operation for circuits to execute quickly, requiring the weighted addition of n^2 partial products for an nxn multiplication. Many serial multipliers can accomplish this operation with little area and power consumption but have a high latency. As an alternative, there exist many styles of parallel multipliers, such as tree (e.g., Dadda/Wallace), array, or recursive multipliers, that add partial products simultaneously. Among these, we have selected a recursive baseline multiplier over a tree or array based multiplier. The reasons include:

- Recursive multipliers are easy to lay-out due to their highly regular structure.

- Recursive multipliers can easily be decomposed into modular blocks to create reconfigurable units.

- Recursive multipliers ($O(log(n))$ delay, $O(n^2log(n))$ circuit complexity) have comparable performance to Dadda/Wallace multipliers ($O(log(n))$ delay, $O(n^2log(n))$ circuit complexity), and they are faster than array multipliers ($O(n)$ delay, $O(n^2)$ circuit complexity) [4, 10].

Recursive multipliers execute by breaking the operation into four smaller multiplications, as illustrated mathematically in Equation 1.

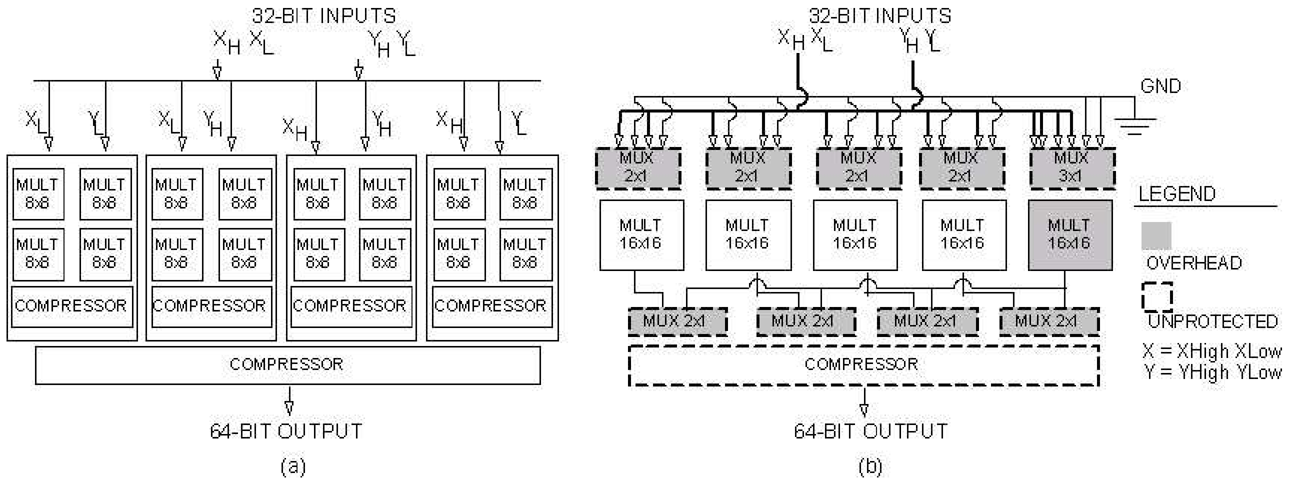

Figure 1: (a) Baseline 32-bit recursive multiplier. Note that 8-bit multipliers are also recursive. (b) Protected 32-bit recursive multiplier. The extra logic over the baseline multiplier is denoted with gray color. The unprotected areas are denoted with dashed borders.

$$X = \sum_{i=\frac{n}{2}}^{n-1} x_i 2^i + \sum_{i=0}^{\frac{n}{2}-1} x_i 2^i = X_H + X_L$$
$$Y = \sum_{i=\frac{n}{2}}^{n-1} y_i 2^i + \sum_{i=0}^{\frac{n}{2}-1} y_i 2^i = Y_H + Y_L$$
$$X \cdot Y = (X_H + X_L) \cdot (Y_H + Y_L) \qquad (1)$$

Each input operand is divided into two parts as high (H) and low (L) bits. The breakdown of the multiplication into smaller multiplication operations can be applied recursively until multiplication units are small enough to be efficiently implemented using high performance logic.

3.1 Baseline Unprotected Multiplier

Figure 1-a shows the block diagram of the baseline unprotected recursive multiplier (illustrated down to the 8-bit level). The 8-bit multipliers are further broken into 4 4-bit multipliers. A compressor circuit performs a weighted addition of the recursive products. The compressor circuit is composed of fast carry-propagating 4:2, 3:2, and 2:2 compressor building blocks. We implement both a non-pipelined (NP) and a pipelined (P) version of the baseline multiplier. We set the pipeline depth of our multiplier to eight cycles in order to match its cycle time to that of the processor model, which we will discuss in greater detail in Section 4. To enable a detailed analysis at the circuit level, we generate our baseline multiplier using the high performance logic designs presented in [11].

