Compiling for Energy Efficiency on Timing Speculative Processors

John Sartori and Rakesh Kumar
University of Illinois at Urbana-Champaign

ABSTRACT

Timing speculation is a promising technique for improving microprocessor yield, in field reliability, and energy efficiency. Previous evaluations of the energy efficiency benefits of timing speculation have either been based on code compiled for a traditional target [2] – a processor that produces no errors, or code that relies on additional hardware support [6]. In this paper, we advocate that binaries for timing speculative processors should be optimized differently than those for conventional processors to maximize the energy benefits of timing speculation. Since the program binary determines the utilization pattern of the processor, which in turn influences the error rate of the processor and the energy efficiency of timing speculation, binary optimizations for timing speculative processors should attempt to manipulate the utilization of different microarchitectural units based on their likelihood of causing errors. An exploration of targeted and standard compiler optimizations demonstrates that significant energy benefits are possible from TS-aware binary optimization.

Categories and Subject Descriptors
D.3.4 [Processors]: Compilers, **General Terms:** Design
Keywords: error resilience, binary optimization, computer architecture, energy efficiency, timing speculation

1. INTRODUCTION

Timing speculation [2] is a promising technique for improving microprocessor yield, in field reliability, and energy efficiency. In the most common usage model, timing speculation involves relaxing voltage or frequency guardbands to improve energy efficiency at the expense of timing errors. Errors are corrected or tolerated by a hardware or software error resilience mechanism [2] to maintain acceptable output quality.

Previous evaluations of the energy efficiency benefits of timing speculation have either been based on code compiled for a traditional target [2] – a processor that produces no errors – or code that relies on instruction set extensions and additional hardware support [6]. For example, [6] advocates the use of instruction set extensions whose circuit implementations have shorter critical paths. Unfortunately, physical design tools render most pipeline stages critical in power-optimized processors [8, 11], reducing the effectiveness of such approaches. Also, instruction set extensions may not be feasible in many settings.

In this paper, we make a case for compiling differently for timing speculative processors in a way that increases energy efficiency without additional hardware support or instruction set extensions. To motivate our approach, we first explain the nature of benefits afforded by timing speculation (TS). The magnitude of energy efficiency benefits available from exploiting TS depends on two factors – (a) *where* and (b) *how often* the processor produces errors when operating at an overscaled voltage or frequency. (For more details, see supplemental Section S1.) The path slack distribution of a timing speculative processor determines which paths do not meet timing constraints (negative slack paths) and thus cause errors when they are toggled. Likewise, the activity distribution of the processor describes how often paths are toggled, and thus determines the frequency of errors caused by a path when it has negative slack. Together, the slack and activity distributions dictate the error distribution of a processor, i.e., the locations and frequencies of errors produced in an overscaled processor – i.e., a processor operating below nominal voltage or above nominal frequency.

Altering the error distribution of a timing speculative processor has the potential to increase the energy benefits from exploiting error resilience. For example, previous works have demonstrated that modifying the slack distribution (*where* errors are produced) can increase the energy efficiency of a timing speculative design [4, 9, 10, 12]. In this paper, we focus on the activity distributions (*how often* errors are produced) of timing speculative processors and make a case for *timing speculation-aware binary optimization*. Since the program binary, in conjunction with the processor architecture, determines a processor's activity distribution, optimizing a program binary for timing speculative processors can manipulate the utilization of different microarchitectural units based on their timing slack distribution to deliver energy efficiency benefits. For example, binary optimizations can be used to change the set of frequently exercised paths in a processor to avoid activating the longest paths. Since these paths are the first to have negative slack when the processor is overscaled, throttling their activity reduces early onset timing violations. Similarly, binary optimizations can be used to reduce error rate by throttling activity in structures of the processor that cause the most errors. Other possibilities include optimizations to overlaps errors in a single cycle to reduce the effective errors per cycle and optimizations to redistribute errors in the processors to reduce the effective error recovery overhead.

This paper on timing speculation-aware binary optimization makes the following contributions.

