Safe Overclocking of Tightly Coupled CGRAs and Processor Arrays using Razor

Alexander Brant, Ameer Abdelhadi, Douglas H.H. Sim, Shao Lin Tang, Michael Xi Yue, and Guy G.F. Lemieux
Dept. of ECE, University of British Columbia, Vancouver, Canada
Email: alexb|ameer|dsim|sltang|xiy|lemieux@ece.ubc.ca

***Abstract*—Overclocking a CPU is a common practice among home-built PC enthusiasts where the CPU is operated at a higher frequency than its speed rating. This practice is unsafe because timing errors cannot be detected by modern CPUs and they can be practically undetectable by the end user. Using a timing speculation technique such as Razor, it is possible to detect timing errors in CPUs. To date, Razor has been shown to correct only unidirectional, feed-forward processor pipelines. In this paper, we safely overclock 2D arrays by extending Razor correction to cover bidirectional communication in a tightly coupled or lockstep fashion. To recover from an error, stall wavefronts are produced which propagate across the device. Multiple errors may arise in close proximity in time and space; if the corresponding stall wavefronts collide, they merge to produce a single unified wavefront, allowing recovery from multiple errors with one stall cycle. We demonstrate the correctness and viability of our approach by constructing a proof-of-concept prototype which runs on a traditional Altera FPGA. Our approach can be applied to custom computing arrays, systolic arrays, CGRAs, and also time-multiplexed FPGAs such as those produced by Tabula. As a result, these devices can be overclocked and safely tolerate dynamic, data-dependent timing errors. Alternatively, instead of overclocking, this same technique can be used to 'undervolt' the power supply and save energy.**

I. Introduction

Tempted by the promise of a faster computer, many PC enthusiasts attempt to 'overclock' their processor. The amount of overclocking can be significant, up to 50% or more, so it often requires boosting the supply voltage as well. To support these enthusiasts, special motherboards and processors exist to give control over clock rates and power supply levels, and success stories fill many user forums.

Two characteristics allow a processor to be overclocked: transistors operate faster than nominally predicted by speed binning, and data-dependent calculations do not exercise the worst-case critical path(s). In both cases, the main problem with operating close to the edge is detecting the first signs of errors in the computation.

Obvious signs of failure are when the processor fails power-on self-test (POST), spontaneously reboots, or presents a corrupt screen. After lowering the clock rate to eliminate these obvious problems, how can one guarantee execution is now perfectly correct?

Safe overclocking requires run-time detection and correction of timing errors. One way to detect timing errors at the circuit level is called Razor [1]–[3]. Shown in Figure 1, Razor augments the critical pipeline registers in a processor with shadow registers. The shadow register captures the data using a delayed clock edge. When the circuit is overclocked, the pipeline register captures data before it is stable, but the shadow register is clocked late enough to capture the correct value. A mismatch between these registers indicates a timing error. Error recovery involves stalling the pipeline to inject the correct data.

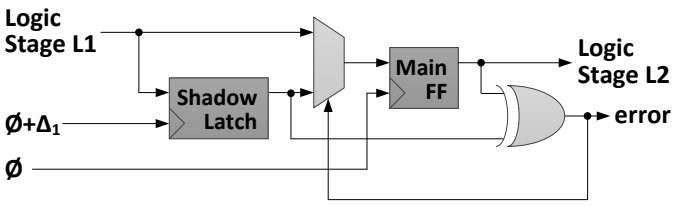

Figure 1. Razor shadow register and error detection

Until now, Razor has only been applied to unidirectional pipelines found in a typical processor. In this paper, we extend the Razor error recovery mechanism to include a 2D array of tightly coupled processing elements with bidirectional communication. By tightly coupled, we mean lockstep, deterministic, prescheduled communication that must guarantee data delivery by a specific clock cycle number, not by elastic methods with data-presence indicators. As a result, this technique can be applied to systolic arrays, CGRAs, and time-multiplexed FPGAs by Tabula.

In extending Razor to bidirectional pipelines, we learn that it can merge stall cycles that are created by multiple errors located in close proximity in time and space. This produces fewer stall cycles than errors. In other words, an overclocked 2D array can scale efficiently, since the number of stall cycles required will grow slower than the number of processing elements.

As a proof-of-concept, we have implemented a system that can be safely overclocked with 63% higher throughput than what is predicted by static timing analysis tools. Our system is a time-multiplexed CGRA implemented on top of an Altera FPGA. We have augmented the processor-to-processor communication paths with Razor timing error detection, and added our own correction mechanism.

II. Background and Previous Work

Razor was introduced as a system to save power in pipelined processors by lowering the supply voltage until timing errors occur [1]. A timing-error detection circuit was introduced to determine when the system was being clocked faster than the supply voltage would support. In 2011, an ARM microprocessor was developed with Razor pipelining [2]. In 2013, Bubble-Razor was introduced to simplify the practical steps of integrating with a real processor design and switching from edge-triggered flip-flops to transparent latches with two-phase clocking [3].

