Appears in the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)
Vancouver, BC, Canada, December 2013

Applying Reduced Precision Arithmetic to Detect Errors in Floating Point Multiplication

Kushal Seetharam, Lance Co Ting Keh, Ralph Nathan, and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
Durham, NC USA
{kushal.seetharam, lance.co.ting.keh, ralph.nathan}@duke.edu, sorin@ee.duke.edu

Abstract—**Prior work developed an efficient technique, called reduced precision checking, for detecting errors in floating point addition. In this work, we extend reduced precision checking (RPC) to multiplication. Our results show that RPC can successfully detect errors in floating point multiplication at relatively low cost.**

Keywords—fault tolerance, floating point, error detection

I. INTRODUCTION

One of the primary purposes of computers is to perform arithmetic. In particular, many categories of software perform vast amounts of floating point arithmetic. Scientific applications—including simulators of all kinds of physical phenomena—use floating point arithmetic to computationally evaluate models. Financial applications use floating point arithmetic to computationally evaluate models of economies and businesses. Other types of applications, including military, avionics, and gaming, use floating point arithmetic as well.

Because floating point arithmetic is critical for many applications, there are many situations in which an error in a floating point calculation could be problematic or even disastrous. Errors can arise due to physical faults such as transient high-energy particle strikes or permanent wearout of transistors, and industry projects that faults are likely to be more common as future transistor and wire dimensions continue to shrink [1].

In this paper, we focus on detecting errors in floating point arithmetic performed by the floating point units (FPUs) in processor cores. In particular, we focus on floating point multiplication, and we extend a previously developed scheme, called *reduced precision checking (RPC)* [2], that has previously been applied only to addition.

Some other prior work exists detecting errors in floating point arithmetic, but it has drawbacks compared to RPC. Lipetz and Schwarz at IBM [6] propose residue checking, which is complete and cost-efficient, but the paper provides insufficient detail for us to reproduce their results. We cannot devise a residue checking scheme that handles rounding. Based on an IBM patent [3], we believe that the rounding information is passed to the residue checker from the checked unit, which then implies that errors in the rounding logic are undetected. Maniatakos et al. [7] propose checking the exponents of floating point computations, which detects many errors but fewer than if one also checks mantissas.

The contribution of this paper is to demonstrate that RPC is a viable option for the important problem of detecting errors in floating point multiplication. In Section II, we present the background work on RPC and how it was previously applied to addition. In Section III, we explain how to extend RPC to multiplication. In Section IV, we describe our hardware implementation of RPC for multiplication. In Section V, we present our experimental evaluation of RPC's ability to detect injected errors. In Section VI, we evaluate RPC's area and power overheads. In Section VII, we compare RPC for multiplication to RPC for addition. We conclude in Section VIII.

II. REDUCED PRECISION CHECKING

In the RPC scheme for addition [2], a 32-bit floating point adder is checked by a k-bit (k<32) reduced-precision floating point checker adder. The checker adder has a sign bit and the same number of exponent bits, but it has fewer mantissa bits. Comparing the most significant bits of the result of the 32-bit adder to the result of the k-bit checker adder reveals whether an error occurred.

RPC offers a tune-able tradeoff of cost versus error coverage. Reducing the checker adder's mantissa size reduces the cost of the checker but increases the likelihood of missing an error. Specifically, a shorter checker adder increases the magnitude of an error that can be undetected by RPC.

The design challenge in RPC is that, even in the absence of errors, the most significant mantissa bits of the checker do not necessarily match the most significant mantissa bits of the adder. This small possible discrepancy is due to the checker adder truncating the least significant mantissa bits. The RPC authors added logic to detect when discrepancies were possible in the absence of errors and thus not signal an error in such situations. As a result, RPC

