# Trifecta: A Nonspeculative Scheme to Exploit Common, Data-Dependent Subcritical Paths

Patrick Ndai, *Student Member, IEEE*, Nauman Rafique, Mithuna Thottethodi, Swaroop Ghosh, Swarup Bhunia, *Member, IEEE*, and Kaushik Roy, *Fellow, IEEE*

***Abstract*—Pipelined processor cores are conventionally designed to accommodate the critical paths in the critical pipeline stage(s) in a single clock cycle, to ensure correctness. Such conservative design is wasteful in many cases since critical paths are rarely exercised. Thus, configuring the pipeline to operate correctly for rarely used critical paths targets the uncommon case instead of optimizing for the common case. In this study, we describe `Trifecta`—an architectural technique that completes common-case, subcritical path operations in a single cycle but uses two cycles when the critical path is exercised. This increases slack for both single- and two-cycle operations and offers a unique advantage under process variation. In contrast with existing mechanisms that trade power or performance for yield, `Trifecta` improves the yield while preserving performance and power. We applied this technique to the critical pipeline stages of a superscalar out-of-order (OoO) and a single issue in-order processor, namely instruction issue and execute, respectively. Our experiments show that the rare two-cycle operations result in a small decrease (5% for integer and 2% for floating-point benchmarks of SPEC2000) in instructions per cycle. However, the increased delay slack causes an improvement in yield-adjusted-throughput by 20% (12.7%) for an in-order (InO) processor configuration.**

***Index Terms*—Architecture, speculative, variation.**

## I. Introduction

THE frequency and voltage at which a pipelined processor operates are typically chosen such that the delay through the critical pipeline stage(s) can be accommodated in a clock cycle. Because the delay through any logic stage is not a single scalar value but a delay distribution over all possible inputs, designs typically accommodate the worst-case delay path (i.e., the critical path) in the clock period. This choice of accommodating worst-case delay leads to: 1) a reduction in instruction throughput due to the slower frequencies or 2) an increase in power due to higher operating voltage. Further, it fails to exploit the observation that critical logic paths are rarely exercised and that the data-dependent paths, which are commonly exercised, are (or can be made) subcritical.

Unfortunately, the three existing approaches to exploit data-dependent subcritical paths have some drawbacks, which we wish to avoid. The first approach is asynchronous computation, in which operation completion automatically triggers subsequent computation without waiting for a clock edge. This option is extremely complicated to design/validate for complex processors. As such, we exclude this option from further consideration in this paper. The second approach, exemplified by circuit-level speculation (CLS) [15], is to speculate that the delay will be subcritical and use a shorter clock cycle than the critical path. The speculative operation will have to be verified by comparing the results to a nonspeculative computation. If an error is detected, a rollback/recovery is necessary to ensure correct completion of the speculative operation (and any dependent operations that used the speculative values). The two major drawbacks of this approach are: 1) the increased area- and power-overheads to compute the nonspeculative result and 2) the cost of rollback/recovery, especially if dependent instructions have consumed the speculative values. Finally, we also consider Razor [8], which is a general post-silicon process adjustment technique that can reduce voltage margins by dynamic detection and correction of delay errors. When data-dependent subcritical paths are common, Razor effectively lowers the voltage until the subcritical paths are barely accommodated in the clock cycle. Razor's unique way of exploitation of data-dependent subcritical paths can also be termed as speculation with rollback and recovery. The chief drawback of Razor is the tight constraints it imposes on the allowable delay errors. We expand on this in the related study later in Section II.

This paper describes a nonspeculative technique—`Trifecta`—that exploits data-dependent subcritical paths without the drawbacks of asynchrony and/or rollback/recovery. `Trifecta` uses a clock-cycle period that is smaller than the critical path delay. In the common case, the subcritical paths are accommodated in a single clock cycle. Operations that cannot be guaranteed to complete in one cycle are allowed to complete over two cycles. `Trifecta` has two key components. First, it uses additional logic to *conservatively* and *quickly* detect operations which complete in one cycle by examining a subset of input bits at runtime. The detection is conservative because the logic delay for any detected input is guaranteed to complete within one cycle (while some short-latency operations can be flagged as two cycles). The detection occurs in parallel with the logic delay and does not extend the critical path. For integer execution units and selection logic, we demonstrate adequately

Manuscript received February 19, 2008; revised July 16, 2008. First published April 28, 2009; current version published December 23, 2009. The work of P. Ndai, S. Ghosh, and K. Roy was supported by the Focused Center Research Program and the work of N. Rafique and M. Thottethodi was supported in part by NSF Award CCF–0702612.

P. Ndai, M. Thottethodi, S. Ghosh, and K. Roy are with the Electrical and Computer Engineering Department, Purdue University, West Lafayette, IN 47907 USA (e-mail: pndai@purdue.edu).

N. Rafique was with the Electrical and Computer Engineering Department, Purdue University, West Lafayette, IN 47907 USA. He is now with Google, San Francisco, CA 94105 USA.

S. Bhunia is with the Electrical and Computer Engineering Department, Case Western Reserve University, Cleveland, OH 44106 USA.

Digital Object Identifier 10.1109/TVLSI.2008.2007491

fast detection can be achieved leaving ample room for corrective action even with an aggressive 10 FO4 clock cycle.

Second, Trifecta must stall the relevant parts of the pipeline to ensure that two-cycle operation is correctly implemented. For correct operation, Trifecta must ensure that: 1) the inputs do not change for two cycles to guarantee correct operation completion; 2) the intermediate (and potentially incorrect) results available after one cycle are not used; and 3) under special circumstances, if the dependents of two-cycle operation are issued prematurely, they are squashed and reissued.

In this paper, we show specific instances of applying Trifecta to the integer execution unit [specifically, a quaternary tree adder [24] (QTA) in the arithmetic logic unit (ALU)] in in-order (InO) processors; and issue logic (specifically selection logic) in out-of-order (OoO) processors as they have been shown to be the critical pipeline stages for the respective processor configurations [8], [18]. For the execution unit, our detection mechanism determines a one-cycle operation by examining a central block of bits to see if it propagates a carry. If it does not, a subcritical operation is guaranteed. For the selection logic, we exploit the observation that older instructions are more likely to be ready than younger instructions. By optimizing the selection logic implementation to grant requests of older requesters expeditiously, we ensure that the selection of older instructions—the common case—occurs in a single cycle. The one-cycle detection logic checks if there is any request from the first $n$ older instructions. If not, it assumes a two-cycle operation. For sake of completeness, we also demonstrate how our scheme can be extended to the integer execution unit in OoO processors. Even though Trifecta works completely nonspeculatively for critical pipeline stages, application of Trifecta to the execution stage of OoO processor can result in squash/replay of immediate dependent instructions. This happens because OoO processors require the latency of operations to be known beforehand for back-to-back issue of dependent instructions. Trifecta dynamically detects the latency of some operations to be two cycle and can cause dependents to be issued prematurely. Note that this happens only for direct dependents which are to be issued in a cycle immediately following the two-cycle operation.

While the earlier discussion assumes deterministic logic delays, Trifecta offers a unique advantage when statistical delay variation due to manufacturing process variation (PV) (more specifically die-to-die (D2D) variation) is considered [22]. Trifecta can operate at nominal voltage and frequency and achieve high yield as long as one-cycle operations can complete within the nominal clock period. For the rare case where the delay cannot be accommodated in one cycle, Trifecta ensures correct operation over two cycles instead of suffering timing errors. Because the instructions per cycle (IPC) degradation is small, there is little performance penalty. As such, Trifecta offers a key improvement over other approaches to tolerating process variations (PVs) in that it increases yield without degrading power (operating voltage) or performance (instruction throughput).[1] Trifecta achieves up to 20% improvement (depending on the frequency of the fastest bin) in yield-adjusted-throughput (YAT) over the base case, with speed binning applied to both the cases.

[1]Hence, the name Trifecta.

*Contributions:* In summary, the three key contributions of this paper are as follows.

1) We develop Trifecta, a nonspeculative technique to detect and exploit fast operations—operations that exercise data-dependent subcritical paths at run time.
2) We demonstrate how Trifecta can be applied to the election and the integer execution logic in InO and OoO processors. Compared to an improved version of CLS, our technique achieves similar IPC on SPEC2000 benchmark suite (better by 2.4% and 7% for integer and floating-point benchmarks, respectively, in InO processors; and 2% for both integer and floating-point benchmarks in OoO processors) with significantly reduced area- and power-overheads.
3) We demonstrate that in contrast to existing techniques to address PV that enhance yield by sacrificing power OR performance, Trifecta improves yield while maintaining power (due to operation at nominal voltage) and performance (due to nominal frequency and negligible IPC degradation).

The rest of this paper is organized as follows. Section II discusses related work. Section III provides a general overview of our technique and presents a uniform framework to discuss specific implementations of Trifecta. Sections IV and V describe the specifics of applying our technique to the integer execution unit and the selection logic, respectively. Section VI describes the impact of PV on Trifecta. Section VII describes the evaluation methodology. We present simulation results in Section VIII. Section IX summarizes and concludes this paper.