- We show that the activity distribution of a processor, and by extension, the error distribution, can be altered through binary optimizations.
- We demonstrate that the energy efficiency of timing speculative processors can be improved by altering their activity distributions through binary optimizations, without any additional hardware support.
- Through careful analysis of the main factors that influence processor error rate, we show that several optimizations that are already supported by existing compilers can improve the energy efficiency of TS.
- We quantify the energy savings from targeted and standard binary optimizations for a family of timing speculative processor architectures. We observe up to 39% additional energy savings from TS-aware binary optimization for a Razor-based processor.

2. BASELINE ARCHITECTURE

Which optimizations are most effective for a processor depend on which processor modules cause the most errors. In this section, we describe the family of processor architectures we study to develop binary optimization strategies and identify their error-critical modules. We also discuss how the error criticality of modules may depend on program characteristics.

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

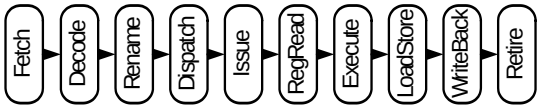


Figure 1: The FabScalar Pipeline. [1]

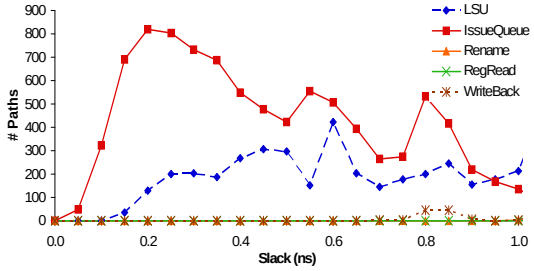


Figure 2: The static slack distributions for the pipeline stages show how many critical paths they have but do not provide information about how often the paths toggle, which is essential in characterizing error rate.

2.1 FabScalar Architecture

We use the FabScalar [1] framework for our architectural evaluations. FabScalar is a parameterizable, synthesizable processor specification that allows for the generation and simulation of RTL descriptions for arbitrarily configured scalar and superscalar processor architectures. FabScalar allows for the configuration of many microarchitectural parameters, including superscalar width (ss), fetch width and depth (fw,fd), numbers and types of functional units, issue width and depth (iw,id), issue queue size (iq), select logic depth (sel), register file depth (rrd), re-order buffer entries (rob), physical registers (reg), and load and store queue sizes (lsq). In this paper, we study a family of superscalar processors by selecting interesting candidates from the available configurations space of FabScalar. Figure 1 shows the FabScalar pipeline.

2.2 Error Criticality Analysis

Different pipeline stages cause errors at different rates, depending on their slack and activity distributions. Figure 2 shows the static slack distributions for the pipeline stages that cause the most errors. While our highly-optimized design flow removes excess slack in all stages, two stages in particular – the issue queue (IQ) and the load store unit (LSU) – have the highest number of critical paths. Based on Figure 2, one might expect that the IQ, having many more critical paths than all other modules combined, would produce the most errors in the processor. However, the static slack distribution only shows the *potential* for paths to cause errors. Not all stages exercise their critical paths often. Stages with frequently exercised critical paths cause the most errors. In Figure 3, we create activity-weighted, *dynamic* slack distributions by showing the sum of toggle rates for all the paths at each value of timing slack (activity from SPEC benchmarks). The more timing critical *activity* a module has, the more errors it is likely to cause. From Figure 3, it is clear that the LSU dominates the error distribution of the processor.

2.3 Program Dependence of Error Criticality

As demonstrated in the previous section, the LSU, and secondarily, the IQ are the primary sources of timing violations for the family of processor architectures that we studied. Below, we describe the implementation of the LSU and the IQ in the FabScalar processor to understand the dependence of error rate on program characteristics.

The LSU (Figure 4) performs memory disambiguation for the processor. This involves checking for dependencies between loads and stores. After address resolution, a store must search the address CAM of the LQ and process all entries with matching addresses to determine if any load issued out of order and broke a RAW dependence. Load disambiguation is more complicated because it may include store-to-load forwarding. In addition to a search through the SQ address CAM, a load must generate a mask vector indicating all preceding stores in program order. Matching entries from the CAM search are filtered by the mask vector, and the latest resulting entry, if any, forwards data from the SQ data RAM to the load.