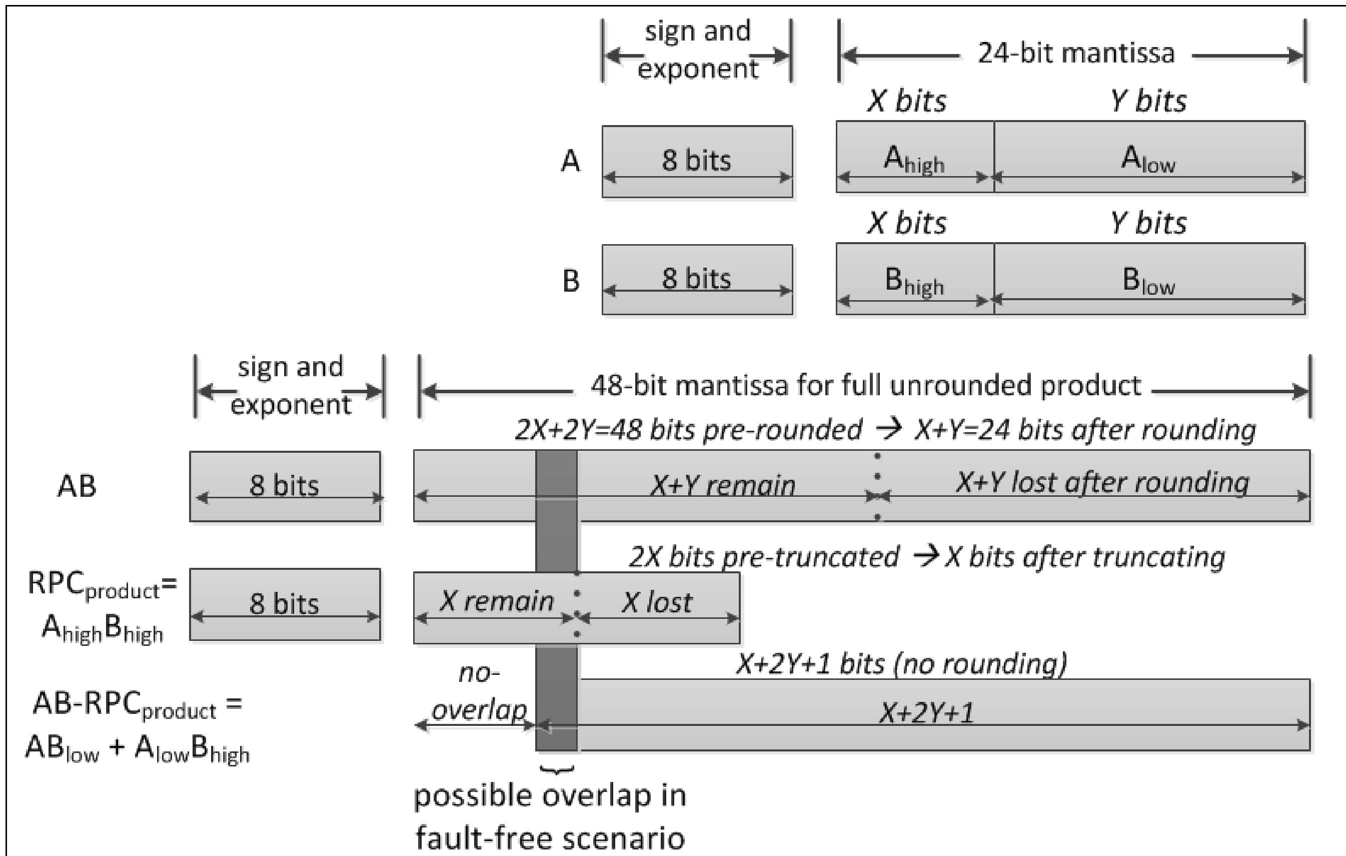

Figure 1. RPC for Multiplication

detects all large errors and some fraction of small errors determined by the checker's mantissa length.

RPC is viable, but thus far it has been applied only to addition. Modern processors—including both CPUs and GPUs—have hardware support for many floating point operations, but numerically sophisticated code generally uses only the hardware adder and multiplier (and constructs other operations in software, if needed). Thus processors must also be able to check floating point multiplication.

