BlueShift: Designing Processors for Timing Speculation from the Ground Up*

Brian Greskamp, Lu Wan, Ulya R. Karpuzcu, Jeffrey J. Cook,
Josep Torrellas, Deming Chen, and Craig Zilles
Departments of Computer Science and of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{greskamp, luwan2, rkarpu2, jjcook, torrella, dchen, zilles}@illinois.edu

Abstract

Several recent processor designs have proposed to enhance performance by increasing the clock frequency to the point where timing faults occur, and by adding error-correcting support to guarantee correctness. However, such Timing Speculation (TS) proposals are limited in that they assume traditional design methodologies that are suboptimal under TS. In this paper, we present a new approach where the processor itself is designed from the ground up for TS. The idea is to identify and optimize the most frequently-exercised critical paths in the design, at the expense of the majority of the static critical paths, which are allowed to suffer timing errors. Our approach and design optimization algorithm are called BlueShift. We also introduce two techniques that, when applied under BlueShift, improve processor performance: On-demand Selective Biasing (OSB) and Path Constraint Tuning (PCT). Our evaluation with modules from the OpenSPARC T1 processor shows that, compared to conventional TS, BlueShift with OSB speeds up applications by an average of 8% while increasing the processor power by an average of 12%. Moreover, compared to a high-performance TS design, BlueShift with PCT speeds up applications by an average of 6% with an average processor power overhead of 23% — providing a way to speed up logic modules that is orthogonal to voltage scaling.

1 Introduction

Power, design complexity, and reliability concerns have dramatically slowed down clock frequency scaling in processors and turned industry's focus to Chip Multiprocessors (CMPs). Nevertheless, the need for per-thread performance has not diminished and, in fact, Amdahl's law indicates that it becomes critical in parallel systems.

One way to increase single-thread performance is Timing Speculation (TS). The idea is to increase the processor's clock frequency to the point where timing faults begin to occur and to equip the design with microarchitectural techniques for detecting and correcting the resulting errors. A large number of proposals exist for TS architectures (e.g., [1, 5, 6, 9, 11, 14, 20, 24, 25]). These proposals add a variety of hardware modifications to a processor, such as enhanced latches, additional back-ends, a checker module, or an additional core that works in a cooperative manner.

We argue that a limitation of current proposals is that they assume traditional design methodologies, which are tuned for worst-case conditions and deliver suboptimal performance under TS. Specifically, existing methodologies strive to eliminate slack from all timing paths in order to minimize power consumption at the target frequency. Unfortunately, this creates a *critical path wall* that impedes overclocking. If the clock frequency increases slightly beyond the target frequency, the many paths that make up the wall quickly fail. The error recovery penalty then quickly overwhelms any performance gains from higher frequency.

In this paper, we present a novel approach where the processor itself is designed *from the ground up* for TS. The idea is to identify the most *frequently-exercised* critical paths in the design and speed them up enough so that the error rate grows much more slowly as frequency increases. The majority of the static critical paths, which are rarely exercised, are left unoptimized or even deoptimized — relying on the TS microarchitecture to detect and correct the infrequent errors in them. In other words, we optimize the design for the common case, possibly at the expense of the uncommon ones. We call our approach and design optimization algorithm *BlueShift*.

This paper also introduces two techniques that, when applied under BlueShift, improve processor performance. These techniques, called *On-demand Selective Biasing (OSB)* and *Path Constraint Tuning (PCT)*, utilize BlueShift's approach and design optimization algorithm. Both techniques target the paths that would cause the most frequent timing violations under TS, and add slack by either forward body biasing some of their gates (in OSB) or by applying strong timing constraints on them (in PCT).

Finally, a third contribution of this paper is a taxonomy of design for TS. It consists of a classification of TS architectures, general approaches to enhance TS, and how the two relate.

We evaluate BlueShift by applying it with OSB and PCT on modules of the OpenSPARC T1 processor. Compared to a conventional TS design, BlueShift with OSB speeds up applications by an average of 8% while increasing the processor power by an average of 12%. Moreover, compared to a high-performance TS design, BlueShift with PCT speeds up applications by an average of 6% with an average processor power overhead of 23% — providing a way to speed up logic modules that is orthogonal to voltage scaling.

This paper is organized as follows. Section 2 gives a background; Section 3 presents our taxonomy for TS; Section 4 introduces BlueShift and the OSB and PCT techniques; Sections 5 and 6 evaluate them; and Section 7 highlights other related work.

*This work was supported by Sun Microsystems under the UIUC OpenSPARC Center of Excellence, the National Science Foundation under grant CPA-0702501, and SRC GRC under grant 2007-HJ-1592.

2 Timing Speculation (TS)

As we increase a processor's clock frequency beyond its *Rated Frequency* f_r, we begin to consume the guardband that was set up for process variation, aging, and extreme temperature and voltage conditions. As long as the processor is not at its environmental limits, it can be expected to operate fault-free under this overclocking. However, as frequency increases further, we eventually reach a *Limit Frequency* f_0, beyond which faults begin to occur. The act of overclocking the processor past f_0 and tolerating the resulting errors is *Timing Speculation (TS)*.

TS provides a performance improvement when the speedup from the increased clock frequency subsumes the overhead of recovering from the timing faults. To see how, consider the performance *perf(f)* of the processor clocked at frequency *f*, in instructions per second:

$$\begin{aligned} perf(f) &= \frac{f}{CPI_{norc}(f) + CPI_{rc}(f)} = \\ &= \frac{f}{CPI_{norc}(f) \times (1 + P_E(f) \times rp)} = \\ &= \frac{f \times IPC_{norc}(f)}{1 + P_E(f) \times rp} \end{aligned} \quad (1)$$

where, for the average instruction, $CPI_{norc}(f)$ are the cycles taken without considering any recovery time, and $CPI_{rc}(f)$ are cycles lost to recovery from timing errors. In addition, P_E is the probability of error (or error rate), measured in errors per non-recovery cycle. Finally, *rp* is the recovery penalty per error, measured in cycles.

Figure 1 illustrates the tradeoff. The plots show three regions. In Region 1, $f < f_0$, so P_E is zero and *perf* increases consistently, impeded only by the application's increasing memory CPI. In Region 2, errors begin to manifest, but *perf* continues to increase because the recovery penalty is small enough compared to the frequency gains. Finally, in Region 3, recovery overhead becomes the limiting factor, and *perf* falls off abruptly as *f* increases.

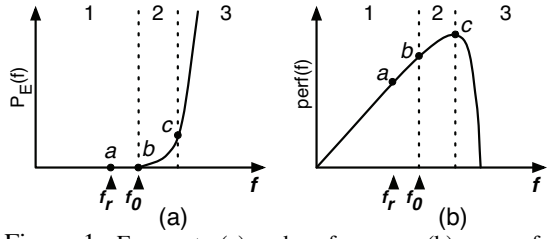


Figure 1: Error rate (a) and performance (b) versus frequency under TS.

Conventional processors work at point *a* in the figures, or at best at *b*. TS processors can work at *c*, therefore delivering higher single-thread performance.

2.1 Overview of TS Microarchitectures

A TS microarchitecture must maintain a high IPC at high frequencies with as small a recovery penalty as possible — all within the confines of power and area constraints. Unsurprisingly, differing design goals give rise to a diversity of TS microarchitectures. In the following, we group existing proposals into two broad categories.

2.1.1 Stage-Level TS Microarchitectures

Razor [5], TIMERRTOL [24], CTV [14], and X-Pipe [25] detect faults at pipeline-stage boundaries by comparing the values latched from speculatively-clocked logic to known good values generated by a checker. This checker logic can be an entire copy of the circuit that is safely clocked [14, 24]. A more efficient option, proposed in Razor [5], is to use a single copy of the logic to do both speculation and checking. This approach works by wave-pipelining the logic [4] and latching the output values of the pipeline stage twice: once in the normal pipeline latch, and a fraction of a cycle later in a *shadow latch*. The shadow latch is guaranteed to receive the correct value. At the end of each cycle, the shadow and normal latch values are compared. If they agree, no action is taken. Otherwise, the values in the shadow latches are used to repair the pipeline state.

Another stage-level scheme, Circuit Level Speculation (CLS) [9], accelerates critical blocks (rename, adder, and issue) by including a custom-designed speculative "approximation" version of each. For each approximation block, CLS also includes two fully correct checker instances clocked at half speed. Comparison occurs on the cycle after the approximation block generates its result, and recovery may involve re-issuing errant instructions.