The shadow register, shown in Figure 1, is fed by a delayed clock. To ensure the shadow register holds the *correct* value, the clock period plus the shadow register delay should be greater than the worst-case critical path delay. Similarly, minimum path constraints to the shadow register must prevent 'double-clocking', where data in the upstream flip-flops arrive too early and are clocked into the shadow register on the current cycle rather than the next cycle.

To correct a timing error, the input to the next pipeline stage can be switched to the output of the shadow register. Since there is not enough time left in the clock cycle to accept the shadow value and then compute the required result for the downstream flip-flop, a stall cycle is required. Forward progress is ensured, since the worst-case performance loss is 50% due to a stall occurring every other cycle.

To investigate our Razor extension to 2D arrays, we have implemented a CGRA based upon the MALIBU architecture [4] shown in Figure 2. Each PE contains a full 32-bit integer ALU, including a single-cycle multiplier. ALU results for local re-use by the PE are stored in a 32-bit wide 'R' memory. Results destined for the four neighbouring PEs are written to memories labelled N, S, E, and W; by symmetry, this gives rise to bidirectional 2D communication. The CGRA follows a deterministic schedule of operations. Communication is tighly coupled in that writes to a memory in cycle t can be picked up in the next clock cycle, $t + 1$.

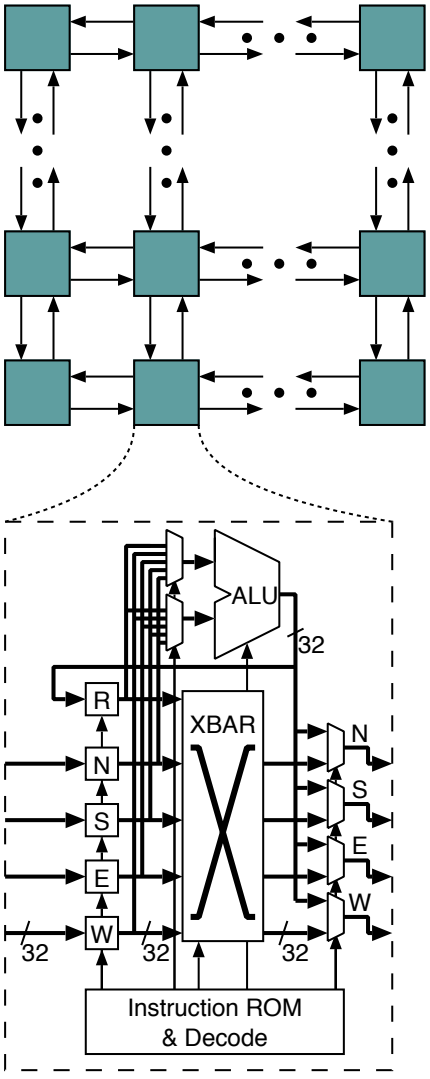

Figure 2. The tiled MALIBU architecture with PE detail.

III. Razor Extension for 2D Arrays

This section details the design of our Razor system for detecting and correcting timing errors in a 2D array.

A. Error Detection and Correction in the 2D Array

Due to the long delay of the multiplier, the critical path in the MALIBU CGRA starts at a memory read port (one of R, N, S, E, or W), goes through the multiplier, and ends at a memory write port. To detect timing errors, Razor shadow registers are placed on the write ports of each memory. Other paths in the system are not critical and left unprotected.[1]

To add Razor-style error detection, the RAMs must be augmented with a shadow register as shown in Figure 3. The RAM is written using the main clock, while the shadow register is clocked after a slight delay from the main clock. To determine if there is a timing error during cycle t, the value to the shadow register is compared to the value written to the RAM. A mismatch, detected in cycle $t + 1$, indicates that erroneous data was written to the RAM.

Hence, cycle $t+1$ is used to correct the error that occurred during cycle t. This involves taking the correct value from the shadow register and writing it to the correct location in the RAM. The correct location from cycle t, the previous cycle, is simply taken from an address register.

In the PE with the detected error, the operation that should have occurred in cycle $t+1$ is instead executed in $t+2$. At the same time during cycle $t+1$, the detected error is propagated in the form of a *stall signal* to the four neighbours.

During cycle $t + 2$, the four neighbours will repeat the operation they were to have executed during cycle $t + 1$.

[1]Given our final speed-up results, we should re-check this assumption.