## II. Related Work

In this section, we explain the previous work done on CLS and adder optimizations.

Ghosh *et al.* have proposed a design/synthesis methodology for data-dependent varicycle operation, which optimizes the delay of subcritical paths by resorting to transistor sizing [11]. In contrast, our techniques work at the architecture/circuit level without the need for transistor sizing to create subcritical paths. Ghosh *et al.* have further applied their technique to pipelines in later work [10]. However, they only consider simple pipelines of random logic without any pipeline loops. Recently, Brooks *et al.* [14] proposed a technique that enhances yield at the nominal voltage and frequency by reconfiguring architectural blocks to consume variable latencies. Their technique is a test-time reconfiguration technique and not a data-dependent (i.e., runtime) technique, like Trifecta. Also, both the earlier techniques focus on improving the yield and frequency of logic circuits under PV. In contrast, our technique is useful even if there is absolutely no PV. However, its benefit is magnified in the presence of PV.

Chen *et al.* [7] developed the cascaded carry-select adder ($C^2SA$). The adder nonspeculatively detects the input conditions required for the carry to propagate through the middle set of bits. Whenever the carry propagates through the middle bits, the operation is termed as a long-latency operation and the addition takes two cycles. Otherwise, the operation takes a single cycle. Our two-cycle detection logic is similar to the one used by Chen *et al.* [7]. However, Chen *et al.* studied the adder in isolation,

while we did our analysis by merging the adder in a pipeline and studied its behavior with real benchmarks. Moreover, they did not analyze the behavior of the adder circuit under PV. Wolrich *et al.* [23] proposed an adder that detected all operations of carry length of 19 or greater. It generated a stutter signal to extend the clock if a long carry chain was detected. But the circuit used for generating stutter signal is significantly more complicated than that of Chen *et al.* [7].

Ernst *et al.* proposed Razor [8], which is a circuit-level timing speculation technique for enabling low-power operation. Razor employs feedback-based dynamic control to adjust the voltage levels in such a way that error rates due to critical path timing failures are kept at an acceptable level. The errors are detected using "shadow" latches, which latch the delayed (correct) results from the pipeline stage. In case of errors, the value from "shadow" latches is used. Due to this dynamic feedback, Razor is capable of correcting dynamic errors, such as those associated with variations in voltage and temperature. `Trifecta` is different from Razor in the sense that `Trifecta` detects long-latency operations and provides them two cycles to finish; Razor waits until a long-latency operation actually results in an error and employs error detection and recovery. Moreover, contrary to `Trifecta`, Razor imposes strict constraints on the timing of the delayed clock that drives "shadow" latches, and requires additional buffers to be inserted if there are short paths in circuit. Another key difference is that `Trifecta`, being pre-silicon technique, cannot handle dynamic errors.

Liu *et al.* [15] proposed a CLS technique to reduce the logic delay of a pipeline stage. They proposed a simpler version of the logic to implement the pipeline stage. The simpler version is expected to give correct results most of the time. They also included the complete version of the logic in the pipeline, which concurrently performs the computation. Once an operation is finished on a complete version, its result are matched against the speculative version. If the results fail to match, the operation is issued again on the complete version. All the instructions that used the results of the speculative version are reissued. Liu *et al.* had a large area overhead because they required the complete versions of each pipeline stage. Nevertheless, we present the IPC numbers of their scheme in Section VIII for comparison.

## III. `Trifecta` Overview

`Trifecta` is a technique that partitions inputs to any logic stage into two sets such that all inputs in one set complete the operation in one cycle and inputs in the other set complete in two cycles. We describe `Trifecta` in terms of the following set of four questions. The set of questions serves as a common framework to discuss implementations of `Trifecta` as applied to specific pipeline stages in Sections IV and V.

1) *Which pipeline stage should* `Trifecta` *be applied to*?
   `Trifecta` must be targeted to the clock-critical pipeline stage, since there is no performance advantage if it is applied to a pipeline stage that does not determine the frequency. Note that such clock-determining stages are usually "singleton pipeline loops," which feed the output of the pipeline stage back to the input for the next instruction. Pipelining singleton loops prevents back-to-back issue of dependent instructions and thus cannot be accomplished without significant loss of performance [4]. In InO pipelines, the EX stage (with bypassing) is a singleton loop and forms a critical stage [8]. In a dynamically scheduled, OoO superscalar processor, rename logic, issue logic, and EX stage are the critical singleton pipeline loops [18], [4], [13]. However, we omit the rename logic from consideration because previous studies show that it is not clock-critical [18]. All other pipeline stages can be pipelined without significant penalty since they do not affect back-to-back issue of dependent instructions. For example, Hrishikesh *et al.* have demonstrated that nonsingleton loops in predictable outer loops (e.g., branch resolution loop) can be pipelined without any significant performance degradation [13]. As such, we apply `Trifecta` to those two stages/processor configurations.
2) *Are subcritical paths common? If not, can they be made common*?
   Just as caches are beneficial only when there is locality, `Trifecta` is beneficial only when subcritical delays are the common case. Therefore, we do not make a claim that `Trifecta` can be applied to all pipeline stages. However, in some cases, critical paths are naturally rare (i.e., under random inputs) compared to subcritical paths because they are exercised by a very small subset of inputs. For example, the critical path in a ripple-carry adder is exercised only when the carry ripples through the entire width of the word. As long as those input values are not frequent in the workload, common-case subcritical path operation can be assumed. A more interesting case occurs when some input combinations are more popular than others but they exercise a critical path. In this case, it may be possible to redesign the logic to reduce the delay suffered by the frequently occurring inputs. We demonstrate an example of this approach in Section V.
3) *How does* `Trifecta` *detect one-/two-cycle operation*?
   If there exists a subset of inputs for which the common-case operation is guaranteed to never exceed one-cycle (as discussed earlier), then the detection logic simply has to detect those input combinations. Ideally, the input combinations will be a function of a small subset of input bits and can be evaluated with little delay.
4) *How is two-cycle operation implemented*?
   In general, two-cycle operation is implemented by ensuring that: 1) inputs do not change after the first clock cycle and 2) the (potentially incorrect) value available after one cycle is not used by any subsequent logic stage.

In Sections IV and V, we use the above framework to describe how `Trifecta` is applied to the integer adder (in InO and OoO pipelines) and instruction selection logic (in an OoO processor pipeline), respectively.

## IV. Integer Execution Units

### A. Base Configuration

The clock-critical paths in integer ALUs are almost always associated with the adder core within the ALU. As such, we limit the discussion in this section to the adder core within the

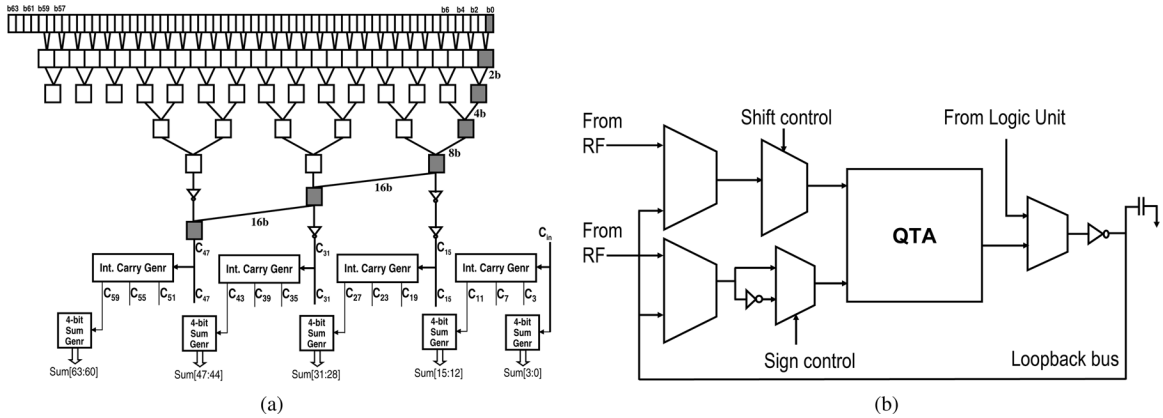


Fig. 1. The execution unit.

ALU. We model the entire ALU when it comes to delay evaluation. This includes input/output muxes and the loopback bus, as shown in Fig. 1(b).

In high-performance microprocessors, 64-bit adders are implemented using carry look-ahead (CLA-based) adders. The Kogge–Stone adder forms the basis of CLA adders, and it implements the full carry-merge operation at each bit. The Han–Carlson adder performs only alternate (either odd or even) carry-merge operations, resulting in a circuit that is about 40% the size of the Kogge–Stone adder [16]. The QTA [24] performs the computation with significantly less overhead than the Han–Carlson. It combines the speed of the CLA adder with the lower complexity of the CSA. It implements a sparse tree adder and generates every 16th carry-out, i.e., C15, C31, and C47, as shown in Fig. 1(a). The generation of C47 forms the critical path.