2.1.2 Leader-Checker TS Microarchitectures

In CMPs, two cores can be paired in a leader-checker organization, with both running the same (or very similar) code, as in Slipstream [20], Paceline [6], Optimistic Tandem [11], and Reunion [18]. The leader runs speculatively and can relax functional correctness. The checker executes correctly and may be sped up by hints from the leader as it checks the leader's work.

Paceline [6] was designed specifically for TS. The leader is clocked at a frequency higher than the Limit Frequency f_0, while the checker is clocked at the Rated Frequency f_r. Paceline allows adjacent cores in the CMP to operate either as a pair (a leader with TS and a safe checker), or separately at f_r. In paired mode, the leader sends branch results to the checker and prefetches data into a shared L2, allowing the checker to keep up. The two cores periodically exchange checkpoints of architectural state. If they disagree, the checker copies its register state to the leader. Because the two cores are loosely coupled, they can be disconnected and used independently in workloads that demand throughput instead of response time.

One type of leader-checker microarchitecture sacrifices this configurability in pursuit of higher frequency by making the leader core functionally incorrect by design. Optimistic Tandem [11] achieves this by pruning infrequently-used functionality from the leader. DIVA [1] can also be used in this manner by using a functionally incorrect main pipeline. This approach requires the checker to be dedicated and always on.

3 Taxonomy of Design for TS

To understand the design space, we propose a taxonomy of design for TS from an architectural perspective. It consists of a classification of TS microarchitectures and of general approaches to enhance TS, and how they relate.

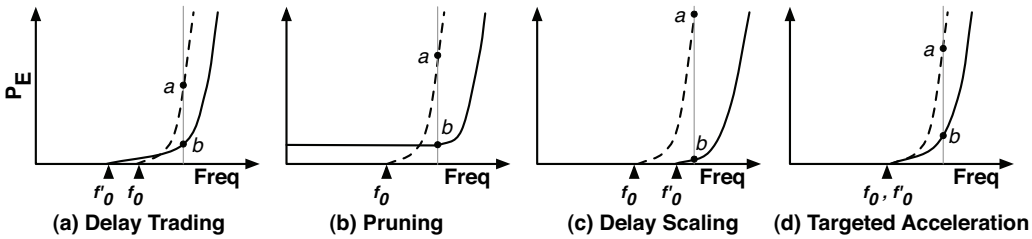


Figure 2: General approaches to enhance TS by reshaping the $P_E(f)$ curve. Each approach shows the curve before reshaping (in dashes) and after (solid), and the working point of a processor before (*a*) and after (*b*).

3.1 Classification of TS Microarchitectures

We classify existing proposals of TS microarchitectures according to: (1) whether the fault detection and correction hardware is always on (*Checker Persistence*), (2) whether functional correctness is sacrificed to maximize speedup regardless of the operating frequency (*Functional Correctness*), and (3) whether checking is done at pipeline-stage boundaries or upon retirement of one or more instructions (*Checking Granularity*). In the following, we discuss these axes. Table 1 classifies existing proposals of TS microarchitectures according to these axes.

Microarchitecture	Checker Persistence	Functional Correctness	Checking Granularity
Razor [5]	Always-on	Correct	Stage
Paceline [6]	On-demand	Correct	Retirement
X-Pipe [25]	Always-on	Correct	Stage
CTV [14]	Always-on	Correct	Stage
TIMERRTOL [24]	Always-on	Correct	Stage
CLS [9]	Always-on	Relaxed	Stage
Slipstream [20]	Always-on	Relaxed	Retirement
Optim. Tandem [11]	Always-on	Relaxed	Retirement
DIVA [1]	Always-on	Relaxed	Retirement

Table 1: Classification of existing proposals of TS microarchitectures.

3.1.1 Checker Persistence

The checker hardware that performs fault detection and correction can be kept *Always-on* or just *On-demand*. If single-thread performance is crucial all the time, the processor will always operate at a speculative frequency. Consequently, an Always-on checker suffices. This is the approach of most existing proposals. However, future CMPs must manage a mix of throughput- and latency-oriented tasks. To save power when executing throughput-oriented tasks, it is desirable to disable the checker logic and operate at f_r. We refer to schemes where the checker can be engaged and disengaged as On-demand checkers.

3.1.2 Functional Correctness

Relaxing functional correctness can lead to higher clock frequencies. This can be accomplished by not implementing rarely-used logic, such as in Optimistic Tandem [11] and CLS [9], by not running the full program, such as in Slipstream [20], or even by tolerating processors with design bugs, such as in DIVA [1]. These *Relaxed* schemes suffer from errors regardless of the clock frequency. This is in contrast to *Correct* schemes, which guarantee error-free operation at and below the Limit Frequency.

Relaxing functional correctness imposes a single (speculative) mode of operation, demanding an Always-on checker. Correctness at the Limit Frequency and below is a necessary condition for checker schemes based on wave pipelining [4] like Razor [5], or On-demand checker schemes like Paceline [6].

3.1.3 Checking Granularity

Checking can be performed at pipeline-stage boundaries (*Stage*) or upon retirement of one or more instructions (*Retirement*). In Stage schemes, speculative results are verified at each pipeline latch before propagating to the next stage. Because faults are detected within one cycle of their occurrence, the recovery entails, at worst, a pipeline flush. The small recovery penalty enables these schemes to deliver performance even at high fault rates. However, eager fault detection prevents them from exploiting *masking* across pipeline stages.

The alternative is to defer checking until retirement. In this case, because detection is delayed, and because recovery may involve heavier-weight operations, the recovery penalty is higher. On the other hand, Retirement schemes do not need to recover on faults that are microarchitecturally masked, and the loosely-coupled checker may be easier to build.

3.2 General Approaches to Enhance TS

Given a TS microarchitecture, Equation 1 shows that we can improve its performance by reducing $P_E(f)$. To accomplish this, we propose four general approaches. They are graphically shown in Figure 2. Each of the approaches is shown as a way of reshaping the original $P_E(f)$ curve of Figure 1(a) (now in dashes) into a more favorable one (solid). For each approach, we show that a processor that initially worked at point *a* now works at *b*, which has a lower P_E for the same *f*.

Delay Trading (Figure 2(a)) slows-down *infrequently-exercised* paths and uses the resources saved in this way to speed up *frequently-exercised* paths for a given design budget. This leads to a lower Limit Frequency f_0' when compared to the one in the base design f_0 in exchange for a higher frequency under TS.

Pruning or *Circuit-level Speculation* (Figure 2(b)) removes the infrequently-exercised paths from the circuit in order to speed-up the common case. For example, the carry chain of the adder is only partially implemented to reduce the response time for most input values [9]. Pruning results in a higher frequency for a given P_E, but sacrifices the ability to operate error-free at any frequency.

Delay Scaling (Figure 2(c)) and *Targeted Acceleration* (Figure 2(d)) speed-up paths and, therefore, shift the curve toward higher frequencies. The approaches differ in which paths are sped-up. Delay Scaling speeds-up largely all paths, while Targeted Acceleration targets the common-case paths. As a result,

TS Microarchitectural Characteristic	Implication on TS-Enhancing Approach
Checker Persistence	*Delay Trading* is undesirable with *On-demand* microarchitectures
Functional Correctness	*Pruning* is incompatible with *Correct* microarchitectures
Checking Granularity	All approaches are applied more aggressively to *Stage* microarchitectures

Table 2: How TS microarchitectural choices impact what TS-enhancing approaches are most appropriate.

while Delay Scaling always increases the Limit Frequency, Targeted Acceleration does not, as f_0' may be determined by the infrequently-exercised critical paths. However, Targeted Acceleration is more energy-efficient. Both approaches can be accomplished with techniques such as supply voltage scaling or body biasing [22].

The EVAL framework of Sarangi *et al*. [13] also pointed out that the error rate versus frequency curve can be reshaped. Their framework examined changing the curve as in the *Delay Scaling* and Targeted Acceleration approaches, which were called *Shift* and *Tilt*, respectively, to indicate how the curve changes shape.

3.3 Putting It All Together

The choice of a TS microarchitecture directly impacts which TS-enhancing approaches are most appropriate. Table 2 summarizes how TS microarchitectures and TS-enhancing approaches relate.

Checker Persistence directly impacts the applicability of Delay Trading. Recall that Delay Trading results in a lower Limit Frequency than the base case. This would force On-demand checking architectures to operate at a lower frequency in the non-TS mode than in the base design, leading to sub-optimal operation. Consequently, Delay Trading is undesirable with On-demand checkers.