3.2 Fault Tolerant Multiplier

Fault tolerance generally requires three features: spare reconfigurable units (RUs) and a reconfiguration mechanism (3.2.1), an error detection mechanism to catch the erroneous output (3.2.2), and a diagnosis mechanism to find the source of the error (3.2.3). In the literature, there are many techniques for implementing each of the detection, diagnosis, and reconfiguration components required for fault tolerance. Our main contribution is focusing on an application domain (microprocessors) and utilizing the built-in capabilities of this application domain to develop an efficient design.

3.2.1 Reconfigurable Units and Reconfigurability

In our recursive multiplier, the RU can be at any level of hierarchy (i.e., 4-bit level, 8-bit level, 16-bit level). The selection of the RU size is a design decision. When the granularity of the RU is small, it is possible to map out only a small portion of a complete circuit while keeping most of the fault-free circuitry untouched. However, finer granularity results in larger area and performance overheads due to the necessity for more multiplexers and wires to enable reconfiguration. For microprocessors, performance overhead is a major concern. Therefore, we have selected the 16-bit multiplier module as the RU to minimize the performance overhead.

Figure 1-b illustrates the 32-bit multiplier with the spare RU and multiplexers. 2x1 multiplexers are added at the inputs and outputs of the $\frac{n}{2}$-bit RUs (except for the spare RU which needs 3x1 multiplexers at the input) to enable reconfiguration. When an RU is mapped out, the inputs of that RU are connected to ground to decrease the power consumption. The spare RU needs 3x1 multiplexers at its input since for each operand both high and low bits and a ground connection are required.

The top level compressor circuit remains unprotected in this configuration. Due to its small size relative to the rest of the circuit, we find this to be an acceptable design decision. Adding a spare compressor unit and using multiplexers to select that spare unit would not provide a better fault tolerance capability since the multiplexers would be of comparable size to the compressor circuit itself.

To keep the multiplication instruction latency constant after reconfiguration, we use the latency of 3x1 multiplexers instead of 2x1 multiplexers when calculating the delay overhead.

3.2.2 Error Detection and Correction

One way of providing an error detection mechanism is the use of an invariant for the circuit. Modulo arithmetic provides such an invariant for the multiplication operation, since:

$$(X)modA \cdot (Y)modA = (X \cdot Y)modA \qquad (2)$$

The modulo checker that utilizes this property includes two 32-bit modulo generators (input modulo generators), a 64-bit modulo generator (output modulo generator), a modulo multiplier, and a comparator. While any number A can be used as the base number, setting it to $(2^n \pm 1)$ enables us to break down the operands into n-bit slices and compute their modulos independently, leading to a low complexity design [14, 13].

We analyzed in detail the effect of using a modulo checker with A=3 and a modulo checker with A=7 for our multiplier design in terms of area overhead and fault coverage. We determined that the modulo checker with A=3 is preferable since it provides almost the same fault coverage as the modulo checker with A=7 with much smaller area. We found that the fault escape probability for the modulo checker with A=3 is 1% for the complete 32-bit multiplier design. The details of this calculation are given in Section 4.1.

A mod-3 generator is a tree structure consisting of 2-bit adders. Our implementation is similar to the one proposed by Piestrak [14]. The modulo multiplier calculates the product of the mod-3 of the inputs, which is a 2-bit number. Finally, the 2-bit mod-3 equality checker, which is a simple comparator, checks the consistency of calculated mod-3 values. Note that the equality checker is aware of the double representation of 0. A schematic showing this mechanism is given in Figure 2. Due to space constraints, the mechanism is shown for an 8-bit multiplier.

For error correction, we utilize the built-in branch misprediction recovery mechanism in modern microprocessors. When a branch instruction is encountered, the microprocessor first guesses the outcome of this instruction and continues executing speculatively without committing until the branch outcome is computed. If the outcome was guessed incorrectly, all instructions after the branch instruction are removed from the pipeline and fetch starts from the actual branch target instruction. We treat all multiply instructions in the same manner as branch instructions. We let

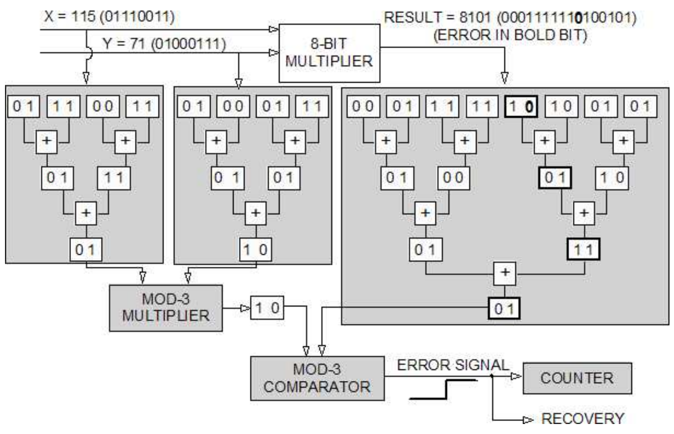

Figure 2: Detection mechanism for an 8-bit multiplier

the multiplier complete its operation and make its results immediately available before the modulo checker completes. However, we only commit the multiplier results after the checker completes. If the checker detects an error, the recovery mechanism flushes the pipeline starting with the multiply instruction.