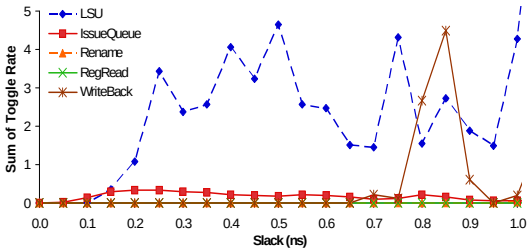


Figure 3: The activity-weighted (dynamic) slack distributions for different pipeline stages indicate how much timing critical activity they have, and by extension, how frequently they will produce errors for a given level of overscaling – i.e., a given $(voltage, frequency)$ pair.

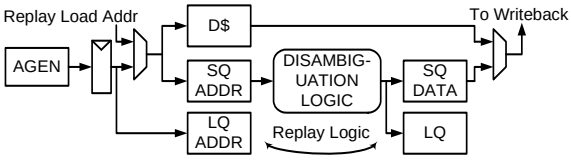


Figure 4: Memory disambiguation is on the critical path of the LSU [1]. The path delay is longest when store-to-load forwarding is required, since this necessitates an access to the SQ data RAM, in addition to the other disambiguation operations.

LSU delay depends on program characteristics for several reasons. The primary reason is that the store-to-load forwarding path is on the static critical path of the LSU. Since many RAW dependencies in a code lead to more forwarding, the timing error rate will be higher for code with a relatively large number of RAW dependencies. Program characteristics also determine the utilization of the LQ and the SQ, which, in turn, dictates access delays for the structures. For example, when the LQ or SQ are nearly full, as may be common for memory-centric codes, more entries must be accessed in a single cycle to generate mask vectors. This increases the length of the propagation path, and consequently, increases delay. Additionally, when there are many dependencies between memory operations, address CAM searches generate many hits, increasing load capacitance and delay for the CAM access. Finally, propagation delay increases when many hits are signaled in parallel (due to many potential dependencies), since the average length of the propagation path from the CAM entries to the port increases. Hence, the average delay is higher for memory-centric codes with a large number of dependencies.

We confirmed that the forwarding paths are timing critical in the LSU, and that more dependencies result in activation of longer paths, by observing the activity-weighted (dynamic) slack distribution of the LSU for two different instruction streams (Figure 5). The first contains a stream of memory operations that access the same address. Each load depends on the previous store and activates the forwarding paths in the LSU. In the second stream, the dependencies are removed. Figure 5 demonstrates that activity on the critical paths of the LSU is greatly reduced when the dependencies are removed and forwarding is not required.

The wakeup-select logic used in the IQ is similar in nature to the memory disambiguation logic in the LSU. For example, wakeup consists of finding all instructions that depend on the destination register of another instruction. This CAM-based dependence check in the IQ is performed in much the same way as the dependence checks in the LSQ. Likewise, select logic, which selects a ready, waiting instruction to execute is somewhat akin to the masking logic that identifies valid, conflicting stores for forwarding. Because of their similarities, the LSU and IQ have similar timing considerations.

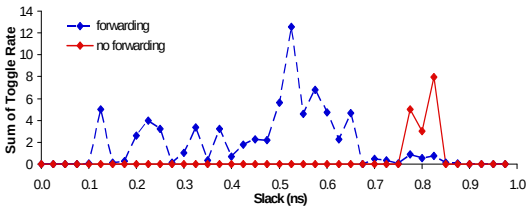


Figure 5: Since forwarding paths are critical in the LSU, eliminating dependencies and the need for store-to-load forwarding reduces activity on the critical paths of the LSU.

Table 1: Average processor-wide Razor overheads.