The *Functional Correctness* of the microarchitecture impacts the applicability of Pruning. Pruning results in a non-zero P_E regardless of the frequency. Consequently, Pruning is incompatible with Correct TS microarchitectures, such as those based on wave pipelining (e.g., Razor) or on-demand checking (e.g., Paceline).

Checker Granularity dictates how aggressively any of the TS-enhancing approaches can be applied. An approach is considered more aggressive if it allows more errors at a given frequency. Since Stage microarchitectures have a smaller recovery penalty than Retirement ones, all the TS-enhancing approaches can be applied more aggressively to Stage microarchitectures.

4 Designing Processors for TS

Our goal is to design processors that are especially suited for TS. Based on the insights from the previous section, we propose: (1) a novel processor design methodology that we call *BlueShift* and (2) two techniques that, when applied under BlueShift, improve processor frequency. These two techniques are instantiations of the approaches introduced in Section 3.2. Next, we present BlueShift and then the two techniques.

4.1 The BlueShift Framework

Conventional design methods use timing analysis to identify the static critical paths in the design. Since these paths would determine the cycle time, they are then optimized to reduce their latency. The result of this process is that designs end up having a *critical path wall*, where many paths have a latency equal to or only slightly below the clock period.

We propose a different design method for TS processors, where it is fine if some paths take longer than the period. When these paths are exercised and induce an error, a recovery mechanism is invoked. We call the paths that take longer than the period *Overshooting* paths. They are not critical because they do not determine the period. However, they hurt performance in proportion to how often they are exercised and cause errors.

Consequently, a key principle when designing processors for TS is that, rather than working with static distributions of path delays, we need to work with *dynamic* distributions of path delays. Moreover, we need to focus on optimizing the paths that overshoot most frequently *dynamically* — by trying to reduce their latency. Finally, we can leave unoptimized many overshooting paths that are exercised only infrequently — since we have a fault correction mechanism.

BlueShift is a design methodology for TS processors that uses these principles. In the following, we describe how BlueShift identifies dynamic overshooting paths and its iterative approach to optimization.

4.1.1 Identifying Dynamic Overshooting Paths

BlueShift begins with a gate-level implementation of the circuit from a traditional design flow. A representative set of benchmarks is then executed on a simulator of the circuit. At each cycle of the simulation, BlueShift looks for latch inputs that change after the cycle has elapsed. Such endpoints are referred to as overshooting. As an example, Figure 3 shows a circuit with a target period of 500ns. The numbers on the nets represent their switching times on a given cycle. Note that a net may switch more than once per cycle. Since endpoints *X* and *Y* both transition after 500ns, they are designated as overshooting for this cycle. Endpoint *Z* has completed all of its transitions before 500ns, so it is non-overshooting for this cycle.

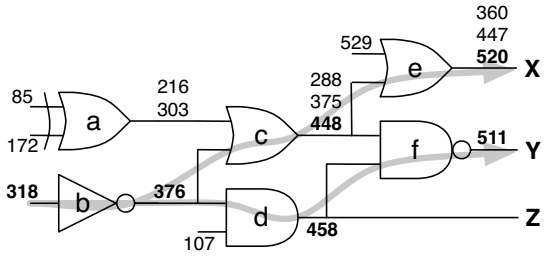


Figure 3: Circuit annotated with net transition times, showing two overshooting paths for this cycle.

Once the overshooting endpoints for a cycle are known, BlueShift determines the path of gates that produced their transitions. These are the overshooting paths for the cycle, and are the objects on which any optimization will operate. To identify these paths, BlueShift annotates all nets with their transition times. It then backtraces from each overshooting endpoint. As it backtraces from a net with transition time t_n, it locates the driving gate and its input whose transition at time t_i caused the change at t_n. For example, in Figure 3, the algorithm backtraces from *X* and finds the path $b \rightarrow c \rightarrow e$. Therefore, path $b \rightarrow c \rightarrow e$ is

overshooting for the cycle shown.

For each path p in the circuit, the analysis creates a set of cycles $D(p)$ in which that path overshoots. If N_{cycles} is the number of simulated cycles, we define the *Frequency of Overshooting* of path p as $d(p) = |D(p)|/N_{cycles}$. Then, the rate of errors per cycle in the circuit (P_E) is upper-bounded by $min(1, \sum_p d(p))$. To reduce P_E, BlueShift focuses on the paths with the *highest* frequency of overshooting first. Once enough of these paths have been accelerated and P_E drops below a pre-set target, optimization is complete; the remaining overshooting paths are ignored.

4.1.2 Iterative Optimization Flow

BlueShift makes iterative optimizations to the design, addressing the paths with the highest frequency of overshooting first. As the design is transformed, new dynamic overshooting paths are generated and addressed in subsequent iterations. This iterative process stops when P_E falls below target. Figure 4 illustrates the full process. It takes as inputs an initial gate-level design and the designer's target speculative frequency and P_E.

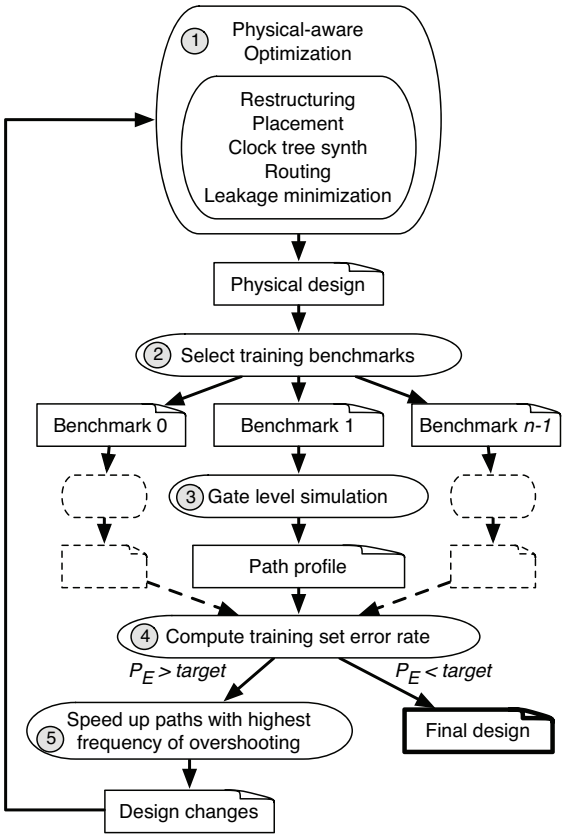


Figure 4: The BlueShift optimization flow.

At the head of the loop (Step 1), a physical-aware optimization flow takes a list of design changes from the previous iteration and applies them as it performs aggressive logical and physical optimizations. The output of Step 1 is a fully placed and routed physical design suitable for fabrication. Step 2 begins the embarrassingly-parallel profiling phase by selecting *n training* benchmarks. In Step 3, one gate-level timing simulation is initiated for each benchmark. Each simulation runs as many instructions as is economical and then computes the frequencies of overshooting for all paths exercised during the execution. Before Step 4, a global barrier waits for all of the individual simulations to finish. Then, the overall frequency of overshooting for each path is computed by averaging the measure for that path over the individual simulation instances. BlueShift also computes the average P_E across all simulation instances.

BlueShift then performs the exit test. If P_E is less than the designer's target, then optimization is complete; the physical design after Step 1 of the current iteration is ready for production. As a final validation, BlueShift executes another set of timing simulations using a different set of benchmarks (the *Evaluation* set) to produce the final P_E versus f curve. This is the curve that we use to evaluate the design.

If, on the other hand, P_E exceeds the target, we collect the set of paths with the highest frequency of overshooting, and use an optimization technique to generate a list of design changes to speed-up these paths (Step 5). Different optimization techniques can be used to generate these changes. We present two next.

4.2 Techniques to Improve Performance

To speed-up processor paths, we propose two techniques that we call *On-demand Selective Biasing (OSB)* and *Path Constraint Tuning (PCT)*. They are specific implementations of two of the general approaches to enhance TS discussed in Section 3.2, namely Targeted Acceleration and Delay Trading, respectively. We do not consider techniques for the other approaches in Figure 2 because a technique for Pruning was already proposed in [11] and Delay Scaling is a degenerate, less energy-efficient variant of Targeted Acceleration that lacks path targeting.

4.2.1 On-Demand Selective Biasing (OSB)

On-demand Selective Biasing (OSB) applies forward body biasing (FBB) [22] to one or more of the gates of each of the paths with the highest frequency of overshooting. Each gate that receives FBB speeds up, reducing the path's frequency of overshooting. With OSB, we push the P_E versus f curve as in Figure 2(d), making the processor faster under TS. However, by applying FBB, we also increase the leakage power consumed.