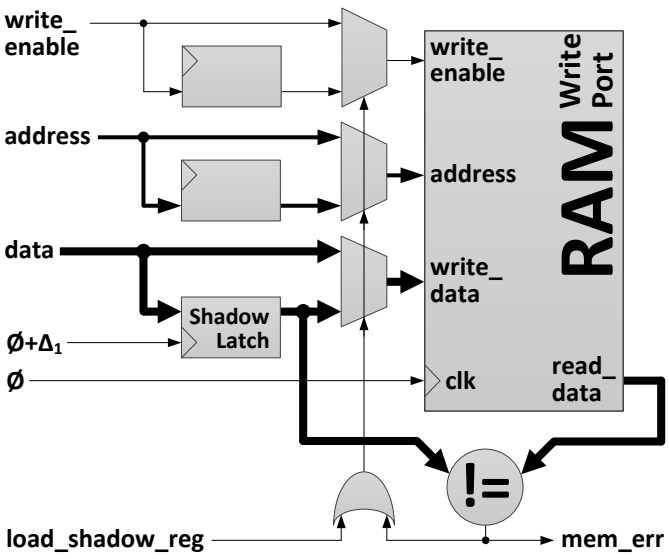

Figure 3. Shadow register applied to Stratix III memory

At the same time, they will also inform their outward neighbours for the next cycle, $t+3$. Ultimately, this produces a *stall wavefront* that propagates outward each cycle until the boundary of the chip is reached.

B. Altera RAM Output

To implement Razor error detection, we exploit a documented feature of the Stratix III block RAMs [5]. Under read-during-write conditions to the same location from the same access port, the block RAM can be configured to output the new data being stored onto the data-out pins right after the clock edge. For this to work correctly, the read-enable signal must also be asserted along with the write-enable signal when a write occurs. This allows us to ensure data is stored correctly with low hardware overhead.

Note that Stratix III cannot output this new data to a different access port if it is attempting to read the same location on the same clock edge as the write. However, the MALIBU CGRA requires this capability to minimize latency. To emulate this, we can augment the RAM shown in Figure 3 with a bypass register [6]. We have implemented this bypass path for MALIBU, but details have been omitted from Figure 3 for clarity.

C. 2D Stall Propagation

The original Razor only corrects timing errors in a unidirectional, feed-forward processor pipeline. In such a system, it is possible for multiple timing errors to arise nearly simultaneously. The errors may occur in the same pipeline stage in nearby clock cycles, or they may occur in nearby pipeline stages at the same clock cycle or similar cycles. To correct these timing errors, the error always propagates forward as a stall; each error creates a stall.

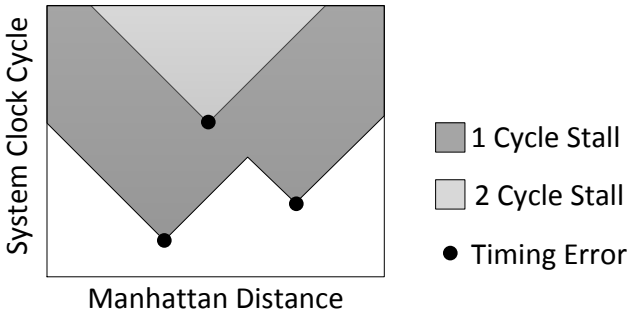

Figure 4. Conceptual graph demonstrating how 2 separate errors join to form one stall region, and how a later error creates a second stall.

In contrast, a bidirectional 1D pipeline must translate a timing error into a stall that propagates in both directions away from the source. For example, consider Figure 4, where a position in the pipeline is represented on the horizontal axis, and the clock cycle number is on the vertical axis. When the first error occurs, a corrective stall wavefront begins propagating in both directions away from the source of the error. When a second error occurs, at a different location but nearby in time, it will create its own two wavefronts. In between the two errors, these wavefronts will meet up and cancel each other out because there is no need to propagate either stall any further. That is, all points in between the two error sources will have stalled for one cycle, which is sufficient to resynchronize with both errors. However, the outer wavefronts continue to expand outwards until they reach the edge of the pipeline.

If a third error occurs at any PE *after* the first or second error has already reached that location, it creates a new stall wavefront that will cause all pipeline stages to execute two cycles behind the original schedule. That new region will expand until it spans the entire pipeline.

In a bidirectional 2D pipeline, two PEs can also stall at nearly the same time and create their own two stall wavefronts. These stalls propagate one block at a time to the four immediate neighbours, creating an expanding diamond-shaped wavefront, as shown in Figure 5. Like the 1D array, two independent errors that occur closely in time may collide and merge. This occurs if the second error occurs before the first error propagates all the way to source of the second error. In this case, only one stall cycle is needed at each stage, so the overall array is stalled only one cycle.

For the MALIBU CGRA, care must be taken to ensure that all memories in a PE hold the correct operands before executing the next operation. When an error is propagating, data entering into the wavefront from the opposite direction must be delayed, also using the shadow register, to prevent that new data from overwriting existing data before the PE is finished using it. This case is explained below.

Consider adjacent PEs A, B and C shown in Figure 6.