The generation of C3, C7, C11, C19, . . ., C59 is implemented in parallel with C15, C31, and C47 in a carry-merge tree assuming a carry-in of 0 or 1. Cin, C15, C31, and C47 are used to select between the outputs of these smaller carry-merge trees as shown in Fig. 1(a). Finally, the sum for every four bits S[3:0], S[7:4], . . ., S[63:60] is computed in parallel assuming that the carry-in is both a 1 and a 0. The carry bits Cin, C3, C7, C11, C15, . . ., C59 are used to select the final sum. Relative to the Han–Carlson adder, the QTA results in an adder implementation that reduces the fan-outs of the carry-merge tree by 50% and reduces the wiring density by 80% [16]. This results in an adder implementation that is faster, smaller, and more power efficient than the Han–Carlson [17], [16], [24]. The rest of the discussion explains how the proposed scheme is applied to a 64-bit QTA that was implemented as the adder core in the integer ALU.

### B. Input Distribution

To ascertain that the data-dependent delays are indeed subcritical in the common case, we exploit the following two observations. First, we exploit the observation that only a small subset of input combinations exercise the critical path in the specific case of the adder. The above observation is intuitive for the ripple-carry adder since the critical path is exercised only when the carry is propagated through every bit-position. However, similar intuition can be applied with minor modifications wherever carry chains exist and it has been used in the context of other adders elsewhere [23], [7].

Second, circuit researchers have shown that long-latency adds can be detected by observing a subset of centrally located input bits [7]. If any one of the central bit positions generates (1 and 1) or sinks (0 and 0) a carry, we know that the long carry chain is broken. Generalizing the observation, if we consider $k$ central bits, the probability that the carry from the lower half of the computation will affect the upper half will be $1/2^k$ for random inputs. Thus, in the common case (assuming random inputs), the carry chain is indeed broken, resulting in subcritical path operations most of the time.

However, since application values are not random, we must explicitly demonstrate that two-cycle operations are rare in real programs. Simulations with SPEC2000 benchmark suite reveal that: 1) two-cycle operations are indeed rare (fewer than 4%) and 2) two-cycle operations invariably occur because of adding two numbers that are of small magnitude and dissimilar sign. In 2's complement representation, small numbers have either 1's or 0's due to sign extension. It is these 1's and 0's that form a carry-propagation chain leading to long-latency adder operations. We discuss this in detail in Section VIII-A.

### C. Two-Cycle Detection

As previously mentioned and as indicated in Fig. 1(a), the critical path of the QTA is the generation of C47. Our detection logic uses the eight central bit-positions from position 32 to position 39. If there is a single pair of bits in bit positions 32:39 that are identical (11 or 00), the carry propagation is broken. In this case, the generation of C47 does not depend on C31. In fact, C47 is generated at the same time as C15, and the delay of this operation becomes the time required to generate C31, which our detection logic flags as a one-cycle operation. The remaining inputs that exercise the full critical path (i.e., the delay it takes to generate C47) use a two-cycle operation.

Note that unlike the ripple-carry adder where the delay of the subcritical paths is approximately half of the overall critical path delay, the delay of the critical path in the QTA is only approximately 14%–20% longer than that of the longest subcritical path. This is because the shorter paths only reduce one level of logic in the tree as shown in Fig. 1(a). Ghosh *et al.* have proposed special sizing algorithms [11], which can be used to

increase the difference between the critical and subcritical operations given an area constraint. For this study, we restricted ourselves to full static logic and we did not employ any specialized sizing algorithms.

### D. Implementation

Since the adder is the critical stage in a single issue InO processor, a simple pipeline stall (either by clock gating or other write-disabling means) can ensure that the correct two-cycle operation occurs. It simultaneously ensures that inputs do not change and that the potentially erroneous results are not used. We have experimentally verified that it is feasible to stall an OoO processor (Section V), thus stalling for InO processor should not be an issue.

Application of the same technique to OoO processors is slightly more complicated. OoO processors issue immediate dependents of operations based on the known latency of the operations. For example, if an `add` instruction is producing a result that is used by a `load` instruction immediately following it, the `load` instruction would be scheduled in the cycle immediately following `add` instruction, since the `add` operation is known to complete in one cycle. However, if we have applied `Trifecta` to the adder, the `add` operation might take two cycles. Even though the `add` operation would be detected to take two cycles soon after it is issued, it would not be possible to stall the `load` instruction as it would be already woken up. Our solution is to squash those immediate dependent instructions and reissue them. This requires no significant hardware modifications, since a similar mechanism is already used for handling cache misses in OoO processors.

## V. Selection Logic

In this section, we give the details of the application of `Trifecta` to the instruction selection logic in an OoO processor. We start by explaining the base configuration. Then, we point out the observations that we made to apply our technique, followed by a description of two-cycle detection. We conclude this section by giving a detailed description of our implementation.

### A. Base Configuration

In an OoO processor, instruction selection logic is required to select instructions from a pool of ready instructions. We assume a split issue window, i.e., instructions go to different issue windows based on the type of functional units they use. Instructions wait in the issue window for their operands to become ready. When all the operands of an instruction are ready, it raises a request signal. The selection logic uses an arbiter tree to select instructions for issue from all the instructions that have raised their request signal. If an instruction is selected, its corresponding grant signal is asserted. Fig. 2(a) reproduces the arbiter tree for the selection logic as described by Palacharla *et al.* [18]. We assume that each arbiter cell gives a higher priority to instructions on the left. Moreover, in each cycle, instructions are compacted in the issue window to the left, making space for newer instructions [9]. Thus, our selection policy is oldest-first. Others have observed that variations in selection policy do not cause any significant changes in performance [6]. In order to handle multiple functional units of the same type, the arbiters are stacked

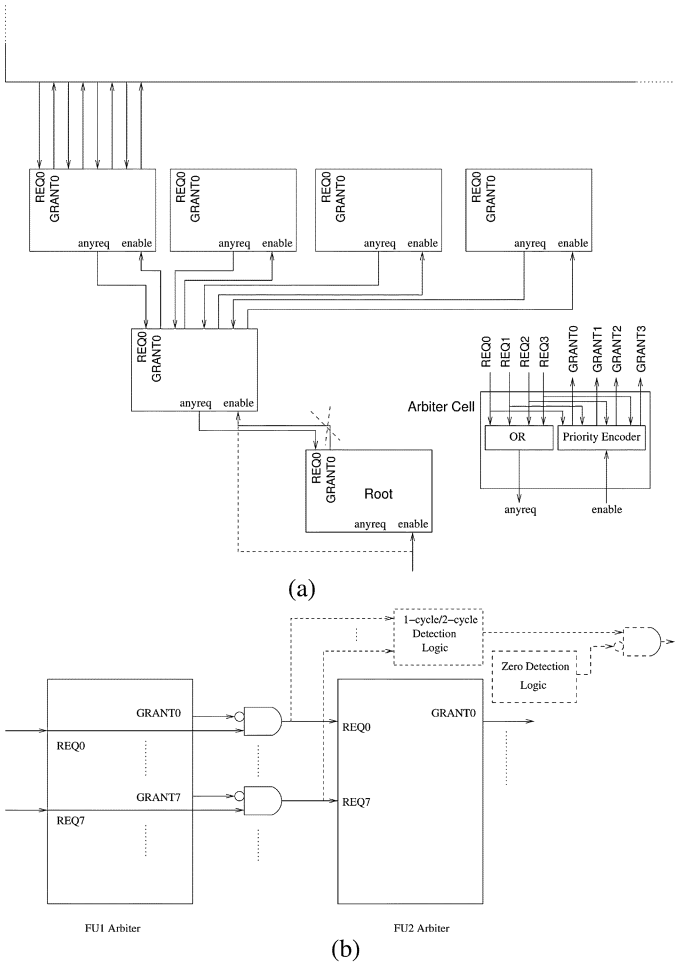


Fig. 2. Selection logic (from [18], the dashed line shows our modifications). (a) Selection logic modification. (b) Two-cycle detection logic.

as shown in Fig. 2(b). The request signal for the second functional unit is asserted only if there was a request for the first functional unit and it was not granted. For further details, we refer the reader to [18].

### B. Input Distribution

We exploit the observation that in the common case, at least some older instructions are ready for issue. Note that this observation does not imply that younger instructions are not ready; only that some older instructions are ready in the common case. Even aggressive ILP exposing techniques that extract ILP from deep within the instruction window do not alter this observation. We demonstrate later that, even with perfect memory disambiguation and perfect branch prediction to expose ILP, the observation remains true. When we consider the earlier observation in terms of inputs to the arbiter tree, we conclude that in the common case, at least some of the first few inputs are likely to be activated. However, the logarithmic delay to traverse the arbiter tree is more or less uniform for all grants. (The priority encoders do create additional delay for lower priority requesters. Thus, older instruction grants are actually slightly faster than younger instruction grants.) However, we can improve the latency of older instruction grants by modifying the arbiter tree.

Instead of using the grant signal from the root node, we bypass the enable signal from the functional unit beyond the root cell. This bypassed signal replaces the grant signal (GRANT0), as shown in Fig. 2(a). Because we always prioritize older instructions, and because priority encoders never forward a grant unless an instruction has raised a request, the bypass path does not alter the functionality of the arbiter tree. By doing so, we reduce the latency of a grant signal to the left-most subtree of the arbiter. After this modification, the delay between a request signal and the grant signal for the first few instructions is shorter than the later instructions. The first few entries, which we look at for selection, will be referred to as *focus window* in the rest of the paper.