At this point, we would begin diagnosis except that some of the detected errors may be due to transient errors, the rate of which is also increasing as device dimensions shrink [16]. For a hard fault tolerance mechanism, it is imperative to prevent transient faults from initiating an unnecessary reconfiguration. Our detection mechanism avoids such unnecessary reconfiguration through multiple passes. If an error is detected, a counter associated with the multiplier is incremented and the microprocessor's recovery mechanism is activated. After recovery, the same operands will enter the multiplier, causing the same erroneous result in case of a hard fault, and producing the correct response in case of a transient fault. If the correct result is encountered the second time, the error is considered transient, the counter is reset, and the operation continues without reconfiguration. To allow for adequate time for the propagation of transient errors, we use a 2-bit counter, which triggers diagnosis if the multiplication produces an erroneous response a third time.

3.2.3 Diagnosis

To prevent the modulo checker from being a SPF, upon entering the diagnosis, we first test the checker by using its inherent redundancy. The 64-bit output modulo generator is composed of two 32-bit modulo generators and a small adder circuit. Thus, we can switch the two 32-bit input modulo generators and the 64-bit output modulo generator by adding a small redundant 2-bit adder. This small adder enables the usage of the two 32-bit input modulo generators as a 64-bit output modulo generator. In this way, we can calculate the mod-3 value of the two operands as well as the product twice with non-overlapping hardware. If the checker is not faulty, the modulo results are expected to match for both passes. A non-matching result for any of the operands or the output is considered to show a *possible* hard fault in the checker. If the results differ, we perform this test a second time to account for transient faults. If the check fails for a second time in a row, the modulo checker is assumed to be faulty and it is disabled. The multiplier continues operating without a fault tolerance mechanism. We could instead reconfigure the circuit to re-use the checker. However, that would add complexity, more hardware and power overhead, so we have decided not to use this option. In a mission critical system, this option can be considered.

For diagnosing the faulty RU, we propose a scheme that re-uses our reconfiguration capability. After recovering the pipeline, the multiply instruction re-executes with the same operands. Thus, by rotating which RU is the spare before each recovery and re-execution, we can systematically diagnose which RU is faulty. The intuition behind this approach is that if an error is observed for a given spare configuration, then the RU used as a spare on that execution does not contain the fault under single fault assumption. Diagnosis incurs a one time penalty of at least one and at most four pipeline flushes. Because hard faults rarely occur, the effect of diagnosis on performance is negligible.

To implement this diagnosis algorithm, we introduce five extra bits of state. These bits, collectively labeled the *fault bits*, identify the current known fault status of each RU. At the start of diagnosis, all RUs are assumed to be faulty, and thus all the fault bits are set. When there is an error at the multiplier's output, the bit corresponding to the spare RU is cleared. This process repeats until only one bit remains unchanged. The RU whose fault bit was not cleared is identified as the faulty RU.

Although unlikely, it is possible for this scheme to produce an incorrect diagnosis. A transient fault that occurs on a cycle when the faulty RU is mapped out can result in the diagnosis algorithm incorrectly identifying a fault-free RU as faulty. While the probability of this occurring is small, we can minimize it by making the diagnosis mechanism a multistage scheme. After the first diagnostic pass completes, we set a *justMappedOut* bit that indicates that our scheme has made an initial guess. We then run a second diagnostic pass and identify the faulty RU. If the results differ, we conclude that a transient error has invalidated our diagnosis process and restart. Otherwise, a permanent reconfiguration bit is set at the end of the run. If a hard fault is again identified in the multiplier, then the entire multiplier is deemed unrecoverable. While this multistage optimization does not completely eliminate susceptibility to transient faults, we assume that the probability of a transient negatively impacting diagnosis twice in a row is effectively zero. A flowchart for the operation of error detection, correction, and diagnosis is given in Figure 3.

4 Experimental Evaluation

There are four goals of this evaluation:

- We want to show that the fault escape probability of our modulo checker for transistor level faults is very low (Section 4.1).
- We want to demonstrate that the unprotected area in our design is low and comparable to that of other fault tolerance techniques with much higher area and power overhead (Section 4.2).
- We want to show that the area and power overheads of our design are low compared to fault tolerance techniques with similar performance overhead. Finally, we want to show that the performance overhead of our design on a modern commodity microprocessor's run-time is very low (Section 4.3).

In most of the prior fault tolerance approaches for functional units, time redundancy has been used for error detection. In order to compare our design with previously proposed designs with similar area and power consumption overhead, we apply the general technique described in REMOD [5] to our recursive multiplier. In order to enable a fair comparison, we consider the same level of fault tolerance: the REMOD-based design also tolerates one fault at the 16-bit multiplier level and leaves the top level compressor circuit unprotected.

4.1 Fault Escape Probability

In our analysis of fault escape rates, we use transistor-level fault models and HSpice simulations with the $0.18\mu m$ process technology parameters at the circuit level. The transistor-level fault models include source-gate short, drain-gate short, source-drain short, bulk-gate short, source open, gate open and drain open. We obtain fault escape probabilities in a hierarchical manner. We simulate each non-modular low-level building block (i.e., the 4-bit multiplier, the compressor units) for all possible inputs and for all faults. From these simulations, we obtain input fault tables for these building blocks. We then propagate the information hierarchically using behavioral models under the single hard-fault assumption. Our goal is to compute what percentage of hard faults, once excited, is detectable when they cause an error. We assume that the inputs are uniformly distributed.