Figure 5(a) shows how OSB is applied, while Figure 5(b) shows pseudo code for the algorithm of Step 5 in Figure 4 for OSB. The algorithm takes as input a constant *k*, which is the fraction of all the dynamic overshooting in the design that will remain un-addressed after the algorithm of Figure 5(b) completes.

The algorithm proceeds as follows. At any time, the algorithm maintains a set of paths that are eligible for speedup (P_{elig}). Initially, at entry to Step 5 in Figure 4, Line 1 of the pseudo code in Figure 5(b) sets all the dynamic overshooting paths (P_{oversh}) to be eligible for speedup. Next, in Line 2 of Figure 5(b), a loop begins in which one gate will be selected in each iteration to receive FBB. In each iteration, we start by considering all paths p in P_{elig} weighted by their frequency of overshooting $d(p)$. We also define the weight of a gate g as the sum of the weights of all the paths in which it participates (`paths`(g)). Then, Line 3 of Figure 5(b) greedily selects the gate (g_{sel}) with the highest weight. Line 4 removes from P_{elig} all the paths in which the selected gate participates. Next, Line 5 adds the selected gate to the set of gates that will receive FBB (G_{FBB}). Finally, in Line 6, the loop terminates when the fraction of all the original dynamic overshooting that remains un-addressed is no higher than k.

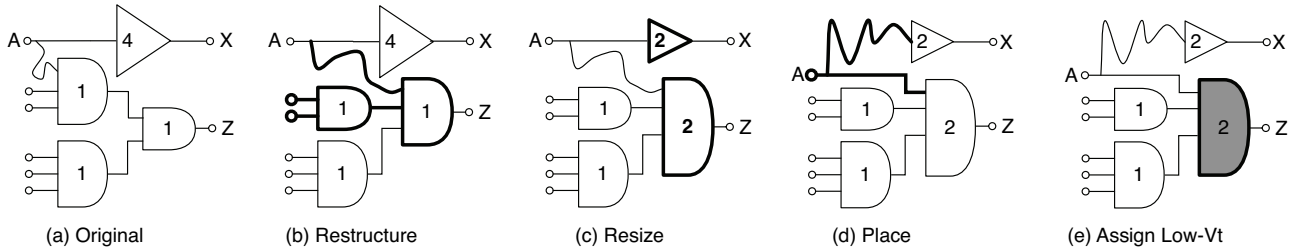
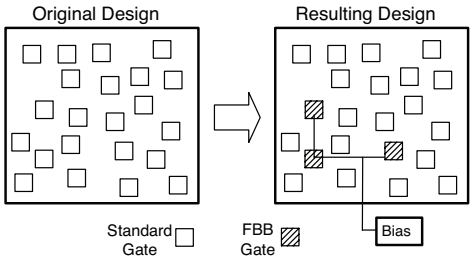


Figure 6: Transforming a circuit to reduce the delay of $A \rightarrow Z$ at the expense of that of the other paths. The numbers represent the gate size.



1 $\mathbf{P_{elig}} \leftarrow \mathbf{P_{oversh}}$
2 repeat
3 $\quad g_{sel} \leftarrow \underset{g}{\operatorname{argmax}} \sum_{p \in (\mathbf{P_{elig}} \cap \mathrm{paths}(g))} d(p)$
4 $\quad \mathbf{P_{elig}} \leftarrow \mathbf{P_{elig}} - \mathtt{paths}(g_{sel})$
5 $\quad \mathbf{G_{FBB}} \leftarrow \mathbf{G_{FBB}} + g_{sel}$
6 while $\sum_{p \in \mathbf{P_{elig}}} d(p) \Big/ \sum_{p \in \mathbf{P_{oversh}}} d(p) > k$

(b)

Figure 5: On-demand Selective Biasing (OSB): application to a chip (a) and pseudo code of the algorithm (b).

After this algorithm is executed in Step 5 of Figure 4, the design changes are passed to Step 1, where the physical design flow regenerates the netlist using FBB gates where instructed. In the next iteration of Figure 4, all timing simulations assume that those gates have FBB. We may later get to Step 5 again, in which case we will take the current dynamic overshooting paths and re-apply the algorithm. Note that the selection of FBB gates across iterations is monotonic; once a gate has been identified for acceleration, it is never reverted to standard implementation in subsequent iterations.

After the algorithm of Figure 4 completes, the chip is designed with body-bias signal lines that connect to the gates in G_{FBB}. The overhead of OSB is the extra static power dissipated by the gates with FBB and the extra area needed to route the body-bias lines and to implement the body-bias generator [22].

In TS architectures with On-demand checkers like Paceline [6] (Table 1), it is best to be able to disable OSB when the checker is not present. Indeed, the architecture without checker cannot benefit from OSB anyway, and disabling OSB also saves all the extra energy. Fortunately, this technique is easily and quickly disabled by removing the bias voltage. Hence the "on-demand" part of this technique's name.

4.2.2 Path Constraint Tuning (PCT)

Path Constraint Tuning (PCT) applies stronger timing constraints on the paths with the highest frequency of overshooting, at the expense of the timing constraints on the other paths. The result is that, compared to the period T_0 of a processor without TS at the Limit Frequency f_0, the paths that initially had the highest frequency of overshooting now take less than T_0, while the remaining ones take longer than T_0. PCT improves the performance of the common-case paths at the expense of the uncommon ones. With PCT, we change the P_E versus f curve as in Figure 2(a), making the processor faster under TS — although slower if it were to run without TS. This technique does not intrinsically have a power cost for the processor.

Existing design tools can transfer slack between connected paths in several ways, exhibited in Figure 6. The figure shows an excerpt from a larger circuit in which we want to speed up path $A \rightarrow Z$ by transferring slack from other paths. Figure 6(a) shows the original circuit, and following to the right are successive transformations to speed up $A \rightarrow Z$ at the expense of other paths. First, Figure 6(b) refactors the six-input AND tree to reduce the number of logic levels between *A* and *Z*. This transformation lengthens the paths that now have to pass through two 3-input ANDs. Figure 6(c) further accelerates $A \rightarrow Z$ by increasing the drive strength of the critical AND. However, we have to downsize the connected buffer to avoid increasing the capacitive load on *A* and, therefore, we slow down $A \rightarrow X$. Figure 6(d) refines the gate layout to shorten the long wire on path $A \rightarrow Z$ at the expense of lengthening the wire on $A \rightarrow X$. Finally, Figure 6(e) allocates a reduced-V_t gate (or an FBB gate) along the $A \rightarrow Z$ path. This speeds up the path but has a power cost, which may need to be recovered by slowing down another path.

The implementation of PCT is simplified by the fact that existing design tools already implement the transformations shown in Figure 6. However, they do all of their optimizations based on static path information. Fortunately, they provide a way of specifying "timing overrides" that increase or decrease the allowable delay of a specific path. PCT uses these timing overrides to specify timing constraints equal to the speculative clock period for paths with high frequency of overshooting, and longer constraints for the rest of the paths.

The task of Step 5 in Figure 4 for PCT is simply to generate a list of timing constraints for a subset of the paths. These constraints will be processed in Step 1. To understand the PCT algorithm, assume that the designer has a target period with TS equal to T_{ts}. In the first iteration of the BlueShift framework of Figure 4, Step 1 assigns a relaxed timing constraint to all paths. This constraint sets the path delays to $r \times T_{ts}$ (where *r* is a relax-

General Processor/System Parameters	
Width: 6-fetch 4-issue 4-retire OoO	L1 D Cache: 16KB WT, 2 cyc round trip, 4 way, 64B line
ROB: 152 entries	L1 I Cache: 16KB WB, 2 cyc round trip, 2 way, 64B line
Scheduler: 40 fp, 80 int	L2 Cache: 2MB WB, 10 cyc round trip (at Rated *f*), 8 way, 64B line,
LSQ Size: 54 LD, 46 ST	shared by two cores, has stride prefetcher
Branch Pred: 80Kb tournament	Memory: 400 cyc round trip (at Rated *f*), 10GB/s max
Paceline Parameters	**Razor Parameters**
Max Leader–Checker Lag: 512 instrs or 64 stores	Pipeline Fix and Restart Overhead: 5 cyc
Checkpoint Interval: 100 instrs	Total Target P_E: 10^{-3} err/cyc
Checkpoint Restoration Overhead: 100 cyc	
Total Target P_E: 10^{-5} err/cyc	

Table 3: Microarchitecture parameters.

ation factor), making them even longer than a period that would be reasonable without TS. When we get to Step 5, the algorithm first sorts all paths in order of descending frequency of overshooting at T_{ts}. Then, it greedily selects paths from this list leaving those whose combined frequency of overshooting is less than the target P_E. To these selected paths, it assigns a timing constraint equal to T_{ts}. Later, when the next iteration of Step 1 processes these constraints, it will ensure that these paths all fit within T_{ts}, possibly at the expense of slowing down the other paths.