### C. Two-Cycle Detection

If the arbiter tree has to select only one instruction for one functional unit, the two-cycle detection operation is very simple. We simply look at the request signal for the *focus window*. If any of those request signals is asserted, then the selection operation is going to be subcritical and can complete in one cycle. If none of the instructions in the *focus window* is ready, then there is a possibility of a two-cycle operation. We observed that in a significant number of cycles, no instruction is ready for issue. If all those cycles are detected as long-latency cycles, there would be a significant degradation in performance. To solve that problem, we introduce a zero-detection logic, which checks if none of the request signals is asserted. This zero-detection logic can work in parallel with the rest of the two-cycle detection circuit and does not introduce an additional delay. Thus, the two-cycle signal is asserted only if the only ready instructions are located outside the *focus window*.

If the arbiter has to select a request from multiple functional units, the two-cycle detection is more complicated and is shown in Fig. 2(b). In this case, simply looking for a ready instruction in the *focus window* will not suffice. This is because the selection logic has to select two ready instructions, one for each functional unit. If there is only one ready instruction in the *focus window*, the selection logic has to select a ready instruction from the rest of the entries in the issue window. We solve this problem by looking at the request signals for the second functional unit. If any of the request signals in the focus window of the second functional unit is asserted, it means that adequate instructions are active in *focus window* and we detect it as a single-cycle operation. Note that the zero-detection logic in this case looks at all the entries other than the *focus window*. Thus, if the *focus window* has an inadequate number of instructions, and if there exist ready instructions in the rest of issue window entries (detected by zero-detection logic), we will flag it as a two-cycle operation.

### D. Implementation

In the rest of this section, we give details of how the modified selection logic interacts with the rest of the pipeline.

When a two-cycle operation is detected, the instruction selection logic will take two cycles to select an instruction from the ready instructions. As previously described in Section III, we must ensure that the inputs are preserved during the second cycle and also that the potentially incorrect output is not used. Therefore, we have to ensure that no new instruction becomes ready during the second cycle, as that would change the inputs to our circuit and may result in inconsistent outcome. As mentioned earlier, we assume a separate instruction issue window for each type of instruction. If a critical operation is detected on any of the issue window, we stall the dispatch stage to ensure no new instructions are placed in issue windows (as new instructions might already be ready to issue). Further, the write back stage is stalled too, so that no tags are broadcasted and no new instructions are woken up. We assume that pipeline stages are stalled by clock gating their input latches.

Since stalling in an OoO processor is more complicated than in an InO processor, it is important to ensure that the detection and stalling delay does not become longer than the delay of the subcritical operation. We evaluated the Illinois Verilog model (IVM) [1], a publicly available OoO processor model. We modified and configured the model to conform to our performance simulations (with exception of a floating-point unit, which the IVM model does not currently have). We then synthesized it to IBM 180 nm technology, and measured the maximum delay required to stall. The total stall delay was 29.7% of the short path. Although this result is specific to the model that we used, it clearly shows that there is sufficient time to detect and stall the pipeline when a two-cycle operation is required.

Our results in Section VIII-A show that two-cycle operations are infrequent and the corresponding decrease in IPC is 5% for integer and 2% for floating-point benchmarks, respectively.

## VI. Process Variations and Trifecta

Thus far, we have presented Trifecta as purely driven by the motivation to exploit common-case, subcritical path operations. In this section, we describe the advantages of Trifecta to improve yield under PV without hurting power or performance.

Manufacturing process parameter variation, and the consequent statistical delay variation of circuits, is quickly becoming a major impediment that is preventing the efficient design of VLSI systems in sub-100nm CMOS technologies [5], [22]. Process variation is usually classified into two categories: D2D or inter-die, which is variation across different dies; and within-die (WID) or intra-die, which is variation among transistors within each die. D2D variation changes the performance corner (fast or slow) of a particular die. This variation affects each transistor in the die in a systematic way; i.e., if a die is in the high $V_{\mathrm{T}}$ (slow) corner, all transistors will have high $V_{\mathrm{T}}$. On the other hand, WID variation affects each transistor differently, and it is possible to have a die with transistors having different $V_{\mathrm{T}}$'s within close proximity.

Traditional responses to D2D PV typically operate by sacrificing one of *power* (i.e., increasing supply voltage to operate the part at nominal frequency and given yield), *yield* (i.e., decreasing yield at nominal frequency and voltage), or *performance* (i.e., decreasing frequency under nominal voltage for a given yield). In contrast, Trifecta offers a clear advantage over the earlier options because it offers the rare triple-combination, wherein parts can achieve significantly higher yield while

TABLE I
SIMULATED PROCESSOR CONFIGURATION

| Processor Configuration | |
|---|---|
| Fetch,Decode,Issue, Commit Width | 4 |
| ROB entries | 256 |
| Issue Queue | 128 entries (See Table II.) |
| Branch prediction | Bimodal & 2-level predictor combined, BTB 2048, RAS 64. |
| Memory Hierarchy | |
| L1 Instruction and Data cache | 2-cycle, 32Kb, 4-way, 32-byte blocks, LRU |
| L2 cache unified | 8-cycle, 1MB, 8-way, 64-byte blocks, LRU |
| Memory latency | 200 cycles |
| Memory Bus Width | 16 bytes |
| Instr and Data TLB | 64 sets, 4-way, LRU |

TABLE II
FUNCTIONAL UNIT AND ISSUE WINDOW CONFIGURATION

| **Instruction Classes** | **Functional Units** | **Issue Queue entries** |
|---|---|---|
| Integer ALU | 2 | 32 |
| Integer Multiply, Divide | 1 | 16 |
| Memory Loads, Stores | 2 | 32 |
| Float Add, Compare, CVT | 2 | 32 |
| Float Mult, Div, SQRT | 1 | 16 |

continuing to operate at nominal voltage and frequency. This is because inputs that would normally cause a timing error can now operate correctly since they are computed over two cycles instead of one. Because the IPC degradation is small and the part operates at nominal frequency, there is little performance loss. One caveat that `Trifecta` must consider is that if WID PV causes the delay of the critical path to exceed twice the delay of the longest subcritical path (which is used to set the clock period), `Trifecta` will fail. Using a combination of architecture-level instruction throughput simulation and Monte Carlo HSPICE simulations, we quantify the benefits of `Trifecta` under PV in Section VI. Although our experiments model both D2D and WID PV, our technique really targets D2D PV. The WID effect is included to show that the delay of critical paths does not exceed twice that of the subcritical paths even under significant WID PV. Our results (shown later in Section VIII) indicate that `Trifecta` achieves 20% and 12.7% improvement in YAT for the SPEC2000 benchmark suite (on average) for the InO and OoO processor configurations, respectively.

## VII. EVALUATION METHODOLOGY

We used Simplescalar 3.0 [3] to perform the architectural simulations with the Alpha ISA. The configuration of the simulated OoO processor is described in Table I. We assume that no more than two integer instructions may be executed in a given cycle (other issue restrictions depend on the number of functional units as listed in Table II). Effectively, our configuration is able to issue four instructions in a cycle only if there are adequate functional units for those instruction types. Similar instruction issue restrictions exist in real processors [12]. We modified Simplescalar to model separate instruction issue queues for different types of instructions. The instruction queue configuration that we used is shown in Table II. We used a *focus window* size of 8 for all the issue windows.

For simulating the InO processor, all the other configuration parameters are the same as the realistic configuration of the OoO processor except that the instructions are issued in order fetch-, decode-, issue-, and commit-bandwidth is limited to 1. We used SimPoint's early simulation points [19] to pick the simulation points, and simulated 100 million instruction for each benchmark in the SPEC2000 benchmark suite.

When applying `Trifecta` to the integer execution unit, we considered all the instructions which use the adder.[2] These include adds, subtracts, and address calculation for load and store instructions. The address calculation part of load and store is treated as a separate instruction, which is executed in the integer ALU. When the address calculation is completed, the result is supplied to the load/store instructions residing in the load/store units. If a long-latency add operation is detected, the adder is set to be occupied for two cycles.

*Circuit-Level Speculation:* Liu *et al.* [15] applied their CLS technique to rename, selection, and execution (adder). For selection, they approximated the full circuit by selecting only from the first few instructions (we use the same definition of *focus window* and use an identical *focus window* size as our scheme, i.e., 8). There is no mis-speculation in this case, and there is no complete version required for this pipeline stage. For the adder, Liu *et al.* assumed a CLA, which looks at only the last 4 bits for generation of a carry. The results of this speculative adder are compared against a slow adder which takes two cycles. CLS requires two such adders in order to start a verification operation in each cycle. On mis-speculation, the offending instruction and all dependent instructions that used the wrong value are reissued. We introduced a simple optimization to CLS. Instead of reissuing the mis-speculated operation, we read the correct value from the complete adder one cycle after the speculated instruction completes. If any dependent instructions are issued before the correct version finishes, they are reissued.