Our analysis of the 4-bit multiplier shows that 99.5% of its faults result in a change of the mod-3 value

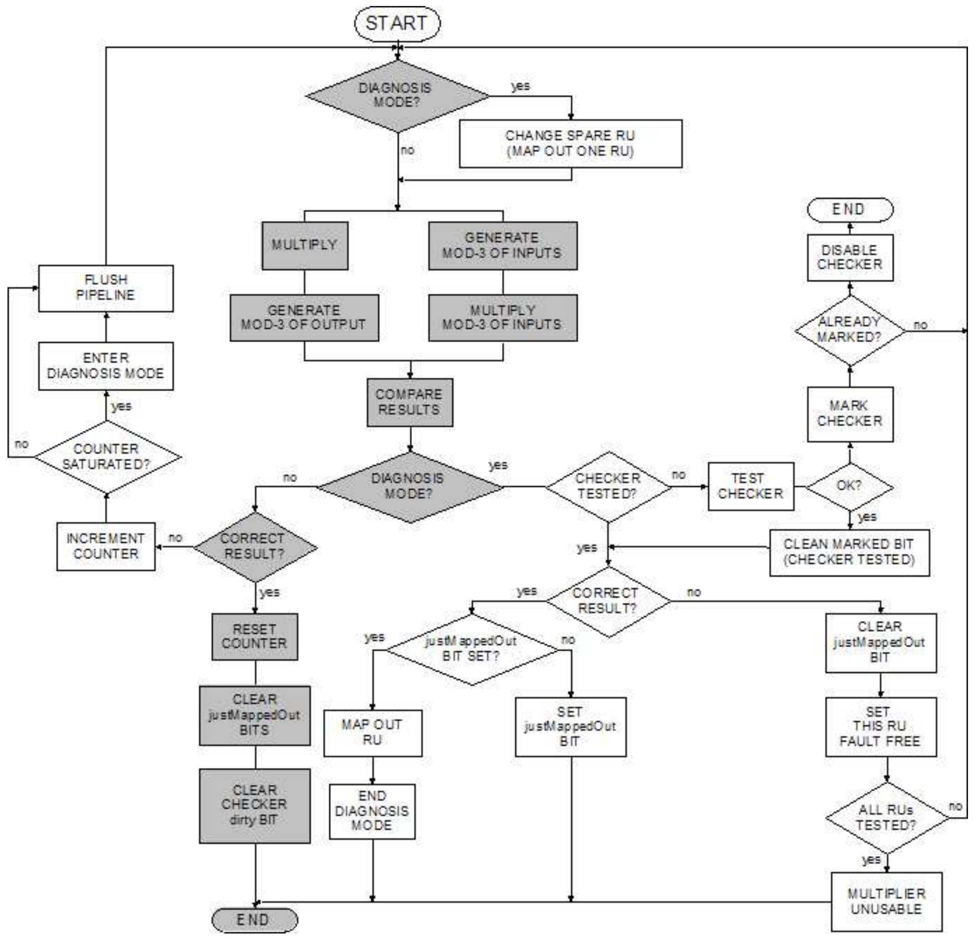

Figure 3: Error detection, diagnosis and correction mechanism flow-chart. Normal operation is shown in gray.

of its output when it contains an error. Note that this fault coverage is circuit dependent. By the single fault assumption, only one of the 4-bit multipliers can be faulty at a time. This error will be propagated to the output of the 32-bit multiplier by the following observation.

Assume that at the 8-bit multiplier level, the two 8-bit inputs are given as $X_H X_L$ and $Y_H Y_L$, where X_H, X_L, Y_H and Y_L correspond to the 4-bit slices of the two inputs. The 4-bit multipliers then will generate the products $X_L Y_L$, $X_L Y_H$, $X_H Y_L$, and $X_H Y_H$. Also, let us denote the output of the 8-bit multiplier by $P_{8\times8}$. The behavioral model of the compressor at the 8-bit multiplier level is the following:

$$P_{8\times8} = X_H Y_H \cdot 2^8 + (X_L Y_H + X_H Y_L)\cdot 2^4 + X_L Y_L \quad (3)$$

The mod-3 value of this product is:

$$(P_{8\times8})mod3 = (X_H Y_H)mod3 \cdot (2^8)mod3 + (X_L Y_H + X_H Y_L)mod3 \cdot (2^4)mod3 + (X_L Y_L)mod3 \quad (4)$$

Since $(2^n)mod3 \equiv 1$ for even n, both 2^8 and 2^4 equal 1 in mod-3. Then the equation reduces to,

$$(P_{8\times8})mod3 = (X_H Y_H)mod3 + (X_L Y_H + X_H Y_L)mod3 + (X_L Y_L)mod3 \quad (5)$$

Since each product $X_L Y_L$, $X_L Y_H$, $X_H Y_L$, and $X_H Y_H$ is generated by a separate multiplier, there is no single fault that can cause an error in more than one of these products. Then, if a fault changes the mod-3 value of the 4-bit multiplier output, it will change the mod-3 value of the 16-bit multiplier output. This fact can be used recursively to conclude that any fault that changes the mod-3 value of the 4-bit multiplier result will propagate to the 32-bit output and will be detected by the mod-3 checker. Thus, the fault escape probability for the faults occurring in the 4-bit multiplier module is only 0.5%.