Hold buffering	Razor FF	Counterflow	Error Recovery
2% energy	23% energy	<1% energy	P cycles

3. METHODOLOGY

To understand the impact of different binary optimizations on the error behavior and energy efficiency of different processor architectures, we used a detailed methodology that carefully models the relationships between execution behavior, power, performance, and reliability. Designs are implemented with the *TSMC* 65GP library (65nm), using *Synopsys Design Compiler* for synthesis and *Cadence SoC Encounter* for layout. In order to evaluate the power and performance of designs at different voltages and to provide V_{th} sizing options for synthesis, *Cadence Library Characterizer* was used to generate low, nominal, and high V_{th} libraries at each voltage (V_{dd}) between $1.0V$ and $0.5V$ at $0.01V$ intervals. Designs are implemented at 500 MHz. Power, area, and timing analyses are performed in *Synopsys PrimeTime*. Gate-level simulation is performed with *Cadence NC-Verilog* to gather activity information for the design, which is subsequently used for dynamic power estimation and error rate measurement. (For more details on error rate measurement, see supplemental Section S2.)

In our evaluations, we compile and run several microbenchmarks to demonstrate architecture-specific TS-aware optimizations. We also run instruction traces from the SPEC benchmark suite (bzip,gap,mcf,parser,vortex). after fast-forwarding the benchmarks to their Simpoints [5]. All benchmarks are compiled with gcc-2.7.2.3 (SPEC benchmarks and gcc version correspond to those supported by FabScalar).

We model Razor-based error resilience [2] in this paper (though our proposed techniques are generally applicable to any timing error resilient processor). Table 1 summarizes the processor-wide static and dynamic overheads of Razor-based error detection and correction. In our design flow, we measure the percentage of die area devoted to sequential elements, as well as the timing slack (with respect to the shadow latch clock skew of 1/2 cycle) of any short paths that need hold buffering. When evaluating energy at the processor level, we account for the increased area and power of Razor flip-flops, hold buffering on short paths, and implementation of the recovery mechanism. Most of the static overhead is due to Razor FFs. Buffering overhead is small, and the availability of cells with high and low V_{th} provides more control over path delay, eliminating the need for buffering on most paths. We also add energy and throughput overheads proportional to the error rate to account for the dynamic cost of correcting errors over multiple cycles. We use a counterflow pipeline Razor implementation [2] with correction overhead proportional to the number of processor pipeline stages (P). We conservatively replace all sequential cells with Razor FFs.

4. RESULTS AND DISCUSSION

We now discuss different architecture-specific binary optimizations that may increase the efficiency of timing speculative processors. The proposed optimizations are primarily geared toward error avoidance in the LSU and IQ. We first discuss targeted loop-based optimizations and quantify their benefits through the use of microbenchmarks. Then, we evaluate the benefits of combining standard *gcc* optimizations using O levels for SPEC benchmarks.

```
for(i=0; i<N; i++)
  sum += A[i];
```

```
for(i=0; i<N; i+=4){
  sum1 += A[i];
  sum2 += A[i+1];
  sum3 += A[i+2];
  sum4 += A[i+3];
}
sum=sum1+sum2+sum3+sum4;
```

Figure 6: Original loop (left) and unrolled loop (right).

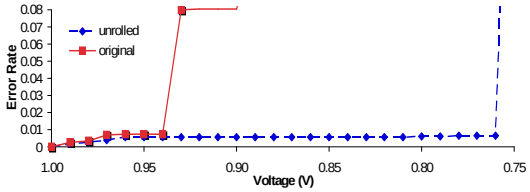


Figure 7: Loop unrolling reduces activity on LSU forwarding paths, resulting in a significant error rate reduction.

4.1 Targeted Optimizations for TS Processors

4.1.1 Loop Unrolling

As described above (Section 2), activity on the static critical paths of the LSU can be reduced by avoiding dependent memory operations and scenarios that cause the LSQ to fill up. This can enable significantly deeper voltage overscaling, since the LSU is often the source of many timing violations.

Loop unrolling is a classic compiler optimization that can eliminate and spread out loop carried dependencies, and thus has the potential to reduce LSU delay. Normally, unrolling would only be used when spin up and spin down costs are overcome by reducing the number of executed instructions. However, TS-aware compilation provides a new use for unrolling – avoiding errors to increase the efficiency of TS by grouping often independent instructions (like vector math) and eliminating often dependent instructions (like branches and loop index updates). Unrolling also allows optimization of register allocation over multiple loop iterations that can eliminate load and store disambiguation, thus reducing pressure on the LSU. Unrolling can also reduce pressure on the branch resolution unit and arithmetic unit, since the number and frequency of branch instructions and loop index updates are reduced. Thus, in addition to fostering critical path avoidance by reducing dependencies, loop unrolling can also be an agent for activity throttling.