III. EXTENDING RPC TO MULTIPLICATION

At a high level, RPC for multiplication is similar to RPC for addition. As with RPC for addition, the checker multiplier has a sign bit and the same number of exponent bits as the multiplier being checked, but it has fewer mantissa bits. Also similar to RPC for addition, the challenge is in determining which discrepancies in results are legitimate (i.e., could occur even in the absence of faults). The primary difference between the RPC adder and the RPC multiplier is the logic that determines how truncation/rounding errors can influence the discrepancy between the higher and reduced precision units in the absence of faults.

In Figure 1, we illustrate RPC for the multiplication of two 32-bit floating point numbers, *A* and *B,* each of which has 1 sign bit, 7 exponent bits, and 24 mantissa bits.[1] We decompose the mantissas of both *A* and *B* into an *X*-bit high portion (A_{high} and B_{high}) and a *Y*-bit low portion (A_{low} and B_{low}), where $X+Y=24$. The RPC product's mantissa is $A_{high}B_{high}$. To detect errors, we compare the RPC product's mantissa, denoted $RPC_{product}$, to the mantissa of the full product, *AB*. The following four equations help to illustrate the relationship between the full mantissa and the RPC product's mantissa.

$$AB = A_{high}B_{high} + A_{high}B_{low} + A_{low}B_{low} + A_{low}B_{high}$$
$$AB = A_{high}B_{high} + AB_{low} + A_{low}B_{high}$$
$$RPC_{product} = A_{high}B_{high}$$
$$AB - RPC_{product} = AB_{low} + A_{low}B_{low}$$

To determine the possible fault-free discrepancies between *AB* and $RPC_{product}$, we must consider their bit lengths to see how they overlap. (Recall that multiplying an *x*-bit number with a *y*-bit number results in a product with *x*+*y* bits.) The pre-rounded mantissa of the full product, *AB,* has 2*X*+2*Y* bits, and the pre-rounded mantissa

[1] A floating point number is stored with a 23-bit mantissa, and the 24th bit is an implicit leading one, but all computations operate on 24-bit mantissas.

of $RPC_{product}$, $A_{high}B_{high}$, has $2X$ bits. The length of $(AB\text{-}RPC_{product})$is $(X+2Y+1)$ bits.

We now compute the overlap between $RPC_{product}$ and $(AB\text{-}RPC_{product})$. Referring to Figure 1, there is a tall, thin, darkly-shaded rectangle that represents this overlap. To compute the width of this rectangle, we first compute the width of the non-overlapping portion, labeled "no-overlap" in the figure.

$$\text{width of no-overlap} = 2X+2Y-(X+2Y+1) = X-1$$
$$\text{width of overlap} = \text{width of truncated } RPC_{product} - \text{width of no-overlap} + 1$$
$$\text{width of overlap} = X - (X-1) + 1 = 2$$

The width of the overlap has what might appear to be an extra "1", but we must account for the discrete point at which two widths meet.

Thus, even in the absence of faults, the RPC product can differ from the upper bits of the full product, because of this 2-bit overlap.

With two bits of overlap, the maximum fault-free difference between the full mantissa and the RPC mantissa is less than three bits. This three-bit discrepancy can manifest only in the lowest three bits of the RPC mantissa. Consequently, our RPC unit checks that the base-10 value of the upper $2X+1$ bits of the full product mantissa is within 7 (i.e., 2^3-1) of the base-10 value of the RPC mantissa. Thus, actual faults that cause errors in low-order bits will not be detected. If a fault occurs, the maximum undetected error is 7 out of a maximum possible mantissa value of 2^X-1. For example, if X=7, then the maximum undetected error is about 5.5%. Intuitively, increasing X decreases the maximum undetected error.

IV. Implementation

We implemented RPC for an open-source VHDL floating point multiplier from opencores.org.[2] As a sanity check, we also (re-)implemented RPC for the adder from opencores.org. Our implementation disables error checking when the operands or the product are non-standard floating point numbers (e.g., infinity, NaN, or denorm). This design decision leads to possibly undetected errors in these rare situations, but it greatly reduces the cost of RPC.