At each successive iteration of BlueShift, Step 5 assigns the T_{ts} timing constraint to those paths that account for a combined frequency of overshooting greater than the target P_E at T_{ts}. Note that once a path is constrained, that constraint persists for all future BlueShift iterations. Eventually, after several iterations, a sufficient number of paths are constrained to meet the target P_E.

5 Experimental Setup

The PCT and OSB techniques are both applicable to a variety of TS microarchitectures. However, to focus our evaluation, we mate each technique with a single TS microarchitecture that, according to Section 3.3, emphasizes its strengths. Specifically, an Always-on checker is ideal for PCT because it lacks a non-speculative mode of operation, where PCT's longer worst-case paths would force a reduction in frequency. Conversely, an On-demand microarchitecture is suited to OSB because it does have a non-speculative mode where worst-case delay must remain short. Moreover, OSB is easy to disable. Finally, the PCT design, where TS is on all the time, targets a high-performance environment, while the OSB one targets a more power-efficient environment. Overall, we choose a high-performance Always-on Stage microarchitecture (Razor [5]) for PCT and a power-efficient On-demand Retirement one (Paceline [6]) for OSB. We call the resulting BlueShift-designed microarchitecures *Razor+PCT* and *Paceline+OSB* respectively.

Table 3 shows parameter values for the processor and system architecture modeled in both experiments. The table also shows Paceline and Razor parameters for the OSB and PCT evaluations, respectively. In all cases, only the core is affected by TS; the L2 and main memory access times remain unaffected.

5.1 Modeling

To accurately model the performance and power consumption of a gate-level BlueShifted processor running applications requires a complex infrastructure. To simplify the problem, we partition the modeling task into two loosely-coupled levels. The lower level comprises the BlueShift circuit implementation, while the higher level consists of microarchitecture-level power and performance estimation.

At the circuit-modeling level, we sample modules from the OpenSPARC T1 processor [19], which is a real, optimized, industrial design. We apply BlueShift to these modules and use them to compute P_E and power estimates before and after BlueShift. At the microarchitecture level, we want to model a more sophisticated core than the OpenSPARC. To this end, we use the SESC [12] cycle-level execution-driven simulator to model the out-of-order core of Table 3.

The difficulty lies in incorporating the circuit-level P_E and power estimates into the microarchitecural simulation. Our approach is to assume that the modules from the OpenSPARC are representative of those in any other high-performance processor. In other words, we assume that BlueShift would induce roughly the same P_E and power characteristics on the out-of order microarchitecture that we simulate as it does on the in-order processor that we can measure directly.

In the following subsections, we first describe how we generate the BlueShifted circuits. We then show how P_E and power estimates are extracted from these circuits and used to annotate the microarchitectural simulation.

5.1.1 BlueShifted Module Implementation

To make the level of effort manageable, we focus our analysis on only a few modules of the OpenSPARC core. The chosen modules are sampled from throughout the pipeline, and are shown in Table 4. Taken together, these modules provide a representative profile of the various pipeline stages. For each module, the *Stage* column of the table shows where in the pipeline (Fetch/Decode, EXEcute, or MEMory) the module resides. The next two columns show the size in number of standard cells and the shortest worst-case delay attained by the traditional CAD flow without using any low-V_t cells (which consume more power).

The next two columns show the per-module error rate targets under PCT and OSB. This is the P_E that BlueShift will try to ensure for each module. We obtain these numbers by apportioning a "fair share" of the total processor P_E to each module — roughly according to its size. With these P_E targets, when the full pipeline is assembled (including modules not in the sample set), the total processor P_E will be roughly 10^{-3} errors/cycle for PCT and 10^{-5} for OSB. These were the target total P_E numbers in Table 3. They are appropriate for the average recovery overhead of the corresponding architectures: 5 cycles for Razor (Table 3) and about 1,000 cycles for Paceline (which include 100 cycles spent in checkpoint restoration as per Table 3). Indeed, with these values of P_E and recovery overhead, the total performance lost in recovery is 1% or less.

The largest and most complex module is *sparc_exu*. It contains

Module Name	Stage	Num. Cells	T_r (ns)	Target P_E (Errors/Cycle) PCT	OSB	Description
sparc_exu	EXE	21,896	1.50	10^{-4}	10^{-6}	Integer FUs, control, bypass
lsu_stb_ctl	MEM	765	1.11	10^{-5}	10^{-7}	Store buffer control
lsu_qctl1	MEM	2,336	1.50	10^{-5}	10^{-7}	Load/Store queue control
lsu_dctl	MEM	3,682	1.00	10^{-5}	10^{-7}	L1 D-cache control
sparc_ifu_dec	F/D	765	0.75	10^{-5}	10^{-7}	Instruction decoder
sparc_ifu_fdp	F/D	7,434	0.94	10^{-5}	10^{-7}	Fetch datapath and PC maintenance
sparc_ifu_fcl	F/D	2,299	0.96	10^{-5}	10^{-7}	L1 I-cache and PC control

Table 4: OpenSPARC modules used to evaluate BlueShift.

Feature size	130nm scaled to 32nm
Metal	7 layers
T_{max}	$100°C$
Low-V_t devices	10x leakage; 0.8x delay
f guardband	10%

Table 5: Process parameters.

# Benchmarks run per iteration	200 (PCT) 400 (OSB)
# Cycles per benchmark	25K
r: PCT relaxation factor	1.5
k: Fraction of all the dynamic overshooting that remains un-addressed after each OSB iteration of Figure 4	0.01

Table 6: BlueShift parameters.

the integer register file, the integer arithmetic and logic datapaths along with the address generation, bypass and control logic. It also performs other control duties including exception detection, save/restore control for the SPARC register windows, and error detection and correction using ECC. This module alone is larger than many lightweight embedded processor cores.

Using Synopsis Design Compiler 2007.03 and Cadence Encounter 6.2, we perform full physical (placed and routed) implementations of the modules in Table 4 for the standard cell process described in Table 5. To make the results more accurate for a near-future (e.g., 32nm) technology, we scale the cell leakage so that it accounts for ≈30% of the total power consumption. The process has a 10% guardband to tolerate environmental and process variations. This means that $f_0 = 1.10 \times f_r$, where f_r and f_0 are the Rated and Limit Frequencies, respectively. The process also contains low-V_t gates that have a 10x higher leakage and a 20% lower delay than normal gates [15, 23]. These gates are available for assignment in high-performance environments such as those with Razor. Finally, the FBB gates used in OSB are electrically equivalent to low-V_t gates when FBB is enabled and to standard V_t gates when it is not.

Table 6 lists the BlueShift parameters. In the *Razor+PCT* experiments, we add hold-time delay constraints to the paths[1] to accommodate shadow latches. Moreover, shadow latches are inserted wherever worst-case delays exceed the speculative clock period. Each profiling phase (Step 2 of Figure 4) comprises a parallel run of 200 (or 400 for OSB) benchmark samples, each one running for 25K cycles.

We use the unmodified RTL sources from OpenSPARC, but we simplify the physical design by modeling the register file and the 64-bit adder as black boxes. In a real implementation, these components would be designed in full-custom logic. We use timing information supplied with the OpenSPARC to build a detailed 900MHz black box timing model for the register file; then, we use CACTI [21] to obtain an area estimate and build a realistic physical footprint. The 64-bit adder is modeled on [27], and has a worst-case delay of 500ns.

Although we find that BlueShift is widely applicable to logic modules, it is not effective on array structures where all paths are exercised with approximately equal frequency. As a result, we classify caches, register files, branch predictor, TLBs, and other memory blocks in the processor as *Non-BlueShiftable*. We assume that these modules attain performance scaling without timing errors through some other method (e.g. increased supply voltage) and account for the attendant power overhead.

[1]For some modules, the commercial design tools that we use are unable to meet the minimum path delay constraints, but we make a best effort to honor them.