*Process Variation:* We evaluated `Trifecta` for PV by performing HSPICE Monte Carlo simulations to measure the delays of both the critical path and the longest subcritical path operations. Since it is possible that a near-critical path will become critical under PV, we measured two sets of delays. One set contained all the delays that were within 5% of the critical path (delay for the long operation), while the other set contained all the paths whose delays were within 5% of the short operation. The actual delay for each set was the maximum delay in the set.

We lump the effects of various PV sources (random dopant fluctuation, effective length, and oxide thickness variations) into the threshold voltage, $V_{\mathrm{T}}$. For each simulation, we assign a threshold voltage for each transistor from the following equation: $V_{\mathrm{T}} = V_{\mathrm{T0}} + \sigma_{\mathrm{VT,D2D}} + \sigma_{\mathrm{VT,WID}}$, where $V_{\mathrm{T0}}$ is the nominal threshold voltage when there is no PV, $\sigma_{\mathrm{VT,D2D}}$ and $\sigma_{\mathrm{VT,WID}}$ are the normally distributed D2D and WID standard deviations, respectively. In each circuit, the $V_{\mathrm{T,D2D}}$ is the same for all transistors, while each transistor is assigned a different

[2]Recall that instructions that do not exercise the adder do not exercise the critical path.

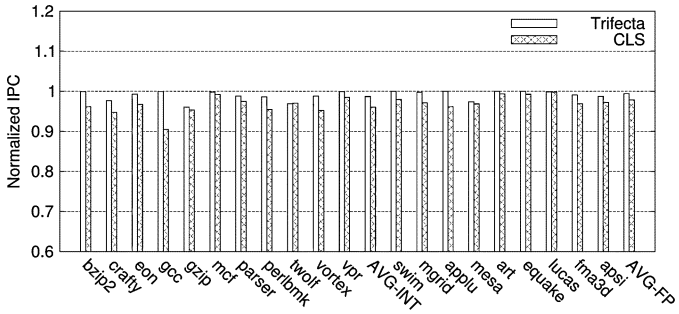


Fig. 3. Trifecta: IPC degradation on single issue InO.

$V_{T,WID}$. We measure the maximum delays of the short and long paths for 1000 simulations for both the selection logic and the ALU. Note that the integer ALU included muxes and drivers as shown in Fig. 1(b). The loopback bus for result bypassing was simulated as a fixed load capacitance to avoid feedback-related convergence problems in SPICE. The adder delay accounts for 80% of the total ALU delay.

## VIII. Results

The three primary conclusions from our simulations are the following.

1) Trifecta achieves comparable IPC as CLS, with 2.4%–7% improvement in IPC on average. Note, the minor gains in performance must be viewed in conjunction with the fact that CLS has significant area- and power-overheads.
2) The circuit performance results for Trifecta reveal, as expected, that there is a well-defined separation between short and long paths, which is maintained even under PV.
3) Trifecta improves the YAT by as much as 20% (12.7%) for the InO (OoO) configuration compared to the base case with speed binning applied to both the cases.

The remainder of this section is organized as follows. Section VIII-A presents results to show the IPC degradation caused by two-cycle operations in our scheme. For these results, we compare against an ideal base case which completes all operations in one cycle. Next, we describe the results of our HSPICE simulations to quantify circuit performance (Section VIII-B) and the impact of PV (Section VIII-C). Finally, we quantify the improvement in YAT by using a combination of the architectural simulation data and the circuit performance data in Section VIII-D.

### A. Effect of Two-Cycle Operation on IPC

*InO processor:* For the InO processor, we applied our technique to the integer execution unit. The results we obtained are presented in Fig. 3. For the base case, we assumed an integer ALU which can complete all operations in a single cycle. Fig. 3 plots the IPC with occasional two-cycle operations normalized to the base case. It can be seen that the performance loss due to two-cycle additions is negligible (less than 2% and 1% for integer and floating-point benchmarks, respectively). We note that *all* two-cycle additions are due to the addition of small numbers of dissimilar sign. We extend CLS to the InO processor and our results show a trend similar to the OoO processor with

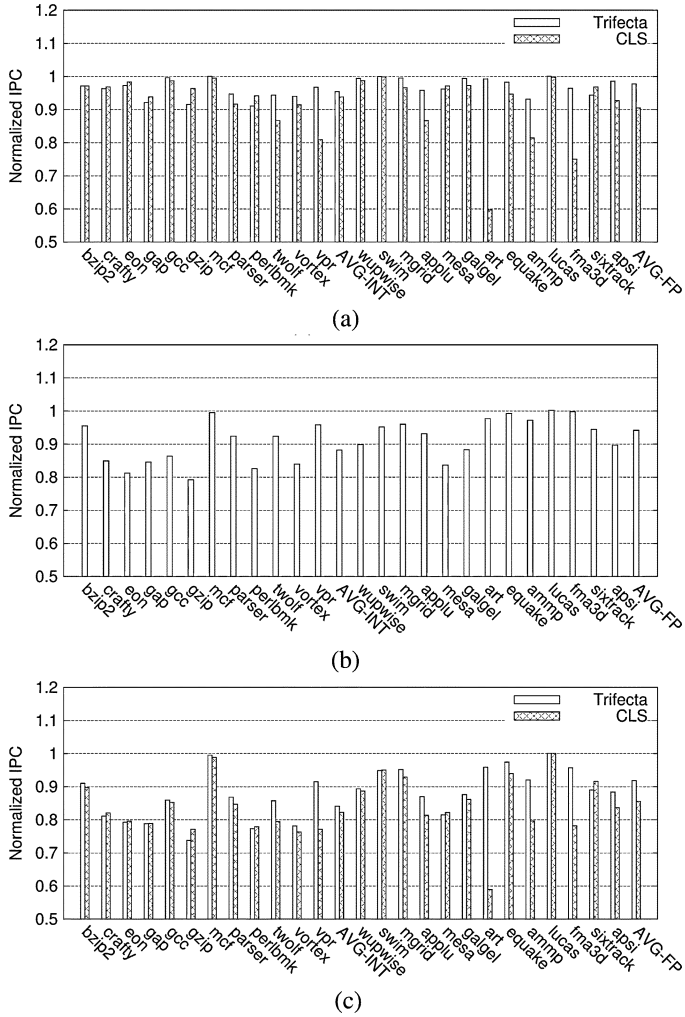


Fig. 4. Trifecta: IPC degradation on OoO with realistic configuration. (a) Issue. (b) Adder. (c) Issue and adder.

CLS losing 4% and 3% of performance respectively, compared to base case. The IPC of CLS is worse than Trifecta by 2% on average, for both integer and floating-point benchmarks.

*OoO processor:* Fig. 4(a) shows the results of the application of our technique to the selection logic. The IPC is normalized to the base case. Results for CLS are also shown for reference. There is a decrease in IPC of 5% and 2% on average for integer and floating-point benchmarks, respectively. Trifecta and CLS achieve comparable IPCs on all benchmarks except vpr, art, ammp, and fma3d where CLS performs worse by up to 40%. This results in IPC degradation of 1.7% and 8.5% for integer and floating-point benchmarks on average for CLS, compared to Trifecta. The reason for the lower performance of CLS relative to Trifecta is that CLS selects ready instructions only from the *focus window*. Thus, if there are older instructions in the issue window, which are waiting for a long-latency operation such as a cache miss, no other instructions would be issued even if there are ready instructions outside the *focus window*.

The minimal decrease in IPC from restricting the issue to the *focus window* for short-latency operations can be explained by

the following observation. We found that in the majority of cycles (91% for integer and 93% for floating point with the realistic configuration, and 89% for integer and 93% for floating point with the ideal configuration), the single-cycle select operation is possible. This means that we can select as many ready instructions as the number of available functional units from the *focus window*. The remainder are two-cycle operations.

To rule out the possibility that branch prediction or memory dependencies are choking ILP (and resulting in a weakened base configuration), we also simulate an *ideal* configuration that has perfect branch prediction and perfect load dependence prediction. With the perfect load dependence predictor, a load is issued as soon as the last store to the same address is ready. A store is defined as ready if its effective address has been calculated and its operands (from which the value to be stored is calculated) are ready. It is assumed that the load directly reads the value from the last store, using the load/store queue. We noticed that with the *ideal* configuration, the IPC of `Trifecta` degrades slightly (within 1%), but the IPC of CLS is worse than `Trifecta` by 4.3% and 8.4% for integer and floating-point benchmarks, respectively. The reason behind this trend is that the ideal configuration exposes more ILP, thus making more of the instructions beyond *focus window* ready to be issued. This affects `Trifecta` to a smaller degree, since the instructions beyond the *focus window* are issued too (using the two-cycle operation). In the case of CLS, instructions beyond the *focus window* are not issued at all, causing its performance to be worse than `Trifecta`.