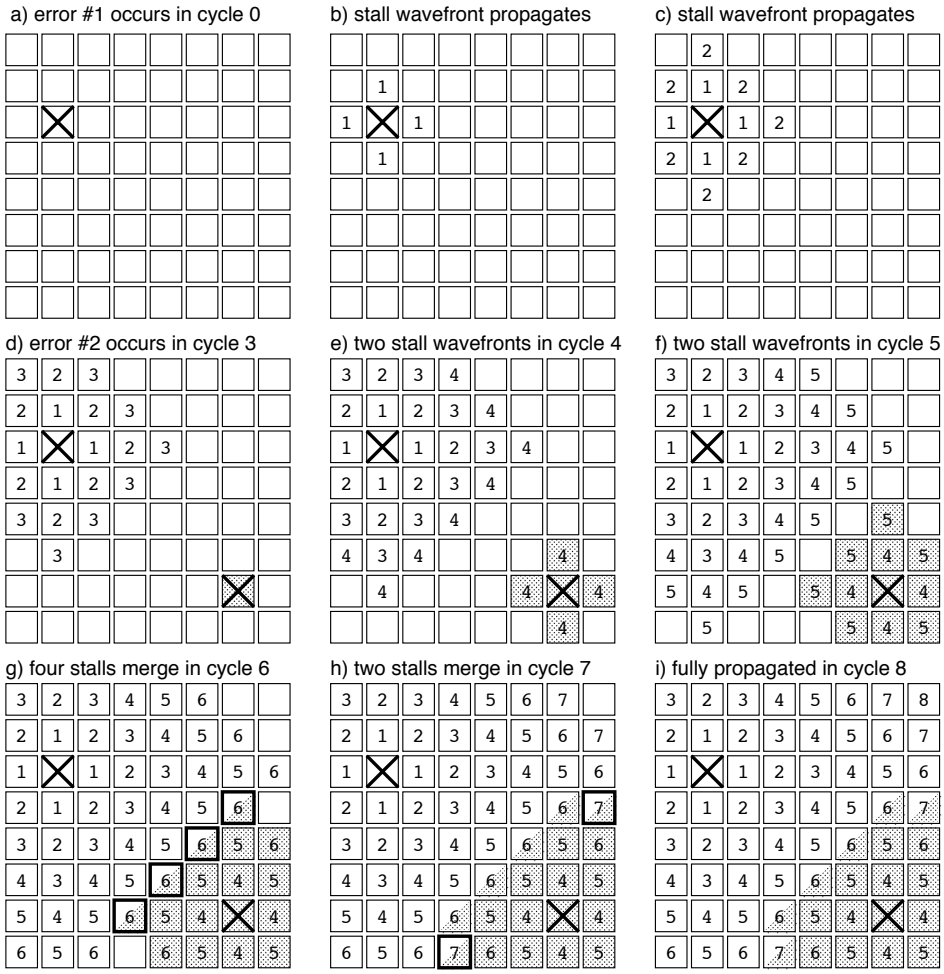

Figure 5. Error propagation in the 2D array, where two errors merge into a single stall cycle. Numbers indicate the clock cycle # of the stall.

All PEs initially perform their own part of instruction Y at cycle t. Suppose A detects an error in cycle t and must re-run Y on cycle $t+1$. During cycle $t+1$, if instruction Y+1 at B is writing to A, it could overwrite a memory location currently in use by A. Instead, A informs B to hold back any writes using 'load_shadow_reg' (abbreviated to load_reg in the figure). When B receives the stall signal at cycle $t+2$, it re-runs instruction Y+1, which involves completing the write back to A. Likewise, C may have written to B during cycle $t+1$ in instruction Y+1 and then write to the same location in B during cycle $t+2$. To ensure the first value is not overwritten before it is used, B sends 'load_shadow_reg' to C, causing C to hold back the write to B during cycle $t+2$. These will be completed in cycle $t+3$.

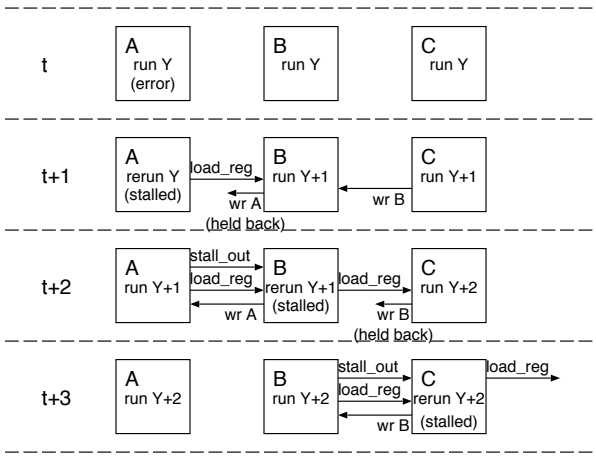

Figure 6. Proper stalling of writes into the wavefront.

At cycle $t+2$, B has the correct data from both A and C produced by instruction Y, A is executing the instruction Y+1 and the new data C computed from instruction Y+1 is now being written from the shadow register to B. B can then execute instruction Y+1 normally. As C is stalling in cycle $t+3$, it is unable to write more data to B. Hence, the data in C's shadow register can be safely written back to B. The stall propagation will continue on until it reaches the edges of the array with each PE having stalled one cycle.