For both the multiplication checker and the addition checker, the checkers use 7-bit mantissas. This 7-bit checker mantissa represents a mid-point in the range of possible mantissa sizes.

V. Error Detection Coverage

The goal of our experimental evaluation is to determine the error detection coverage of RPC. We evaluate both multiplication and, for completeness, addition.

A. Fault Injection Methodology

To evaluate the error detection coverage, we ran an extensive set of fault injection experiments. In each experiment, we forced a single stuck-at fault on a different wire in the floating point unit's flattened netlist. Note that a single stuck-at fault on a wire can often cause multiple faults downstream of the injected fault, due to fan-out, and thus injecting faults at this low level is far more rigorous than injecting faults in a microarchitectural simulator [4]. We injected thousands of single stuck-at faults throughout the multiplier and adder, including in the RPC hardware itself.

We considered injecting transient faults, but a very large fraction of transient faults were masked (i.e., had no impact on execution). Masking occurs for a variety of reasons and is far more common than one might perhaps expect [5]. Transient faults, in particular, are masked with very high probability. To obtain statistically significant data in a tractable amount of time, we injected permanent stuck-at faults, which are less likely to be masked.

B. Results

We randomly chose 1,000 wires in the multiplier and 1,000 wires in the adder. For each chosen wire, we performed 48 experiments in which we injected a fault and performed the relevant computation (multiplication or addition) for a different pair of randomly generated operands. For both the multiplier and adder, approximately 85% of injected faults are masked; this large fraction of masked faults is typical of functional units. It is unlikely that any single fault affects the result of a single computation.

Of those faults that are unmasked in the multiplier and adder, our Reduced Precision Checkers detect 26.5% and 48.8% of the resulting errors, respectively. These detection rates appear quite low, but recall that RPC detects the vast majority of faults with large impact on the result (i.e., faults that cause large errors). For all unmasked/undetected faults, i.e., *silent data corruptions* (SDCs), we plot the percent error in the result for the multiplier and adder in Figure 2 and Figure 3, respectively. The results show that the majority of SDCs have percent errors less than 0.01%, and more than 90% of the SDCs have percent errors less than 1%. Smaller errors are preferable, in general, and they are often easier to tolerate with numerical algorithms, particularly algorithms that iteratively converge on a solution. A small error can often be tolerated (i.e., not affect the final result) at the cost of the algorithm taking more iterations to converge.

These results show that RPC is indeed viable for multiplication. RPC for multiplication, like RPC for addition, detects the vast majority of faults that cause large errors.

VI. Area and Power Overheads

We now evaluate the area and power overheads of our implementation of RPC for multiplication and addition. Recall that our RPC implementation uses 7-bit mantissas for the checkers. The area and power overheads would be less (more) for shorter (longer) checker mantissas.