5.1.2 Module-Level P_E and Power

For each benchmark, we use Simics [10] to fast-forward execution over 1B cycles, then checkpoint the state and transfer the checkpoint to the gate-level simulator. To perform the transfer, we use the CMU Transplant tool [17]. This enables us to execute many small, randomly-selected benchmark samples in gate-level detail. Further, only the gate-level modules from Table 4 need to be simulated at the gate level. Functional, RTL-only simulation suffices for the remaining modules of the processor.

The experiments use SPECint2006 applications as the *Training* set in the BlueShift flow (Steps 1–5 of Figure 4). After BlueShift terminates, we measure the error rate for each module using SPECint2000 applications as the *Evaluation* set. From the latter measurements, we construct a P_E versus f curve for each SPECint2000 application on each module. All P_E measurements are recorded in terms of *the fraction of cycles on which at least one latch receives the wrong value*. This is an accurate strategy for the Razor-based evaluation, but because it ignores architectural and microarchitectural masking across stages, it is highly pessimistic for Paceline.

Circuit-level power estimation for the sample modules is done using Cadence Encounter. We perform detailed capacitance extraction and then use the tool's default leakage and switching analysis.

5.1.3 Microarchitecture-Level P_E and Power

We compute the performance and power consumption of the Paceline- and Razor-based microarchitectures using the SESC [12] simulator, augmented with Wattch [3], HotLeakage [26], and HotSpot [16] power and temperature models. For evaluation, we use the SPECint2000 applications, which were also used to evaluate the per-module P_E in the preceding section. The simulator needs only a few key parameters derived from the low-level circuit analysis to accurately capture the P_E and power impact of BlueShift.

Module	*Paceline Base*		*Paceline+OSB*		*Razor Base*		*Razor+PCT*	
	P_{sta} (mW)	E_{dyn} (pJ)	P_{sta} (mW)	E_{dyn} (pJ)	P_{sta} (mW)	E_{dyn} (pJ)	P_{sta} (mW)	E_{dyn} (pJ)
sparc_exu	68.5	207.8	75.1	207.8	175.1	217.2	130.3	257.8
lsu_stb_ctl	2.1	5.6	2.1	5.6	4.3	6.0	3.9	9.5
lsu_qctl1	4.7	18.8	4.7	18.8	12.4	18.8	15.4	35.9
lsu_dctl	8.8	33.3	9.2	33.3	20.7	35.1	21.3	54.5
sparc_ifu_dec	2.1	1.4	3.3	1.4	5.9	3.9	5.1	5.3
sparc_ifu_fdp	21.7	117.6	23.8	117.6	36.1	119.6	30.0	146.6
sparc_ifu_fcl	5.8	15.7	6.4	15.7	16.5	15.7	10.6	24.3
Total	113.7	400.3	124.7	400.3	271.0	416.1	216.6	533.7

Table 7: Static power consumption (P_{sta}) and switching energy per cycle (E_{dyn}) for each module implementation.

To estimate the P_E for the entire pipeline, we first sum up the P_E from all of the sampled modules of Table 4. Then, we take the resulting P_E and scale it so that it also includes the estimated contribution of all the other BlueShiftable components in the pipeline. We assume that the P_E of each of these modules is roughly proportional to the size of the module. Note that by adding up the contributions of all the modules, we are assuming that the pipeline is a series-failure system with independent failures, and that there is no error masking across modules. The result is a whole-pipeline P_E versus frequency curve for each application. We use this curve to initiate error recoveries at the appropriate rate in the microarchitectural simulator.

For power estimation, we start with the dynamic power estimations from Wattch for the simulated pipeline. We then scale up these *Raw* power numbers to take into account the higher power consumption induced by the BlueShift optimization. The scale factor is different for the BlueShiftable and the Non-BlueShiftable components of the pipeline. Specifically, we first measure the dynamic power consumed in all of the sampled OpenSPARC modules as given by Cadence Encounter. The ratio of the power after BlueShift over the power before BlueShift is the factor that we use to scale up the Raw power numbers in the BlueShiftable components. For the Non-BlueShiftable components, we first compute the increase in supply voltage that is necessary for them to keep up with the frequency of the rest of the pipeline, and then scale their Raw power numbers accordingly.

For the static power, we use a similar approach based on HotLeakage and Cadence Encounter. However, we modify the HotLeakage model to account for the differing numbers of low-V_t gates in each environment of our experiments.

As a thermal environment, microarchitectural power simulations assume a 16-core, 32nm CMP with half of the cores idle. Maximum temperature constraints are enforced.

6 Evaluation

For each of the *Paceline+OSB* and *Razor+PCT* architectures, this section estimates the whole-pipeline $P_E(f)$ curve, the performance, and the total power.

6.1 Implementations of the Pipeline Modules

Our evaluation uses four different implementations of the modules in Table 4. The *Paceline Base* implementation uses a traditional CAD flow to produce the fastest possible version of each module without using any low-V_t gates. We choose this leakage-efficient implementation because our Paceline-based design points target a power-efficient environment (Section 5). This implementation, when used in an environment where the two cores in Paceline are decoupled [6], provides the normalized basis for the frequency and performance results in this paper. Specifically, a frequency of 1 corresponds to the Rated Frequency of the *Paceline Base* implementation, and a speedup of 1 corresponds to the performance of this implementation when the cores are decoupled.

If we run the *Paceline Base* design through the BlueShift OSB flow targeting a 20% frequency increase for all the modules (at the target P_E specified in Table 4), we obtain the *Paceline+OSB* implementation. Note that, in this implementation, if we disable the body bias, we obtain the same performance and power as in *Paceline Base*.

The PCT evaluation with Razor requires the introduction of another non-BlueShifted implementation. Since our Razor-based design points target a high-performance environment (Section 5), we use an aggressive traditional CAD flow that is allowed unrestricted use of low-V_t gates (although the tools are still instructed to minimize leakage as much as possible). Because of the aggressive use of low-V_t devices, the modules in this implementation reach a worst-case timing that is 15% faster than *Paceline Base*. We then apply Razor to this implementation and call the result *Razor Base*.

Finally, we use the BlueShift PCT flow targeting a 30% frequency increase over *Paceline Base* for all modules — again at the target P_E specified in Table 4. This implementation of the modules also includes Razor latches. We call it *Razor+PCT*.

Each implementation offers a different tradeoff between dynamic and static power consumption. Table 7 shows the static power at 85°C (P_{sta}) and the average switching energy per cycle (E_{dyn}) consumed by each module under each implementation. As expected, *Paceline Base* consumes the least power and energy. Next, *Paceline+OSB* has only slightly higher static power.

The two Razor-based implementations have higher static power consumption, mostly due to their heavier use of low-V_t devices. In *Razor Base* and *Razor+PCT*, the fraction of low-V_t gates is 11% and 5%, respectively. Additionally, the Razor-based implementations incur power overhead from Razor itself. This overhead is more severe in *Razor+PCT* than in *Razor Base* for two reasons. First, note that any latch endpoint that can exceed the speculative clock period requires a shadow latch. After PCT-induced path relaxation, the probability of an endpoint having such a long path increases, so more Razor latches are required. Second, *Razor+PCT* requires more hold-time fixing. This is because we diverge slightly from the original Razor proposal [5] and assume that the shadow latches are clocked a constant delay after the main edge — rather than a constant phase difference. With PCT-induced path relaxation, the difference between the long and

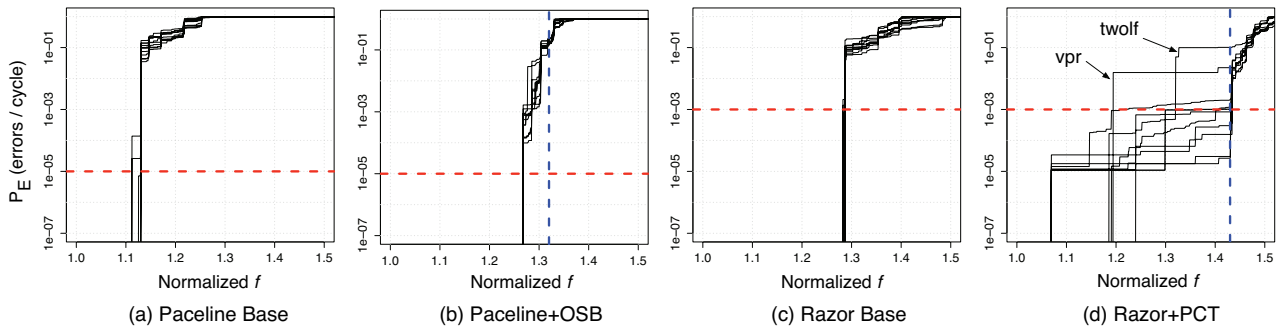


Figure 7: Whole-pipeline $P_E(f)$ curves for the four implementations. The frequencies are given relative to the Rated Frequency (f_r) of *Paceline Base*.

short path delay increases, and the delay between the shadow and main latch clock must be increased. This requires more buffers on the short paths to guarantee sufficient shadow hold time.