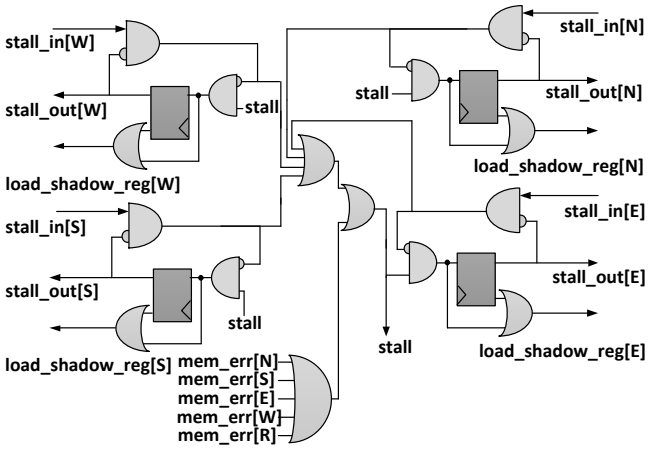

Figure 7. Control logic used to produce the stall signals in each PE.

D. Stall Propagation Logic

The stall propagation logic for each PE is shown in Figure 7. When a memory that belongs to a PE encounters an error, the PE initiates a stall that will spread through the array. The stall will affect the operation of the PE for two cycles: one while it is stalling, and one while its neighbours stall. We first detail the signals used in the stall logic. This is followed by an explanation of how the stalling occurs in the PE that detects the error, and on PEs that stall due to an error from elsewhere. Finally, the merging of separately occurring stall wavefronts is discussed.

The circuit monitors 'mem_err' signals that are asserted when an error is detected at a local memory. It also monitors the 'stall_in' signals that are asserted by the four neighbouring PEs. In response, the circuit provides three outputs: 'stall_out' which are 'stall_in' requests sent to adjacent neighbors, 'load_shadow_reg' which triggers adjacent memories to load their shadow register values, and a local 'stall' signal that holds the instruction counter and inhibits writes from the PE to memories.

A stall begins with an error being detected. Suppose an error occurs on cycle t, so an incorrect value is written to memory. After the shadow register delay, partially into cycle $t+1$, the correct value is latched into the shadow register and compared to the memory contents. Since the values differ, an error has occurred, and the 'mem_err' signal is asserted. Several possible 'mem_err' signals from within the PE (for each of the five different memories – RNSEW for short). are ORed together to create the PE's 'stall' signal.

In the same cycle in which the error is detected ($t+1$), the logic triggers a number of events within the PE. The 'stall' signal prevents the instruction counter from being incremented, and deasserts write enables to any memory from that PE. The 'load_shadow_reg' signal is sent to all memories, which loads the correct data from the previous cycle into the memory, correcting any errors. It also prevents memory writes from the neighbours. These writes must be prevented as they might overwrite data needed by the next instruction. The data from the neighbours is not lost; it will be stored in the shadow registers and written next cycle.

In the cycle after the stall is detected ($t+2$), the PE writes the results of the instruction it tried to execute at $t+1$. The execution happens normally, but the load_shadow_reg is still asserted to the RNSEW memories. This writes any data that neighbouring PEs tried to write previously. No write will be performed to these memories from the neighbours this cycle, as they will be stalled.

PEs that do not encounter an error also need to stall to synchronize with the ones that did encounter an error. On the cycle after an error is detected, the PE with the error asserts a 'stall_out' signal to its immediate neighbours. When a PE receives a valid stall signal, it behaves similarly to a PE that detects a memory error: the instruction counter is not incremented, and no values are written to any memory from that PE. Any writes to the PEs memories from neighbours are also delayed, being stored in the shadow register and performed the following cycle, except from any neighbour(s) that requested the stall. These memories are not loaded from the shadow register, as the neighbour(s) who triggered a stall may need to write a result from the instruction it is catching up on. The 'load_shadow_reg' is also used to trigger the 'stall_out', as the signal is not being sent back to any of the originating PE(s).

When multiple errors occur in the array independently and must be merged, the logic must ensure correct operation. Depending on the distance and timing of the errors, one of two situations will occur: the stalls reach the same tile at once, or two neighbouring tiles stall simultaneously. If the stalls reach the same tiles at once, the tile will stall, but will only propagate the stall forward to tiles that did not send it a stall, so the regions will merge and be synchronized. If two neighbours stall simultaneously, on the cycle following the stall each will receive the 'stall_in' signal from the other, but it can be ignored as they are already synchronized.

E. Performance Tradeoffs

The overall system performance is increased by overclocking, but stall cycles will reduce the effective throughput. The higher the overclocking frequency, the greater the error rate; at one point, the increase in error rate will exceed the clock frequency boost, and throughput will decrease. The error rate will depend on the timing variations in the circuit, the clock frequency, and data-dependent circuit behavior.