We also performed experiments to study the application of `Trifecta` to the execution stage of an OoO processor, as described in Section IV-D. We noticed that the performance of `Trifecta` for the execution stage of an OoO processor is the same as CLS. This is because both of them cause long-latency instructions to take two cycles and cause the immediate dependent instructions to reissue.[3] Fig. 4(b) shows the performance (IPC) results for the execution stage of an OoO processor. Since `Trifecta` and `CLS` are the same numerically, we show only one bar for each benchmark. The average IPC loss is 12% and 6% for integer and floating-point benchmarks, respectively. Fig. 4(c) shows the results for the case when both issue and execution stages can have two-cycle operations. `Trifecta`'s IPC is lower than the base case by 16% and 8% for integer and floating-point benchmarks, on average. `CLS` is worse than `Trifecta` by 2.4% and 7% for integer and floating-point benchmarks. As expected, the IPC in this case is lower (both for `Trifecta` and `CLS`) than the cases when only one of the two stages can have two-cycle operations.

### *B. Circuit-Level Performance of Adder and Select Under Process Variation*

The circuits were implemented in Verilog and synthesized using Synopsys Design Compiler to a 0.25 $\mu$m technology library containing both logic and wire models. A Verilog-HSPICE netlist converter was then used to generate the HSPICE netlists. The dimensions of the circuits were scaled down to 70 nm and the predictive technology model (PTM) 70 nm model was used for HSPICE simulations [2]. The area and power overhead computed by Synopsys design compiler and the delays from HSPICE are shown in Table III.

[3]Notice that `Trifecta` still has less area overhead than CLS, since the adder does not have to be replicated.

TABLE III
NORMALIZED CIRCUIT DELAYS AND OVERHEAD

| Overhead | | Adder | Sel Logic Byp | Sel Logic Base |
|---|---|---|---|---|
| Delay: | Detection circuit | 0.32 | 0.47 | 0.47 |
| | Short path | 0.84 | 0.68 | 0.85 |
| Area | | 1.02 | 1.10 | 1.10 |
| Power | | 1.01 | 1.11 | 1.09 |

*Adder circuit performance:* First, the critical path was measured. It was found that the delay from bit A[0] to bits SUM[63:57] was within 1% of each other depending on whether the signals were rising or falling, so the maximum of delay was measured. Next, the delay of the short addition was measured. Similar to the earlier measurement, it was found that the delay from A[0] to S[36:29] was the maximum delay for the short operations. Moreover, the delay of the short operations was approximately 16% shorter than the delay of the critical path operations.

The delay of the carry length detection circuit (CLDC) is also important to consider. If the delay is larger than the delay of the short addition, then the delay of the short addition becomes the delay of detection. Since we detect 8 bits, the delay of detection is equivalent to the delay of an 8-bit carry-merge tree.

Table III shows the delay of the CLDC and the short addition normalized to the critical path in the adder. It also shows the area and power overheads of our modification normalized to the base case.

*Selection logic circuit:* The selection logic was also implemented as described earlier. HSPICE delay measurements were made to determine the delay of the long selection path, short selection path, and the detection logic. We measured the time it takes from request assertion to the grant signal generation. For the stacked selection logic, the delay $gr_1[15], gr_2[31]$ is the total delay measured from request signal to grant[31] for FU2, assuming that grant[15] for FU1 is also generated (only req[15] and req[31] were asserted). Since the grant signal for FU2 $(gr_2)$ is generated based on the grant of FU1 $(gr_1)$, $gr_2$ will determine the total delay. Similar to the delay measurements of the adder, several paths were found to be near-critical. Therefore, several of these paths were measured. Table III shows the delay of the two-cycle detection circuit. It also shows the delay of longest subcritical path for both the original selection logic and the bypassed selection logic. The delay numbers are normalized to the nominal critical path delay in the selection logic. The table also shows the area and power overheads (normalized to the base case) of our modifications over base selection logic and bypassed selection logic.

### *C. Effects of Process Variation*

Figs. 5 and 6 show the effects of PV on the proposed technique. For this discussion, the term *nominal* refers to the delay when there is no PV. For all simulations, we assumed $\sigma_{\mathrm{VT,D2D}} = 20\%$ and $\sigma_{\mathrm{VT,WID}} = 10\%$. We normalize both the critical and short path delays to the *nominal* critical path delay. For the proposed scheme, it is evident that the scheme

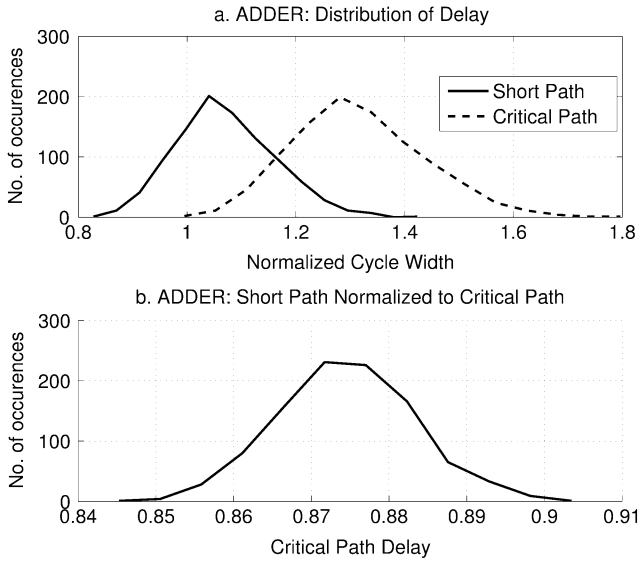


Fig. 5. QTA a. Short and Critical Path Delay Normalized to Nominal Delay, b. Short Path Delay Normalized to Critical Path Delay. $\sigma_{\mathrm{VT,D2D}} = 20\%, \sigma_{\mathrm{VT,WID}} = 10\%$.

would break down if, due to PVs, the delay of the critical paths exceeded twice the delay of the longest subcritical path. In this case, more than two cycles would be required to perform the computation, and this would require a complicated modification to the detection logic. Therefore, it is important to ensure that this situation does not occur. D2D variations are unlikely to cause this situation to occur because the delay of all paths is impacted equally. However, if the effects of WID variation were extreme, it is quite possible that this scenario would occur. We evaluated the proposed scheme, taking into account significant WID variation ($\sigma_{\mathrm{VT,WID}} = 10\%$). For each simulation, we measured the maximum delay of the short paths and the critical path. We first evaluate the QTA under PV. As previously mentioned, the delay of the short path is 84% of the critical path delay. Since no modifications are made to the adder to reduce the short path delay (such as the bypass in the selection logic), it is important to ensure that the circuit operates even under severe variation. Fig. 5(a) shows the delay distribution (normalized to the nominal critical path) for both the critical and longest subcritical paths. From the overlap of the areas under the curves, it would appear as if there is no clear distinction between short and critical paths. However, if we plot the longest subcritical path normalized to the critical path delay of the same circuit (i.e., same D2D variation) as shown in Fig. 5(b), the distinction is clear. Even under significant WID variation, the scheme performs as expected. Similar simulations were then performed for the selection logic, and the results are plotted in Fig. 6(a) and (b). For the modified selection logic, the normalized delay (normalized to the critical path delay) of the short paths remains well below 80%, indicating that even under such WID variation, the bypassed selection logic still performs correctly. The base selection logic exhibits similar distribution in delay under PV, although the normalized short path delay exceeds 90% for a few circuits as shown in Fig. 6(b).

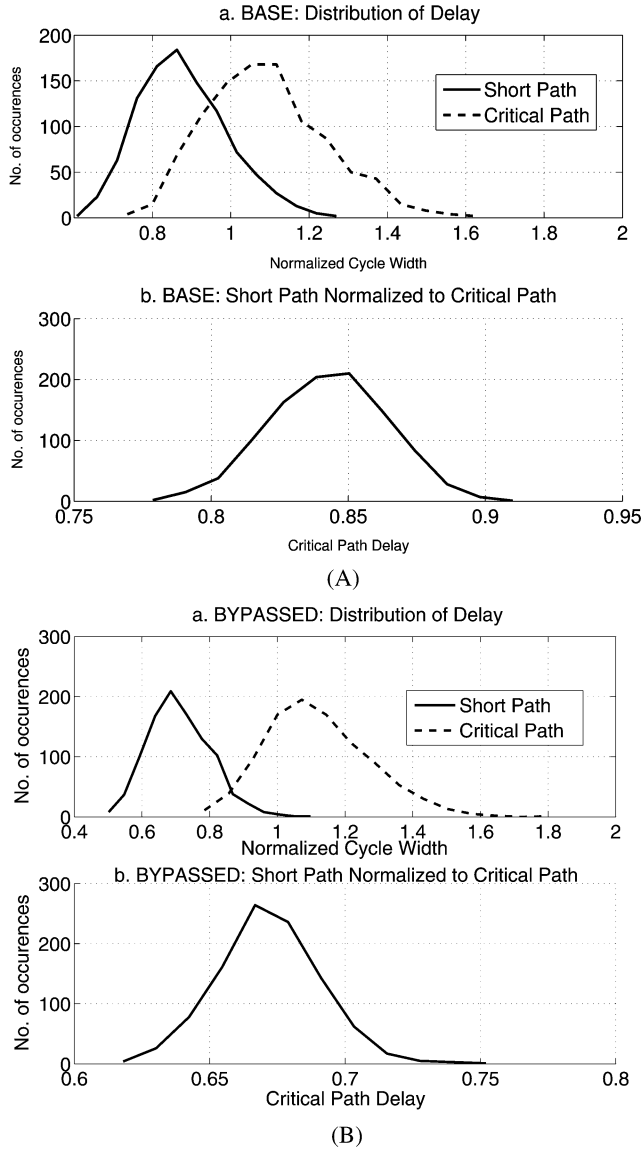


Fig. 6. Selection logic : comparison of nominal, short path and critical path delays. $\sigma_{\mathrm{VT,D2D}} = 20\%, \sigma_{\mathrm{VT,WID}} = 10\%$. (a) Base selection logic. (b) Bypassed selection logic.