6.2 Error Curve Transformations

We now consider the resulting whole-pipeline $P_E(f)$ curves. Figures 7(a)-(d) show the curves for each of the four implementations. These curves do not include the effect of the non-BlueShiftable modules. In each plot, the x axis shows the frequency relative to the Rated Frequency (f_r) of *Paceline Base*. Each plot has one curve for each SPECint2000 application. In addition, there is a horizontal dashed line that marks the whole-pipeline target P_E, namely 10^{-5} errors per cycle for the Paceline-based environments and 10^{-3} for the Razor-based ones.

Figure 7(a) shows the curve for *Paceline Base*. Since we use a 10% guardband (Table 5), the Limit Frequency f_0 is at 1.1 in the plot. As we increase f, P_E takes non-zero values past f_0 (although it is invisible in the figure) and quickly reaches high values. This is due to the critical path wall of conventional designs. Consequently, using TS on a non-BlueShift design can only manage frequencies barely above f_0 before P_E becomes prohibitive.

Figure 7(b) shows the curve for *Paceline+OSB*. This plot follows the *Targeted Acceleration* shape in Figure 2. Specifically, P_E starts taking non-zero values past f_0 like in *Paceline Base* — a static timing analysis shows that the worst-case delays are unchanged from *Paceline Base*. However, the rise in P_E is delayed until higher frequencies. Indeed, P_E remains negligible until a relative frequency of 1.27, compared to about 1.11 in *Paceline Base*. This shows that the application of BlueShift with OSB enables an increase in processor frequency of 14%.

Figure 7(b) also shows a dashed vertical line. This was OSB's frequency target, namely a 20% increase over *Paceline Base* (Section 6.1) — or 1.2×1.1=1.32 after the guardband. However, we see that OSB did not meet its target. This is because the Training application set (which was used in the optimization algorithm of Figure 4) failed to capture some key behavior of the Evaluation set (which was used to generate the P_E curves). Consequently, *Paceline+OSB*, like other TS microarchitectures, will rely on its control mechanism to operate at the frequency that maximizes performance (1.27 in this case) rather than at its target. While higher frequencies may be possible with more comprehensive training, the obtained 14% frequency increase is substantial.

Figure 7(c) shows the curve for *Razor Base*. Since this is not a BlueShifted design, it exhibits a rapid P_E increase as in *Paceline Base*. The difference here is that, because it targets a high-performance (and power) design point, it attains a higher frequency than *Paceline Base*.

Finally, Figure 7(d) shows the curve for *Razor+PCT*. The plot follows the *Delay Trading* shape in Figure 2. Specifically, P_E starts taking non-zero values at lower frequencies than in *Paceline Base*, but the curve rises more gradually than in *Paceline+OSB*. The dashed vertical line shows the target frequency, which was 30% higher than *Paceline Base* (Section 6.1) — or 1.3×1.1=1.43 after the guardband. We can see that most applications reach this frequency at the whole-pipeline target P_E. Compared to the frequency of 1.28 attained by *Razor Base*, this means that BlueShift with PCT enables an increase in processor frequency of 12%. The two exceptions are the *twolf* and *vpr* applications, which fail to meet the target P_E due to discrepancies between the Training and Evaluation application sets. For these applications, the *Razor+PCT* architecture will adapt to run at a lower frequency, so as to maximize performance.

6.3 Paceline+OSB Performance and Power

We compare three Paceline-based architectures. First, *Unpaired* uses the *Paceline Base* module implementation and one core runs at the Rated Frequency while the other is idle. Secondly, *Paceline Base* uses the *Paceline Base* module implementation and the cores run paired under Paceline. Finally, *Paceline+OSB* uses the *Paceline+OSB* module implementation and the cores run paired under Paceline. For each application, *Paceline Base* runs at the frequency that maximizes performance. For the same application, *Paceline+OSB* runs at the frequency that maximizes performance considering only the P_E curves of the BlueShiftable components; then, we apply traditional voltage scaling to the non-BlueShiftable components so that they can catch up — always subject to temperature constraints.

Figure 8(a) shows the speedup of the *Paceline Base* and *Paceline+OSB* architectures over *Unpaired* for the different applications. We see that *Paceline+OSB* delivers a performance that is, on average, 8% higher than that of *Paceline Base*. Therefore, the impact of BlueShift with OSB is significant. The figure also shows that, on average, *Paceline+OSB* improves the performance by 17% over *Unpaired*. Finally, given that all applications cycle at approximately the same frequency for the same architecture (Figures 7(a) and 7(b)), the difference in performance across applications is largely a function of how well individual applications work under Paceline. For example, applications with highly-predictable branches such as *vortex* cause the checker to be a bottleneck and, therefore, the speedups in Figure 8(a) are small.

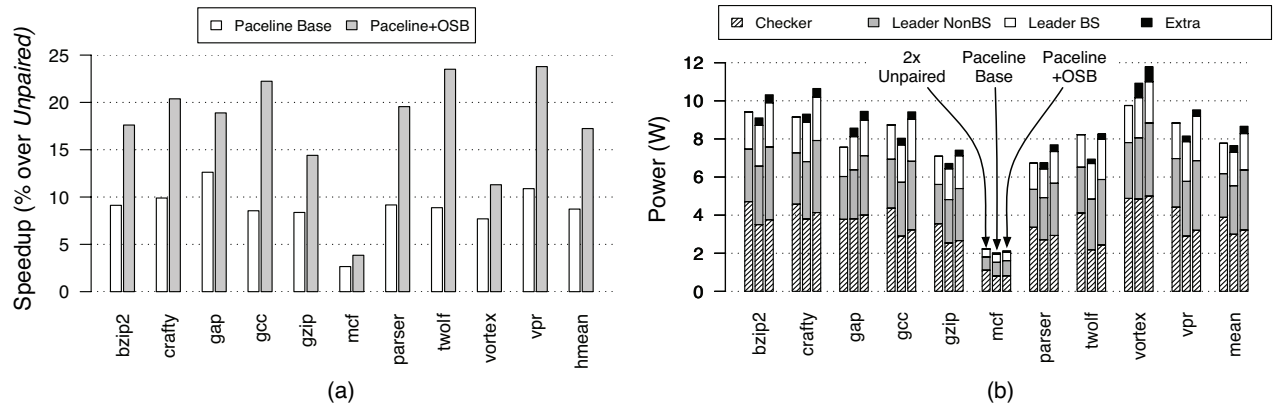


Figure 8: Performance (a) and power consumption (b) of different Paceline-based processor configurations. *BS* and *NonBS* refer to BlueShiftable and non-BlueShiftable modules, respectively.

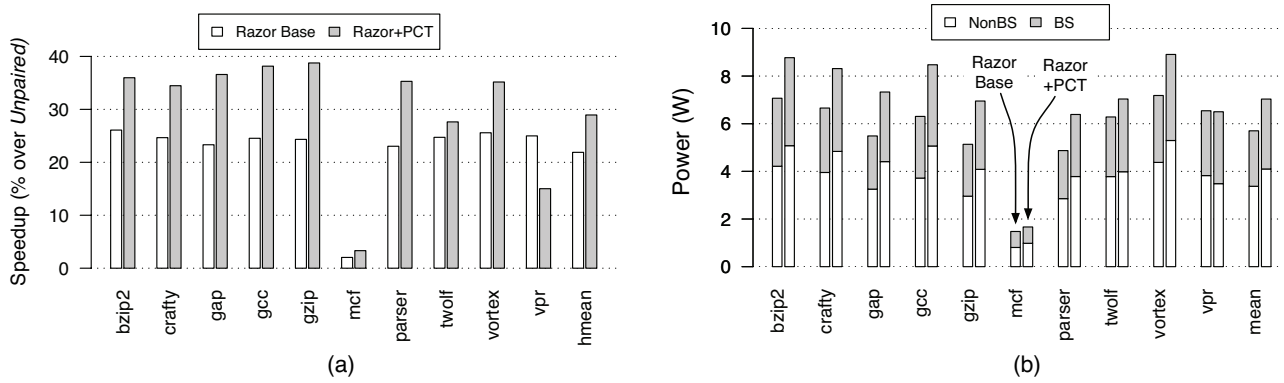


Figure 9: Performance (a) and power consumption (b) of different Razor-based processor configurations.