Also, any decrease to the clock period must be accompanied by an equal increase in delay to the shadow register. To avoid fast path problems and 'double-clocking', the shortest path delay in the pipeline stage, t_{min}, cannot be longer than the delay to the shadow register. Likewise, the clock period cannot be less than t_{min}. These short path delays are ensured by adding constraints to the Quartus fitter.

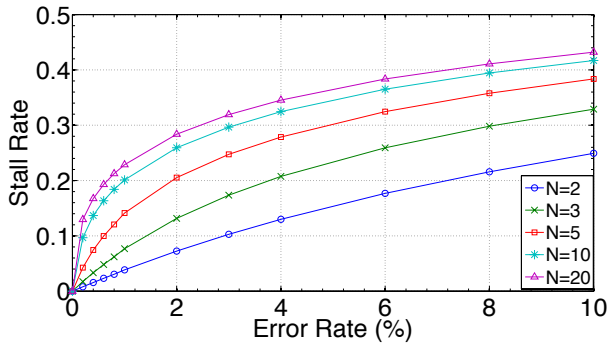

Figure 8. Fraction of performance stalled due to errors ($N \times N$ array).

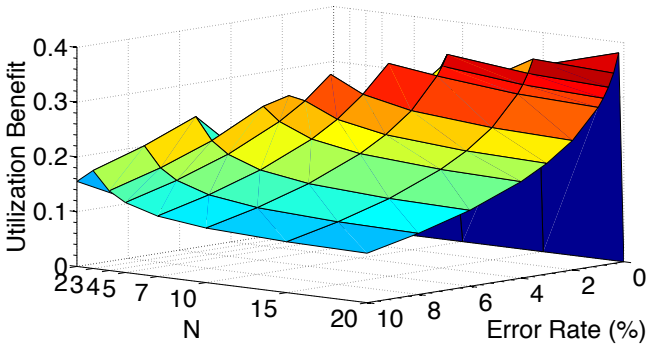

Figure 9. Gain in utilization due to merging of errors ($N \times N$ array).

As t_{min} is increased, the nominal F_{max} of the circuit decreases, so the value must be carefully selected to achieve optimal performance. With a shadow register delay of t_{min}, which is the longest possible without double-clocking, the fastest resulting safe clock period is T_{safe}. Hence, $T_{safe} + t_{min} = T_{init}$, where T_{init} is the initial clock period. Since $T_{safe} \geq t_{min}$, the minimum overclocking period is $T_{safe} \geq T_{init}/2$. For a given t_{min}, the maximum safe frequency is:

$$F_{safe} = \frac{1}{T_{init} - t_{min}} = \frac{F_{init}}{1 - F_{init} \times t_{min}} \leq 2 \times F_{init}.$$

IV. Error Tolerance

When multiple errors occur in close proximity, their stall signals propagate until they collide and merge. This allows multiple errors to be tolerated by a single stall cycle. As a result, more errors can be tolerated than the number of stall cycles that are added to tolerate them. This section analyzes how many errors can be tolerated using a Monte Carlo simulation approach. The array size, $N \times N$, determines the number of PEs that are working together. For each clock cycle, we check for an error with the instruction being executed at each PE by comparing a uniform random value [0,1) to e, the error rate. For each error, we track the propagation of stall wavefronts and merges through simulation.

In Figure 8, we measure stall rate, or the fraction of all possible execution time slots lost to stalls; the worst possible stall rate is 0.5. Stall rates are kept low for small arrays less than 5×5, or error rates less than 1%.

In Figure 9, we measure utilization benefit, or the increase in utilization as a result of merging stall wavefronts; the best possible benefit is less than 0.5, where half of all execution cycles are errors that can be merged into a single stall cycle. Maximum benefit is derived from merging wavefronts at low error rates, but less benefit is realized at high error rates. Virtually no benefit is gained when the error rate approaches zero, because there are no stalls from which to recover.

V. Results

The implementation area and performance of the base MALIBU CGRA (CGRA) and CGRA with timing error detection and correction due to Razor (CGRA-Razor) is presented in this section. Both cases use Verilog and are compiled into an Altera DE3 evaluation board with a Stratix III EP3SL150 FPGA. For CGRA-Razor, one of the FPGA PLL outputs produces the stable, delayed clock needed by the shadow registers.

A. Area

Logic, memory, DSP block, and register usage are reported for both implementations in Table I. The ALM and register usage goes up slightly to implement Razor, but RAM and DSP block usage are identical for both. Note that we have only added Razor protection to the memory-to-memory compute paths. With this area footprint, a 6×7 array of PEs can fit into the largest Stratix III, which has 135,000 ALMs. In particular, the addition of Razor does not reduce the maximum size array that will fit in the device.

B. Experimental Results

The correctness of the Razor stall logic was initially tested using ModelSim. Various benchmarks were compiled to arrays from 1x1 to 4x4 in size. During simulation, errors are introduced randomly during writes to the CGRA-Razor memories. The results compared successfully to a CGRA simulation run without errors.