To analyze the compressor circuit, we can use the hierarchy in a similar way. Compressor circuits at any level (8-bit multiplier, 16-bit multiplier or 32-bit multiplier) are composed of 3 different compressor building blocks: 4:2, 3:2, and 2:2. Each of these building blocks generates only one output bit, and they are connected in the form of a chain through carry signals. Because mod-3 numbers are 2-bit numbers, we can

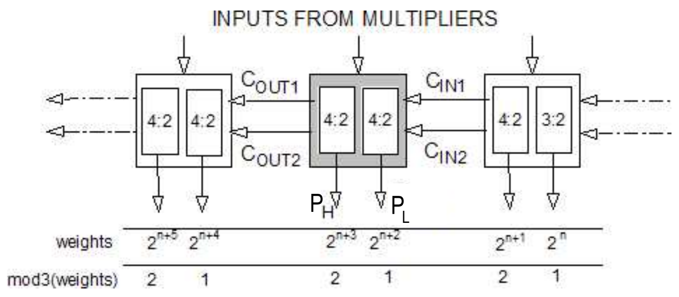

Figure 4: Compressor building blocks are grouped as 2-bit slices for HSpice simulations. An example of a 2-bit slice is shown in gray. The weights of the output bits are shown below the blocks.

partition the whole compressor circuit into 2-bit slices, as shown in Figure 4. In this way, each slice will generate a 2-bit part of the output. Since $(2^n)mod3 \equiv 1$ for even n, each 2-bit slice has the same weight for mod-3 calculation, and these slices can be added to obtain the overall mod-3 of the output.

Each 2-bit slice can change its own output bits (P_H, P_L) and its carry-out bits (C_{OUT1},C_{OUT2}). Thus, the contribution of each 2-bit slice to the mod-3 of the output is:

$$(P_L + 2 \cdot P_H + 4 \cdot (C_{OUT1} + C_{OUT2}))mod3 \quad (6)$$

which can be reduced to

$$(P_L + 2 \cdot P_H + C_{OUT1} + C_{OUT2})mod3 \quad (7)$$

Thus, each 2-bit slice can change the output without changing the mod-3 of the output only if its contribution $(P_L + 2 \cdot P_H + C_{OUT1} + C_{OUT2})mod3$ does not change.

In order to find the fault escape probability of the compressor circuits, the building blocks are simulated exhaustively - as described above - in slices of two, and the probability that the above mentioned situation can happen is calculated for each different slice. Then, the overall fault escape probability of a compressor circuit is found as the weighted average of the individual probabilities. Areas (approximated by the total number of transistors) of the 2-bit slices are used as their weights. The fault escape probabilities of compressor circuits are given in Table 1, where Compressor-X denotes the compressor circuit in an X-bit multiplier.

Errors due to multiplexer gates can be detected with 100% probability since an error that changes its output means a single bit-flip at the output of the multiplexer, which in turn corresponds to a bit-flip at the input of the compressor or the multiplier block. Any single bit-flip at the output can be detected with 100% probability as described by Equations 3-5. Similarly, faults in buffers (and in latches for the pipelined multiplier) can be detected with 100% probability.

Table 1: Fault Escape Probabilities (FEP)

	Frac. of circuit (%)		**FEP (%)**	
Circuit	**NP**	**P**	**NP**	**P**
Compressor-8	15.2	10.8	3.82	
Compressor-16	7.7	5.5	4.16	
Compressor-32	3.1	2.2	4.34	
Mult4x4	65	46.3	0.5	
Other	9	35.2	0.0	
Overall Mult.	100		1.27	0.97

To sum up, a mod-3 checker can miss a small fraction of faults in the compressor circuit and the 4x4 multiplier circuit. A weighted average of all probabilities yields an overall fault escape probability of 1.3% and 1% for the complete 32-bit NP- and P-multiplier designs, respectively.

For the REMOD-based design, the fault escape probability up to the 16-bit multiplier level is zero since the multiply operation itself is checked. However, the top level compressor has 100% fault escape probability (i.e. any fault that results in an error will go undetected). Thus, the overall fault escape probability of the REMOD-based design is given by the area fraction of the top-level compressor circuit: 3.1%.

4.2 Single Point of Failure Analysis

Our design can protect all 16-bit multipliers within the 32-bit multiplier. However, the top level compressor circuit and multiplexers are unprotected and they constitute single point of failures, i.e. a fault in these components will make the entire multiplier unusable. In addition, some parts of the modulo-3 checker (mod-3 multiplier and mod-3 comparator) are also single points of failure. In order to enable fair comparisons with respect to the baseline multiplier and other techniques, we first calculate the total number of transistors that are single points of failures. We then calculate what fraction of area of *the baseline multiplier* these single points of failures correspond to. We denote this percentage by *SPF*. The SPF for the baseline multiplier is 100%.