### D. Improvements in YAT

We use a metric called YAT [20], [21] to evaluate the improvement offered by our scheme. YAT factors in yield, IPC, and clock frequency and gives a consolidated metric for comparison in the presence of PV. We apply speed binning to both `Trifecta` and the base case. For binning, we choose the bin spacing to be 10% of the nominal clock period, and take as many bins as necessary to ensure that all the chips function (i.e., overall yield is 100%). Consider an arbitrary bin (say $k$th) that operates at frequency $f_k$ and has $s_k$ samples out of a total of $s_n$ samples. The contribution of the given bin to the overall YAT is defined as $(s_k/s_n) \times \mathrm{IPC} \times f_k$ where $(s_k/s_n)$ is the yield of the bin. The aggregate YAT is computed as the summation of the YAT contributions of all bins and is given by $\mathrm{YAT} = \mathrm{IPC}(\Sigma_n((f_k \times s_k)))/s_n$. Since `Trifecta` sets the frequency of a chip based on subcritical paths, `Trifecta` will be able to operate at a higher frequency, resulting in more yield

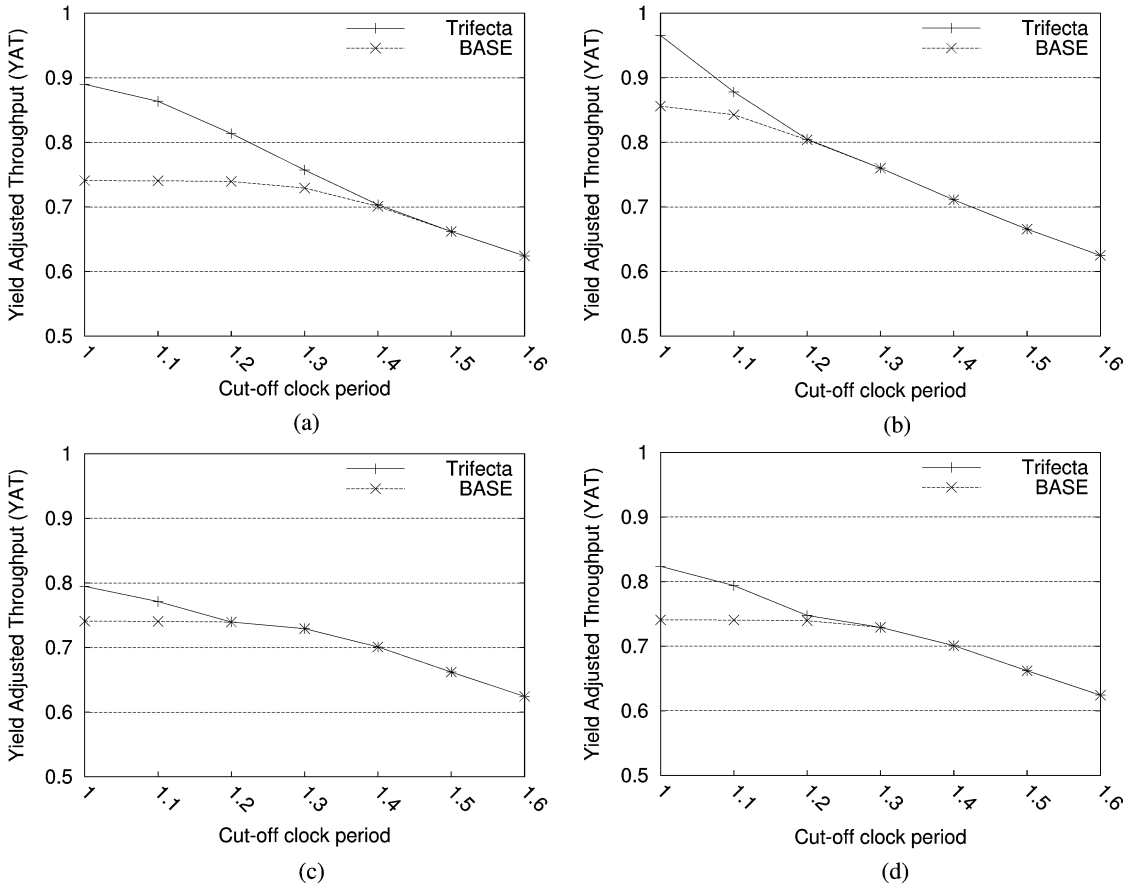


Fig. 7. YAT with speed binning. (a) InO processor with adder. (b) OoO processor with select. (c) OoO processor with adder. (d) OoO processor with adder and select.

at higher frequency bins. Moreover, since the IPC degradation due to `Trifecta` is small, the YAT of `Trifecta` will be significantly better. We measured YAT at different cut-off clock periods (cutoff period corresponds to the frequency of the fastest bin). Fig. 7(a) plots the YAT for an InO processor with `Trifecta` applied to the adder logic. We used average IPC (for both integer and floating-point benchmarks) for the purpose of these computations. The *y*-axis shows the aggregate YAT while the *x*-axis shows the cut-off period, which is the period of the fastest bin. Fig. 7(a) shows that `Trifecta` has an improvement of 20% over base case if the fastest bin is operated at the nominal frequency. The improvements diminish as the cutoff clock period increases. This is because even the base case is able to operate at the lower frequencies (higher cutoff clock periods) with reasonable yield. Beyond a threshold, slowing the cutoff clock period burdens `Trifecta` with the disadvantages of two-cycle operation (thus reducing IPC). However, `Trifecta` loses the advantage of improved yield because both `Trifecta` and the base case can operate correctly at slow frequencies. Consequently, the lower IPC of `Trifecta` leads to a lower YAT. Therefore, we disable `Trifecta` for cut-off frequencies which are low enough to show no YAT improvement. Disabling `Trifecta` can be trivially achieved by always de-asserting the two-cycle detection logic, which requires a single-bit enable/disable signal. We empirically observe from Fig. 7 that the thresholds were approximately 20% of nominal clock period for OoO designs and approximately 40% of nominal clock period for the InO processor. This is effectively equivalent to saying that yield degradation due to PV can be countered by having 20% and 40% slower clock periods for OoO and InO processors, respectively. Fig. 7(b) shows YAT results for an OoO processors with `Trifecta` applied to the selection logic. In this case, `Trifecta` shows a YAT improvement of 12.7% over the base case for nominal frequency. For the sake of completeness, we also show YAT results for the adder in the OoO processor in Fig. 7(c). Again, `Trifecta` shows 11.2% improvement over base case at nominal clock frequency, with gains diminishing as clock frequency of the fastest bin is lowered. Finally, when `Trifecta` is applied to both select and adder as shown in Fig. 7(d), the improvement in YAT is 7.3%. This decrease in IPC is primarily due to the lower IPC that results since two-cycle operations are now possible both in the select and the execution stages of the pipeline (see Section VIII-A).

## IX. Conclusion

The conventional design approach wherein clock cycles are designed to accommodate the worst-case delay is a wasteful design approach that leads to reduced performance and/or increased power consumption. Previous approaches for exploiting common-case data-dependent subcritical paths suffer from either high design complexity (asynchronous computation) or in-

creased performance overhead of speculation and potential rollback and recovery.

This paper proposes `Trifecta`—a nonspeculative technique to exploit data-dependent subcritical paths. `Trifecta` uses fast, conservative one-cycle operation detection to ensure that inputs that exercise subcritical paths complete in a single cycle. Other inputs require two cycles to complete. We demonstrate the application of `Trifecta` to the critical stages of InO (integer execution) and OoO (select) processor configurations. Simulations reveal that `Trifecta` performs better by 2.4% and 7% than CLS for integer and floating-point benchmarks respectively, for the OoO processor. For the InO processor, `Trifecta` improves the performance by 2% over an improved version of CLS. Although the performance gap is not significant, the nonspeculative nature of `Trifecta` reduces the area- and power-overheads associated with CLS, while achieving comparable performance. `Trifecta`'s benefits are significant when we consider D2D PV. Unlike other PV-tolerating techniques, such as speed binning, adaptive body biasing (ABB) [5][22], and dynamic voltage scaling, `Trifecta` offers the rare triple combination of power-performance-yield. For the InO (OoO) processor configuration, `Trifecta` offers a 20% (12.7%) improvement in YAT for SPEC2000 benchmark suite.