To keep our place and route times low, we decided to physically implement a 2×2 CGRA array size. Both designs were compiled using Quartus II, and the maximum clock frequency reported by the static timing analyzer (STA) was recorded.

The STA reports a maximum clock speed of 90 MHz for the base CGRA, and 88 MHz for the CGRA-Razor. The 2 MHz speed loss for CGRA-Razor is due to the extra multiplexer in the write path of the memories. Notice the STA result is independent of which benchmark is being run on the CGRA, because it always assumes the worst-case

Table I
RESOURCE USAGE OF CGRA IMPLEMENTATION, PER TILE.

Area	ALMs	Registers	Block Memory Bits	DSP 18-bit Elements
CGRA Tile	2,958	304	15,688	4
CGRA-Razor Tile	3,082	517	15,688	4

Table II
MAXIMUM CLOCK RATE OF EACH CIRCUIT (MHZ) WITH STATIC TIMING ANALYSIS (STA) AND TESTED ON BOARD.

Benchmark	CGRA (STA, safe)	Overclocked CGRA (tested, **unsafe**)	CGRA-Razor (STA, safe)	Overclocked CGRA-Razor (tested, safe)	CGRA-Razor Stall Rate	Effective CGRA-Razor Throughput	CGRA-Razor vs. CGRA Speedup
Random	90	135	88	163	5.0%	155	1.72×
Mean	90	121	88	144	1.3%	142	1.58×
Wang	90	131	88	147	0.71%	146	1.62×
PR	90	136	88	145	1.7%	143	1.59×
average	90.0	130.4	88.0	149.4	2.0%	146.5	1.63×

critical path. Normally, the STA result is the rate at which a system would be clocked.

Since the base CGRA has no way to detect errors when being overclocked, we use a test jig that includes a Nios II processor. The jig can apply random input data to the CGRA and capture the outputs. An initial run at very low speed is done to capture the expected output; to detect errors, this is compared against data captured during a high-speed run.

The test jig is used for both the base CGRA and CGRA-Razor. The clock frequency and Razor shadow register delay are both increased manually until the circuit fails to produce the correct result. Failure is determined by comparison to original results obtained at a lower clock speed. For CGRA-Razor, we continue increasing the clock frequency, and obtain increasing error rates, until we are satisfied that we have exceeded the maximum throughput, and report the maximum throughput result.

In CGRA-Razor, the minimum short path between pipeline stages is the limiting factor for increasing the operating frequency with reliable error detection. We observed a minimum short path delay of 2.5 ns and a maximum clock frequency of 88 MHz reported by STA. This should result in a maximum reliable overclock rate of 112 MHz, or a gain of 25%. At this frequency, no errors are actually observed when running our tests. We can achieve further speedup than the predicted 112 MHz by lengthening the minimum short path. This is done using fitter constraints.

Four benchmarks, which are CGRA configurations, are used to gauge overclocking performance. The first benchmark is composed of a chain of Random operations on input data. The next benchmark calculates the Mean of 256 different input values. The final two benchmarks are the DSP-oriented circuits Wang and PR [7]. The CGRA is loaded into the FPGA, and then each of the four benchmark designs are then loaded into the CGRA and run.

Thousands of iterations of each benchmark were run until the maximum throughput (ie, clock rate for CGRA, clock rate degraded by the stall rate for CGRA-Razor) is determined. The base CGRA frequency is swept initially from the STA result, ending at the point where incorrect outputs are produced. The CGRA-Razor frequency is similarly swept, but when failure occurs the delay to the shadow register is increased. Increases in frequency and shadow register delay are performed until the design will not run successfully regardless of the shadow register delay. Only the result with the maximum throughput is recorded.

Results for this experiment are presented in Table II. The results show the base CGRA can be overclocked to roughly 130 MHz, and CGRA-Razor can be overclocked to roughly 150 MHz. Not shown in the table are the shadow register delays, which range from 0.5ns to 2ns for these benchmarks.

The base CGRA achieves a lower overclock rate than CGRA-Razor because the base CGRA has no way to recover from timing errors; the first error determines the overclocking limit. More importantly, overclocking the base CGRA by any amount is unsafe and should never be relied upon. (Our test jig is performing error detection, but it cannot be used in a real on-line system.)

In contrast, the CGRA-Razor design is able to run at a higher clock rate because it does not fail on the first error. Instead, it corrects each error by introducing a stall. The table shows not only the overclocked frequency, but also the overall stall rate. The stall rate is used to degrade the clock frequency, producing the actual throughput. For example, after applying the average 2% stall rate to 149.4 MHz performance, the average effective throughput is 146.5 MHz. The CGRA-Razor overclocked throughput is, on average, 1.62 times higher than the base CGRA (STA), 1.66 times higher than CGRA-Razor (STA), and 1.12 times higher than the unreliable overclocked CGRA.