[2] http://opencores.org/project,fpu100

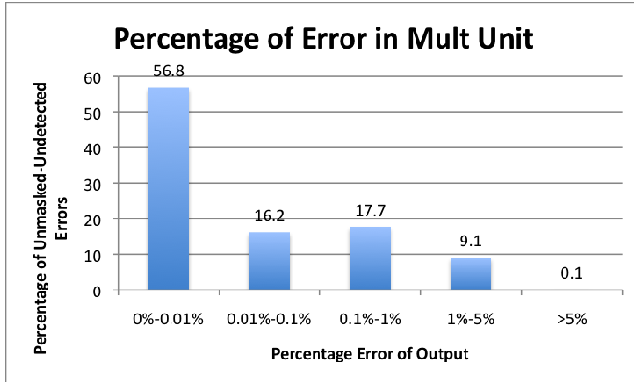

Figure 2. RPC for Multiplication

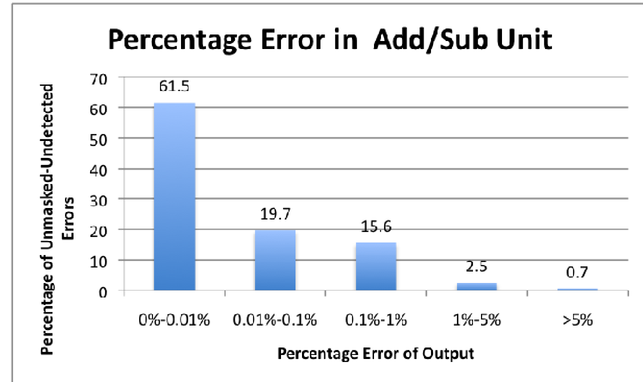

Figure 3. RPC for Addition/Subtraction

A. *Area*

We used Synopsys CAD tools to layout our FPU–consisting of a floating point multiplier and adder—with and without RPC, in 45nm technology [8]. The results show that RPC's area overhead is 17.8%.

B. *Power*

As with the area analysis, we used Synopsys tools to determine the dynamic and static power overheads of RPC. After laying out the circuitry, we obtained the parasitic resistances and capacitances and back-annotated the circuits with them.

To determine the dynamic power, we provided the multiplier and adder with a sequence of 1,000 randomly generated floating point number inputs. The results for this experiment are shown in Table 1.

RPC's total power overhead, including both dynamic and static power, is 28%. Its dynamic power overhead is 35% and its static power overhead is 16%.

Table 1. FPU Power. RPC's percent overhead in parentheses.

	Baseline	RPC (overhead)
Dynamic Power	0.155mW	0.210mW (35%)
Static Power	0.098mW	0.114mW (16%)
Total Power	0.253mW	0.324mW (28%)

VII. Comparing Multiplication to Addition

Having developed RPC for both addition and multiplication, we now present a comparison based on our experiences.

In RPC for addition, the RPC mantissa is within 2 (base-10) of the full-precision mantissa [2]. As explained in Section III, in RPC for multiplication, the RPC mantissa is within 7 (base-10) of the full-precision mantissa. The reduced sensitivity in RPC for multiplication led to about three times more SDCs with magnitudes greater than 1%, compared to RPC for addition (as shown in Figure 2 and Figure 3). Nevertheless, 90.8% of the SDCs in RPC for multiplication were less than 1%, thus showing that RPC for multiplication can still detect the vast majority of large errors.

Despite the error sensitivity differences between RPC for multiplication and addition, the hardware costs of both RPC implementations are quite similar.

VIII. Conclusions

The goal of this research was to evaluate whether reduced precision checking could be successfully extended from addition to multiplication. Our experiences in developing RPC for multiplication revealed it is non-trivial to extend but viable. The experimental results show that RPC for multiplication detects the vast majority of faults that cause large errors and is thus a practical technology for fault tolerant FPUs.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-111-5367. We thank Pat Eibl for his feedback on this work and paper.

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, Dec. 2005.

[2] P. J. Eibl, A. D. Cook, and D. J. Sorin, "Reduced Precision Checking for a Floating Point Adder," in *Proceedings of the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009.

[3] S. Iacobovici, "United States Patent: 7769795 - End-to-end residue-based protection of an execution pipeline that supports floating point operations," U.S. Patent 776979503-Aug-2010.

[4] M.-L. Li, P. Ramachandran, R. U. Karpuzcu, S. K. S. Hari, and S. Adve, "Accurate Microarchitecture-level Fault Modeling for Studying Hardware Faults," in *Proceedings of the Fourteenth International Symposium on High-Performance Computer Architecture*, 2009.

[5] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[6] D. Lipetz and E. Schwarz, "Self Checking in Current Floating-Point Units," in *20th IEEE Symposium on Computer Arithmetic*, 2011, pp. 73–76.

[7] M. Maniatakos, Y. Makris, P. Kudva, and B. Fleischer, "Exponent Monitoring for Low-Cost Concurrent Error Detection in FPU Control Logic," in *IEEE VLSI Test Symposium*, 2011, pp. 235–240.

[8] Nangate Development Team, "Nangate 45nm Open Cell Library." 2012.