Figure 8(b) shows the power consumed by the processor and L1 caches in *Paceline Base*, *Paceline+OSB*, and two instances of *Unpaired*. The power is broken down into power consumed by the checker core (which is never BlueShifted), non-BlueShiftable modules in the leader, BlueShiftable modules in the leader, and extra Paceline structures (checkpointing, VQ, and BQ). On average, the power consumed by *Paceline+OSB* is 12% higher than that of *Paceline Base*. Consequently, BlueShift with OSB does not add much to the power consumption while delivering a significant performance gain. Note that the checker power is generally lower when the cores run in paired mode. This is because the checker core saves energy by skipping the execution of many wrong-path instructions.

6.4 Razor+PCT Performance and Power

We now compare two Razor-based architectures. *Razor Base* uses the *Razor Base* module implementation, while *Razor+PCT* uses the *Razor+PCT* one obtained by applying BlueShift with PCT. As before, *Razor+PCT* runs at the frequency given by the P_E curves of the BlueShiftable components; then, we apply traditional voltage scaling to the non-BlueShiftable components so that they can catch up.

Figure 9(a) shows the speedup of the *Razor Base* and *Razor+PCT* architectures over the *Unpaired* one used as a baseline in Figure 8(a). Since these Razor-based architectures target high performance, they deliver higher speedups. We see that, on average, *Razor+PCT*'s performance is 6% higher than that of *Razor Base*. This is the impact of BlueShift with PCT in this design — which is not negligible considering that *Razor Base* was already designed for high performance. We also see that *vpr* and, to a lesser extent, *twolf* do not perform as well as the other applications under *Razor+PCT*. This is the result of the unfavorable P_E curve for these applications in Figure 7(d).

Figure 9(b) shows the power consumed by the two processor configurations. The power is broken down into the contributions of the non-BlueShiftable and the BlueShiftable modules. On average, *Razor+PCT* consumes 23% more power than *Razor Base*. This is because it runs at a higher frequency, uses a higher supply voltage for the non-BlueShiftable modules, and needs more shadow latches and hold-time buffers.

Given *Razor+PCT*'s delivered speedup and power cost, we see that BlueShift with PCT is not compelling from an $E \times D^2$ perspective. Instead, we see it as a technique to further speed-up a high-performance design (at a power cost) when conventional techniques such as voltage scaling or body biasing do not provide further performance. Specifically, for *logic* (i.e., BlueShiftable) modules, BlueShift with PCT provides an *orthogonal* means of improving performance when further voltage scaling or body biasing becomes unfeasible. In this case however, for the pipeline as a whole, non-BlueShiftable stages remain a bottleneck that must be addressed using some other technique.

6.5 Computational Overhead

Although most modules of Table 4 were fully optimized with BlueShift in one day on our 100-core cluster, the optimization of *sparc_exu* took about one week. Such long turnaround times dur-

ing the frantic timing closure process would be unacceptable in industry. Fortunately, the current implementation is only a prototype, and drastic improvements in runtime are possible. Specifically, referring to Figure 4, a roughly equal amount of wall time is spent in physical implementation (Step 1) and profiling (Step 3). Luckily, the profiling phase is embarrassingly parallel, so simply adding more processors can speed it up. However, the CAD tools in Step 1 are mostly sequential. To reduce the overall runtime, the number of BlueShift iterations in Figure 4 must be reduced. This can be done, for example, by adding more constraints at each iteration. In practice, our experiments included few constraints per iteration to avoid overloading the commercial CAD tools, which have a tendency to crash if given too many constraints.

7 Other Related Work

Several proposals have analyzed or improved the performance of specific functional blocks, such as adders, under TS [2, 7]. However, work on general-purpose profile-driven design flows [2] is just beginning. Of existing work, Optimistic Tandem [11] is most closely related to BlueShift. It uses a profile-based approach to select infrequently-used RTL statements to prune from the design. Also related, BTWMap [8] is a gate-mapping algorithm (part of the logic synthesis flow) that uses a switching activity profile to minimize the common-case delay.

8 Conclusion

Timing Speculation (TS) is a promising technique for boosting single-thread performance. In this paper, we made three contributions related to TS. First, we introduced BlueShift, a new design approach and optimization algorithm where the processor is designed from the ground up for TS. The idea is to identify and optimize the most frequently-exercised critical paths in the design, at the expense of the majority of the static critical paths. The second contribution was two techniques that, when applied under BlueShift, improve processor performance: On-demand Selective Biasing (OSB) and Path Constraint Tuning (PCT). These techniques target the most frequently-exercised critical paths, and either add forward body bias to some of their gates or apply strong timing constraints on them. The third contribution was a taxonomy of design for TS.

We applied BlueShift with OSB and PCT on modules of the OpenSPARC T1 processor. Compared to a conventional Paceline implementation, BlueShift with OSB sped up applications by an average of 8% while increasing the processor power by an average of 12%. Moreover, compared to a high-performance conventional Razor implementation, BlueShift with PCT sped up applications by an average of 6% with an average processor power overhead of 23%. These figures assume that traditional voltage scaling is applied to non-BlueShiftable components. However, BlueShift provides a new way to speed up logic modules that is *orthogonal* to voltage scaling. We are currently extending our work to show the potential of combining both techniques.

References

[1] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *International Symposium on Microarchitecture*, May 1999.

[2] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst case design. In *Asia-South Pacific Design Automation Conference*, January 2005.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[4] W. P. Burleson, M. Ciesielski, F. Klass, and W. Liu. Wave-pipelining: A tutorial and research survey. *IEEE Transactions on VLSI Systems*, 6(3), September 1998.

[5] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Zeisler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *International Symposium on Microarchitecture*, December 2003.

[6] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale CMPs through core overclocking. In *International Conference on Parallel Architecture and Compilation Techniques*, September 2007.

[7] R. Hegde and N. Shanbhag. Soft digital signal processing. *IEEE Transactions on VLSI Systems*, 9(6), December 2001.

[8] J. Kong and K. Minkovich. Mapping for better than worst-case delays in LUT-based FPGA designs. In *International Symposium on Field Programmable Gate Arrays*, February 2008.

[9] T. Liu and S-L. Lu. Performance improvement with circuit-level speculation. In *International Symposium on Microarchitecture*, December 2000.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[11] F. Mesa-Martinez and J. Renau. Effective optimistic-checker tandem core design through architectural pruning. In *International Symposium on Microarchitecture*, December 2007.

[12] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. R. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[13] S. R. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. EVAL: Utilizing processors with variation-induced timing errors. In *International Symposium on Microarchitecture*, November 2008.

[14] T. Sato and I. Arita. Constructive timing violation for improving energy efficiency. *Compilers and operating systems for low power*. Kluwer Academic Publishers, 2003.

[15] G. Sery, S. Borkar, and V. De. Life is CMOS: Why chase the life after. In *Design Automation Conference*, June 2002.

[16] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *International Symposium on Computer Architecture*, June 2003.

[17] J. Smolens and E. Chung. Architectural transplant, 2007. http://transplant.sunsource.net/.

[18] J. Smolens, B. Gold, B. Falsafi, and J. Hoe. Reunion: Complexity-effective multicore redundancy. In *International Symposium on Microarchitecture*, December 2006.

[19] Sun Microsystems. OpenSPARC T1 RTL release 1.5. http://www.opensparc.net/opensparc-t1/index.html.

[20] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[21] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, April 2008.

[22] J. Tschanz, J. Kao, S. Narendra, R. Nair, D. Antoniadis, A. Chandrakasan, and V. De. Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid State Circuits*, 37(11), November 2002.

[23] S. Tyagi et al. A 130nm generation logic technology featuring 70nm transistors, dual Vt transistors and 6 layers of Cu interconnects. In *IEEE Electron Devices Meeting*, December 2000.

[24] A. Uht. Achieving typical delays in synchronous systems via timing error toleration. Technical Report 032000-0100, University of Rhode Island Department of Electrical and Computer Engineering, March 2000.

[25] X. Vera, O. Unsal, and A. Gonzalez. X-Pipe: An adaptive resilient microarchitecture for parameter variations. In *Workshop on Architectural Support for Gigascale Integration*, June 2006.

[26] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, University of Virginia, March 2003.

[27] R. Zlatanovici and B. Nikolic. Power–performance optimal 64-bit carry-lookahead adders. In *European Solid State Circuits Conference*, September 2003.