Table 2 compares the SPF of our design to triple modular redundancy (TMR), full duplication (without a diagnosis mechanism), and the REMOD-based scheme. Our analysis shows that the SPF of our multiplier is 14% (12%) for the non-pipelined (pipelined) design.

The SPF of the REMOD-based scheme stems from the top-level compressor circuit, multiplexers to switch the inputs, and the comparator circuits to compare partial products. Since the total bit width of the partial products is larger than the modulo of the product, the overall comparison circuit is larger, resulting in slightly larger SPF than our design. The 8% SPF of the TMR multiplier stems from the majority voter cir-

Table 2: Single Point of Failure Analysis Results. All percentages are based on transistor count

	Fraction of Unprotected Area (%)		Hardware Overhead (%)	
Circuit	**NP**	**P**	**NP**	**P**
Our Multiplier	14%	12%	38%	26%
REMOD	16%	N/A	36%	N/A
TMR	8%	5%	208%	205%
Duplication	2%	2%	102%	102%

cuit. In the full duplication case, the 2% SPF is due to input and output multiplexers, assuming a cold spare is implemented.

Our multiplier has a slightly smaller SPF compared to the REMOD-based scheme, which has similar area overhead and much higher performance degradation (discussed in Section 4.3). Moreover, our technique does not have much higher SPF than TMR, which has high area and power consumption overheads. Although the SPF of the duplication scheme is lower, such a scheme can only detect and reconfigure, but cannot diagnose which unit is faulty.

4.3 Performance, Hardware, and Power

Fault tolerant schemes incur hardware, performance, and power consumption overheads.

Hardware: The details of the hardware overhead of our fault tolerant multiplier are in Table 3. The largest overhead is due to the reconfiguration logic which includes the spare RU. The error detection circuit is 11% of the baseline multiplier. The area overhead of the diagnosis mechanism is extremely low due to the re-use of the reconfiguration capability during the diagnosis process.

Performance: The delay overhead of our fault tolerance mechanism, as shown in Table 4, is due to the multiplexers on the critical path. To evaluate timing and power overheads, we use HSpice simulations with $0.18\mu m$ process technology parameters. For the REMOD-based scheme, the delay overhead mainly arises from the recomputation of the 16-bit multiplications. This operation corresponds to 53% of the overall multiplier delay. For the REMOD-based scheme, we have optimistically ignored the additional delay overhead due to the multiplexers, as we did not simulate this scheme using HSpice.

We evaluate the performance overhead of the fault tolerance scheme in two phases. First, we evaluate the absolute delay overhead of the extra logic that falls on the critical path, which we find to be 6% for our fault tolerant multiplier and 53% for the REMOD scheme. Since an X% increase in the delay of the multiplier does not necessarily result in a X% overall microprocessor performance degradation, we simulate the execution flow of a modern microprocessor pipeline. There are several reasons for the diminished effect of the absolute delay overhead. First, multipliers are not typically used in every clock cycle even with workloads that are computation intensive, reducing the overall impact. Second, with the out-of-order execution in modern microprocessors, the increased latency of the multiplier can be overlapped with other data-independent instructions.

To evaluate the impact of the fault tolerance schemes on the microprocessor performance, we use a modified version of a detailed microprocessor simulator, SimpleScalar [2]. We model a superscalar out-of-order microprocessor that roughly resembles the AMD Opteron/Athlon [1]. Table 5 shows the details of the modeled microprocessor. The pipeline width of this processor is 3 macro instructions, but these are then cracked into multiple micro-ops, which is why there are so many functional units. We evaluate the performance overhead of the non-pipelined design and compare this overhead with that of the REMOD-based scheme. We also evaluate the performance overhead of the pipelined design. Since the application of the REMOD technique is non-trivial for pipelined designs, we do not consider this scheme in simulations for the pipelined multiplier. For benchmarks, we use the full SPECCPU2000 suite running with the reference input set. We sample the benchmarks with the SimPoint methodology [15] to reduce the simulation time.

For the non-pipelined case, to model our fault tolerant multiplier, we add a 1-cycle overhead to the 8-cycle multiplier latency to account for the 6% delay penalty found by HSpice simulations. We also increase the operational latency of the REMOD-based scheme by 4 clock cycles to account for its 53% latency overhead. Furthermore, for our scheme, we assume that the multiplication result is immediately available when the operation completes, but the instruction can-

Table 3: Hardware Overhead for Our Scheme

	Modules	**Hardware Overhead**	
Mechanism		**NP**	**P**
Reconfig.	Mux & spare unit	26.6%	17.7%
Error Det.	Mod-3 checker	10.8%	7.8%
Diagnosis	Control logic	0.7%	0.5%
Total	All extra logic	38.1%	26%

Table 4: Summary of Overheads ($0.18\mu m$ process)

	Area		**Delay**	**Avg Power**	
Circuit	**NP**	**P**		**NP**	**P**
Our multiplier	38%	26%	6%	36%	28%
REMOD	36%	N/A	53%	36%	N/A