## REFERENCES


[1] Illinois Verilog Model [Online]. Available: http://www.crhc.uiuc.edu/ACS/tools/ivm/about.html

[2] Predictive Technology Model [Online]. Available: http://www.eas.asu.edu/~ptm

[3] T. Austin, E. Larson, and D. Ernst, "Simple scalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[4] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *Proc. 8th Int. Symp. High-Performance Comput. Architect.*, Feb. 2002, pp. 299–310.

[5] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Proc. Des. Autom. Conf. (DAC)*, 2003, pp. 338–342.

[6] M. Butler and Y. Patt, "An investigation of the performance of various dynamic scheduling techniques," in *Proc. 25th Annu. Int. Symp. Microarchit.*, Los Alamitos, CA, 1992, pp. 1–9.

[7] Y. Chen, H. Li, K. Roy, and C.-K. Koh, "Cascaded carry-select adder (C2SA): A new structure for low-power CSA design," in *Proc. 2005 Int. Symp. Low Power Electron. Des. (ISLPED'05)*, New York, 2005, pp. 115–118.

[8] 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 *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit*, Washington, DC, 2003, pp. 7–18.

[9] J. A. Farrell and T. C. Fischer, "Issue logic for a 600-MHz out-of-order execution microprocessor," *IEEE J. Solid-State Circuits*, vol. 33, no. 5, pp. 707–712, May 1998.

[10] S. Ghosh, P. Ndai, S. Bhunia, and K. Roy, "Tolerance to small delay defects by adaptive clock stretching," in *Proc. 13th IEEE Int. On-Line Testing Symp. (IOLTS-07)*, Jul. 2007, pp. 244–252.

[11] S. Ghosh, S. Bhunia, and K. Roy, "A new paradigm for low-power, variation-tolerant circuit synthesis using critical path isolation," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des. (ICCAD'06)*, New York, 2006, pp. 619–624.

[12] L. Gwennap, Microprocessor Rep. pp. 12–15, Oct. 1998.

[13] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, Opt. Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays 2002, pp. 14–24.

[14] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *Proc. 39th Annu. IEEE/ACM Int. Symp. on Microarchit. (MICRO 39)*, Washington, DC, Dec. 2006, pp. 504–514.

[15] T. Liu and S.-L. Lu, "Performance improvement with circuit-level speculation," in *Proc. 33rd Annu. ACM/IEEE Int. Symp. Microarchit.*, New York, 2000, pp. 348–355.

[16] S. Matthew, R. Krishnamurthy, M. Anders, R. Rios, K. Mistry, and K. Soumyanath, "Sub-500 ps 64-b ALUs in 0.18 mm SOI/Bulk CMOS: Design and scaling trends," *IEEE J. Solid State Circuits*, vol. 36, no. 11, pp. 1636–1646, Nov. 2001.

[17] V. G. Oklobdzija, B. R. Zeydel, H. Q. Dao, S. Mathew, and R. Krishnamurthy, "Comparison of high-performance VLSI adders in the energy-delay space," *IEEE Trans. VLSI Syst.*, vol. 13, no. 6, pp. 754–758, Jun. 2005, 2005.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity effective superscalar processors," in *Proc. 24th Annu. Int. Symp. Comput. Archit.*, Jun. 1997, pp. 206–218.

[19] E. Perelman, G. Hamerly, and B. Calder, "Picking statistically valid and early simulation points," in *Int. Conf. Parallel Archit. Compilat. Techn.*, Sep. 2003, pp. 244–255.

[20] E. Schuchman and T. N. Vijaykumar, "Rescue: A microarchitecture for testability and defect tolerance," in *Proc. 32nd Int. Symp. Comput. Archit.*, New York, Jun. 2005, pp. 160–171.

[21] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger, "Exploiting microarchitectural redundancy for defect tolerance," in *Proc. 21st Int. Conf. Comput. Des. (ICCD'03)*, Washington, 2003, pp. 481–488.

[22] J. Tschanz, K. Bowman, and V. De, "Variation-tolerant circuits: Circuit solutions and techniques," in *Des. Autom. Conf. (DAC)*, Jun. 2005, pp. 762–773.

[23] G. Wolrich, E. Mclellan, L. Harada, J. Montanaro, and R. A. J. Yodlowski, "A high performance floating point coprocessor," *IEEE J. Solid-State Circuits*, vol. 19, no. 5, pp. 690–696, Oct. 1984.

[24] R. Woo, S.-J. Lee, and H.-J. Yoo, "A 670 ps, 64 bit dynamic low power adder design," in *IEEE Int. Symp. Circuits. Syst.*, May 2000, pp. 28–31.


**Patrick Ndai** (S'00) received the B.S.E.E. degree (*magna cum laude*) from Washington State University, Pullman, in 2004. He is currently working toward the Ph.D. degree at Purdue University, West Lafayette, IN.

During summer 2006 and 2007, he was an Intern at Circuit Research Lab, Intel Corporation, where he worked on fault-tolerant adders and register file min-Vcc issues. His research interests include low power, high performance VLSI systems and architecture-aware circuit design for resilience and variation tolerance.

**Nauman Rafique** received the B.S. degree in electronics from Ghulam Ishaq Khan Institute, Pakistan, in 2001 and the M.Sc. and Ph.D. degrees in computer engineering from Purdue University, West Lafayette, IN, in 2008.

Since 2008, he has been working at Google, San Francisco, CA. His research interests include resource management issues for multicore processors, computer architecture, operating systems, and large scale distributed systems.

**Mithuna Thottethodi** received the B. Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Kharagpur, in 1996 and the Ph.D. degree from Duke University, Durham, NC, in 2002.

He is currently an Assistant Professor of electrical and computer engineering at Purdue University, West Lafayette, IN. His research interests include computer architecture with a focus on parallel architectures and interconnection networks

**Swaroop Ghosh** received the B.E. degree from the Indian Institute of Technology, Roorkee, in 2000, the Masters degree from the University of Cincinnati, OH, in 2004, and the Ph.D. degree from Purdue University, West Lafayette, IN, in 2008.

From 2000 to 2002, he was with Mindtree Technologies, Bangalore, India. Currently, he is working in the Advanced Design Group, Intel, Hillsboro, OR. His research interests include low-power, process tolerant adaptive circuit and system design, fault tolerant design and digital testing for nanometer technologies.

**Swarup Bhunia** (S'00–M'05) received the B.E. degree (Hons.) from Jadavpur University, Kolkata, India, in 1995, the M.Tech. degree from the Indian Institute of Technology (IIT), Kharagpur, India, in 1997, and the Ph.D. degree from Purdue University, West Lafayette, IN, in 2005.

Currently, he is an Assistant Professor of electrical engineering and computer science at Case Western Reserve University, Cleveland, OH. His research interests includes low-power and robust VLSI design, adaptive nanocomputing and bioimplantable electronics.

Dr. Bhunia received the 2005 SRC Technical Excellence Award as a team member, the Best Paper Award at the International Conference on Computer Design (ICCD 2004), the Best Paper Award at the Latin American Test Workshop (LATW 2003), and the Best Paper nomination at the Asia and South Pacific Design Automation Conference (ASP-DAC 2006).

**Kaushik Roy** (SM'95–F'01) received the B.Tech. degree in electronics and electrical communications engineering from the Indian Institute of Technology, Kharagpur, India, and the Ph.D. degree from the University of Illinois at Urbana-Champaign, in 1990.

He was with the Semiconductor Process and Design Center of Texas Instruments Incorporated, Dallas, where he worked on FPGA architecture development and low-power circuit design. He joined the Electrical and Computer Engineering Faculty at Purdue University, West Lafayette, IN, in 1993, where he is currently a Professor and is the Roscoe H. George Professor of Electrical and Computer Engineering. His research interests include VLSI design/CAD for nano-scale silicon and non-silicon technologies, low-power electronics for portable computing and wireless communications, VLSI testing and verification, and reconfigurable computing. He has published more than 450 papers in refereed journals and conferences, holds eight patents, and is a coauthor of two books on low-power CMOS VLSI design.

Dr. Roy received the National Science Foundation Career Development Award in 1995, the IBM Faculty Partnership Award, the ATT/Lucent Foundation Award, 2005 SRC Technical Excellence Award, SRC Inventors Award, Purdue College of Engineering Research Excellence Award, and Best Paper Awards at the 1997 International Test Conference, the IEEE 2000 International Symposium on Quality of IC Design, the 2003 IEEE Latin American Test Workshop, the 2003 IEEE Nano, the 2004 IEEE International Conference on Computer Design, the 2006 IEEE/ACM International Symposium on Low Power Electronics and Design, he was co-recipient of the 2005 IEEE Circuits and System Society Outstanding Young Author Award, and received the 2006 IEEE Transactions on VLSI Systems Best Paper Award. He is a Purdue University Faculty Scholar. He was a Research Visionary Board Member of Motorola Labs (2002). He has been on the Editorial Board of the *IEEE Design and Test*, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS, and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED SYSTEMS. He was Guest Editor for the Special Issue on Low-Power VLSI in the *IEEE Design and Test* (1994), the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATED SYSTEMS (June 2000), and the *Institution of Electrical Engineers Computers and Digital Techniques Proceedings* (July 2002).