C. Error Probability Observations

We've made a number of observations regarding the frequency of errors arising in CGRA-Razor.

From analysis of the Quartus II Static Timing Analyzer, it is clear that a few instructions will often go through the critical path, as the longest register-to-register delays run

through the multiplier. The probability of the path failing is also determined by the data values, as higher bits of data often take longer to propagate to the registers. This was observed in our tests; when inputs vectors were set to all ones, the worst case error count was often observed.

The Random benchmark is composed of several random operations, using most possible paths through the ALU. This leads to a wide range of delays to the shadow register, including a number of high delays. Accordingly, this allows the frequency to be increased a significant amount before a large number of errors are detected.

Both Wang and PR use a number of multiplication and addition operations. Since most operations occur on the same paths, the frequency can only be increased by a small amount before many errors are noticed.

The Mean benchmark contains a large number of additions ending with a final shift. The shift goes through the multiplier, so it is much slower than the additions. Typically it was this single shift that produces the timing errors.

VI. Conclusions

We have presented an extension to the Razor system which can be applied to detecting and correcting timing errors in tightly coupled 2D processor arrays. In addition, the error recovery process can merge multiple stalls created by multiple errors into just a single stall cycle if they occur in close proximity. This provides promise of scalability, since the number of stalls will grow more slowly than the number of timing errors detected in large arrays.

By overclocking, we were able to speed up an array of processors an average of 1.45×, from 90 MHz to 130.4 MHz, over that predicted by static timing analysis. The speed limit is determined by occurrance of the first timing error that is detected by our testing jig. Overclocking this array without Razor is unsafe because it has no method of detecting errors that might arise.

Using our Razor technique on the 2D array, we can increase overall throughput by an additional 12% beyond the overclocked base CGRA. The additional overclocking is possible because Razor detects and corrects multiple errors. As the clock rate goes up, however, the error rate will also increase; eventually, the rate of stalls is so high that throughput goes down. At the point where maximum throughput is reached for each benchmark, the average speedup in clock speed is 1.66×, or 149.4 MHz. After accounting for stalls, the overall speedup in throughput is 1.63×. At maximum throughput, the stall rate introduced by timing error correction is modest, in the range of 1–5%.

Most importantly, since timing errors are detected and corrected, the proposed overclocking method *safely* achieves the 63% throughput increase. Overclocking the base CGRA achieves a lower throughput and is unsafe.

The original Razor system can be used to save power by undervolting the device, causing the transistors to run more slowly. This use case is also compatible with our design.

While our CGRA system is specific, tightly coupled processor arrays or systolic arrays that rely upon similar one-cycle communication between Manhattan neighbours, either communicating directly through registers or through memory, can expect to achieve similar results. Our technique is only of benefit to tightly coupled arrays; loosely-coupled arrays that have adaptive flow control or use data-presence indicators do not need to propagate the error across the chip; they can simply stall locally until the result is correct.

Also, time-multiplexed systems such as Tabula FPGAs can benefit from this technique. Tabula supports up to 8 contexts or time-slices per user clock cycle. By reserving 1 or 2 spare contexts for error recovery (stalls), a faster system clock can be applied. When an error is detected using a Razor latch or flip-flop, a stall signal would be propagated to downstream neighbouring logic. This would delay a context to the next time-slice, and use up a spare context. As long as these spare contexts are available for error recovery, the same technique can be used to safely overclock Tabula FPGAs.

VII. Acknowledgments

The authors would like to thank Altera for donations of hardware and software, as well as NSERC for funding.

References

[1] D. Ernst, N.S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003, pp. 7–18.

[2] D. Bull, S. Das, K. Shivashankar, G.S. Dasika, K. Flautner, and D. Blaauw, "A power-efficient 32 bit ARM processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation," *IEEE J. of Solid-State Circuits*, vol. 46, no. 1, pp. 18–31, Jan. 2011.

[3] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D.M. Harris, and D. Blawww, "Bubble Razor: Eliminating timing margins in an ARM Cortex-M3 processor in 45nm CMOS using architecturally independent error detection and correction," *IEEE J. of Solid-State Circuits*, vol. 48, no. 1, pp. 66–81, Jan. 2013.

[4] D. Grant, C. Wang, and G. Lemieux, "A CAD framework for MALIBU: an FPGA with time-multiplexed coarse-grained elements," in *International Symposium on Field-Programmable Gate Arrays*, 2011, pp. 123–132.

[5] Altera Corporation, *Stratix III Device Handbook*, 10.0 edition, Mar. 2011.

[6] A. Brant, A. Abdelhadi, A. Severance, and G. Lemieux, "Pipeline frequency boosting: Hiding dual-ported block RAM latency using intentional clock skew," in *Field-Programmable Technology*, December 2012.

[7] M.B. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput," *IEEE Transactions on VLSI Systems*, vol. 3, no. 1, pp. 2 –19, Mar. 1995.