Table 5: Parameters of the Microprocessor System

Feature	Details
pipeline stages	12
width: fetch/issue/commit	3/3/3
branch predictor	2-level GShare, 4K entries
instruction fetch queue	72 entries
reservation stations	54 entries
reorder buffer	72 entries
load/store queue	44 entries
integer ALU	6 units, 1-cycle
integer multiply/divide	1 unit, 8-cycle mult, 74-cycle div
floating point ALU	3 units, 5-cycle
floating point multiply/divide	1 unit, 24-cycle mult, 26-cycle div
floating point sqrt	35-cycle
L1 I-Cache	64KB, 2-way, 64-byte blocks, 3-cycles
L1 D-Cache	64KB, 2-way, 64-byte blocks, 3-cycles
L2 (unified)	1MB, 16-way, 128-byte blocks, 20-cycles

not be committed until checked for correctness. We add 2 clock cycles between complete and commit for multiplication instructions for this purpose. For the pipelined multiplier, we once again increase the latency of the multiplier from 8 to 9 cycles to account for the multiplexer delays. The pipelined multiplier can produce a result every clock cycle. We also add a 2-cycle latency between the complete and commit phases for the check operation to finish.

In the non-pipelined multiplier case, the normalized number of instructions per clock cycle (IPC) of the benchmarks are given in Figure 5 for the baseline multiplier, our protected multiplier, and the REMOD-based protected multiplier. For each benchmark, IPC is normalized using the baseline value. For our fault tolerant multiplier, the highest IPC degradation is around 10% for some floating point benchmarks, while the IPC degradation is much lower for the integer benchmarks. For the REMOD-based scheme, up to 26% IPC degradation is observed (for *art*). The degradation in the harmonic mean of the IPCs is found to be 5% for our scheme and 21% for the REMOD-based scheme. Thus, the actual performance overhead of our fault tolerant design on the execution of a microprocessor using an equally weighted mix of these benchmark applications is low. However, the effect of the REMOD-based scheme is considerable.

The simulation results for a pipelined multiplier are given in Figure 6. Our scheme's highest IPC degradation is 8% for *applu*, and the degradation in the harmonic mean of the benchmark IPCs is 2.5%.

The delay overheads due to diagnosis and pipeline flushes are only transient effects during reconfiguration. Thus, we do not include these effects in our calculations.

Power Consumption In order to keep the power consumption overhead low, we connect the inputs of the spare RU to ground in the normal functional mode, making its power consumption negligible (due to leakage only). After reconfiguration, the inputs of the faulty RU are connected to ground by switching the multiplexer select lines. This enables a uniform power consumption before and after reconfiguration. As shown in Table 4, the power consumption overhead of our multiplier is 36% for non-pipelined and 28.5% for pipelined designs, including the modulo checker.

5 Conclusion

We have developed a low-cost, fault-tolerant recursive multiplier for use in commodity microprocessors. Our design can detect and correct errors, diagnose the fault location, and reconfigure itself to map out faulty sub-units. We conclude that cost-effective fault tolerant multipliers can be designed with sufficient performance to be feasible for high-performance commodity microprocessors. The key insight was to consider the multiplier as it operates within the microprocessor, rather than in isolation. This insight enabled us to leverage the microprocessor's existing recovery mechanism, and it focused our circuit design efforts on those portions of the multiplier that impacted the critical path of microprocessor operation.

Acknowledgments

We would like to thank Dr. Ismet Bayraktaroglu for his inputs on modulo checkers.

References

[1] AMD. Software Optimization Guide for the AMD64 Processors. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs /25112.PDF, Sep 2005.

[2] T. Austin, E.Larson, and D.Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb 2002.

[3] L. Chen and T. Chen. Fault-tolerant Serial-parallel Multiplier. In *IEE Proc. on Computers and Digital Techniques*, volume 138, pages 276–280, July 1991.

[4] A. Danysh and E. Swartzlander. A Recursive Fast Multiplier. In *Proc. of Asilomar Conf. on Signals, Systems and Computers*, volume 1, pages 197–201, Nov 1998.

[5] S. Dutt and F. Hanchek. REMOD: A New Methodology for Designing Fault-Tolerant Arithmetic Circuits. *IEEE Trans. on VLSI Systems*, 5(1):34–56, Mar 1997.

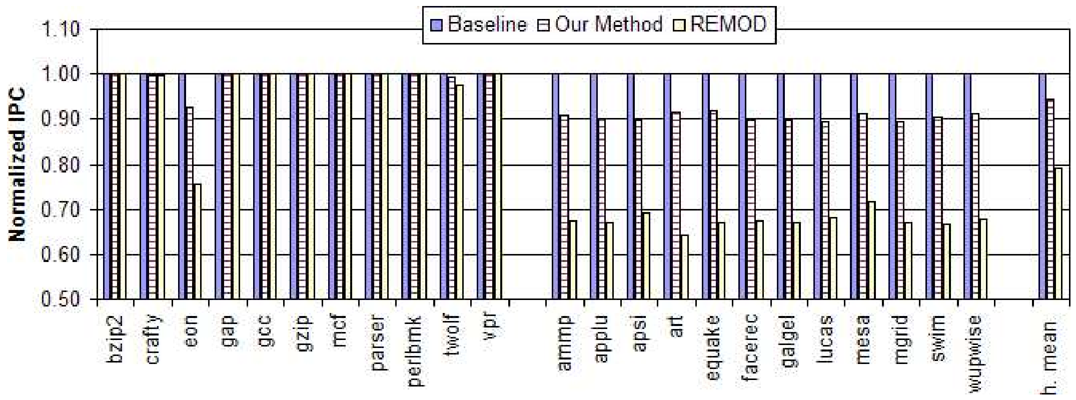

Figure 5: Performance Impact of fault tolerance schemes for the microprocessor with the non-pipelined multiplier

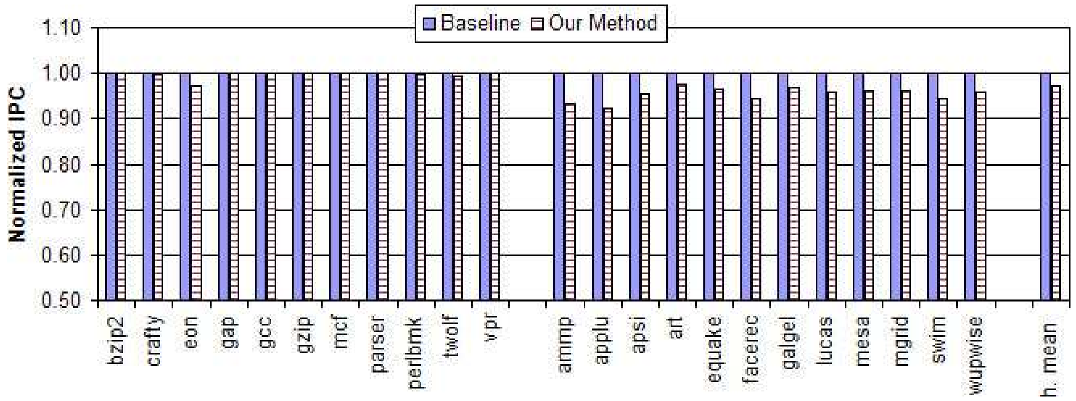

Figure 6: Performance Impact of the fault tolerance scheme for the microprocessor with the pipelined multiplier

[6] J. Forster. Single Chip Test and Burn-in. In *Proc. of IEEE Electronic Components and Technology Conf.*, pages 810–814, May 2000.

[7] ITRS. International Technology Roadmap For Semiconductors . http://public.itrs.net/Files/2003ITRS, 2003.

[8] 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 JSSC*, 23(1):208–215, Feb 1988.

[9] M. Khalil and C. Wey. High-voltage Stress Test Paradigms of Analog CMOS ICs for Gate-oxide Reliability Enhancement. In *Proc. of IEEE VTS*, pages 333–338, May 2001.

[10] J. Kim and E. Swartzlander. Improving the Recursive Multiplier. In *Proc. of Asilomar Conf. on Signals, Systems and Computers*, volume 2, pages 1320–1324, Nov 2000.

[11] R. Lin, M. Margala, and N. Kazakova. A Novel Self-Repairable Parallel Multiplier Architecture. In *Proc. of IEEE Asia-Pacific Conf. on ASIC Design and Test*, pages 29–32, Aug 2002.

[12] P. Mokrian, M. Ahmadi, G. Jullien, and W. Miller. A Reconfigurable Digital Multiplier Architecture. In *IEEE Canadian Conf. on Electrical and Computer Engineering*, volume 1, pages 125–128, May 2003.

[13] S. Piestrak. Design of Residue Generators and Multi-operand Adders Modulo 3 Built of Multioutput Threshold Circuits. *IEE Proceedings - Computers and Digital Techniques*, 141(2):129–134, Mar 1994.

[14] S. Piestrak. Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders. *IEEE Trans. on Computers*, 43(1):68–77, Jan 1994.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[16] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *IEEE/ACM Int. Symp. on Computer Architecture*, pages 276–287, Jun 2004.

[17] J. Stathis. Physical and Predictive Models of Ultra-thin Oxide Reliability in CMOS Devices and Circuits. *IEEE Trans. on Device and Materials Reliability*, 1(1):43–59, Mar 2001.

[18] J. Tao, J. Chen, N. Cheung, and C. Hu. Modeling and Characterization of Electromigration Failures Under Bidirectional Current Stress. *IEEE Trans. on Electron Devices*, 43(5):800–808, May 1996.

[19] W. Townsend, J. Abraham, and E. Swartzlander. Quadruple Time Redundancy Adders. In *Proc. of IEEE DFT*, pages 250–256, Nov 2003.

[20] J. Yang, J. Oh, K. Im, I. Baek, C. Ahn, J. Park, W. Cho, and S. Lee. Thermal Scaling of Ultra-thin SOI: Reduced Resistance at Low Temperature RTA. In *Proc. of IEEE European Solid-State Device Research Conf.*, pages 153–156, Sep 2004.