Unrolling can cause binary size to increase, which may reduce instruction cache efficiency and may be undesirable in some embedded processors. Unrolling may also cause an increase in dynamic power. When exploiting TS-aware binary optimization, it is important to consider the impact on performance and power, as well as energy efficiency.

Figure 6 shows an example of loop unrolling by a factor of 4. Figure 7 shows the error rate of the processor when executing the two code sequences of Figure 6. Unrolling significantly reduces the error rate by reducing activity on the forwarding paths in the LSU. This error rate reduction enables additional overscaling and results in a substantial energy reduction for a Razor-based TS processor, as shown in Table 2. Microarchitectural parameters not specified in Table 2 are $iq = 16, rob = 64, reg = 64, lsq = 8 + 8 = 16$.

In the error-free case, the same unrolled loop causes dynamic power to increase significantly, even as it increases throughput. Thus, unrolling has the potential to reduce error rate but may also increase power for a conventional processor where TS is not allowed. So, most energy-efficient binary optimization depends

Table 2: Razor-based TS and error-free energy savings (%) for loop unrolling. (ss = superscalar width)

CORE	original	unrolled	unrolled error-free
ss1	11.8	43.1	1.6
ss2	6.4	20.8	2.0
ss4	4.0	42.9	3.2

```
for(i=0; i<N; i++)
  sum += A[i];
```

```
for(i=0,j=N/2;i<N/2;i++,j++){
  sum1 += A[i];
  sum2 += A[j];
}
sum = sum1+sum2;
```

Figure 8: Original code (left – ILP 1) and code with more ILP exposed (right – ILP 2).

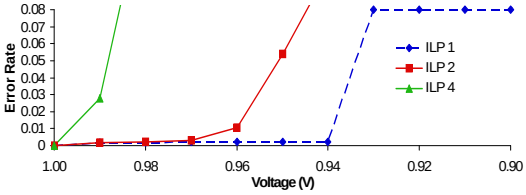


Figure 9: When hardware parallelism is not available ($ss1$), exposing parallelism floods backend queue structures and increases the error rate.

on whether the target uses TS. This demonstrates the need for TS-aware compiler analysis and optimization.

4.1.2 Balancing Instruction-Level Parallelism

In an out-of-order processor, instructions are dispatched to the processor backend as long as there is available space in the appropriate backend structures, namely, the reorder buffer (ROB), IQ, and LSQ. However, when there are not enough execution units to handle ready, waiting instructions, backend structures fill up and remain full. As discussed above, this leads to longer propagation delays for these structures – especially for queues.

Thus, we observe that when hardware parallelism is limited, optimizing the binary to promote software parallelism can actually increase energy in a timing speculative processor by increasing logic delay and limiting overscaling. Consequently, when hardware parallelism is limited, a TS-aware compiler should actually throttle parallelism to prevent instructions from reaching the backend. This kind of compiler optimization is contrary to conventional wisdom, which promotes ILP whenever possible for potential performance gains.

On the other hand, when hardware parallelism is available, the scenario is reversed. Dependencies that hinder ILP keep queues full and increase the delay of dependence-checking logic. Thus, when adequate hardware resources are available, enhancing parallelism can eliminate dependencies and lead to better TS efficiency.

To illustrate the above points, we have run the codes in Figure 8 on TS processors with different superscalar widths. Figure 9 compares the error rates of the code sequences for the $ss1$ case. In this case, hardware parallelism is not available, and exposing more instructions to the processor backend causes queue structures to fill, increasing propagation delays. Thus, the error rate increases as more parallelism is exposed (e.g., ILP4).

For a processor with more hardware parallelism (e.g., $ss2$), the backend can handle increased software parallelism without putting excessive fill pressure on queue structures. In this case, the reduced dependencies of the more parallel code reduce activity in the timing critical disambiguation logic and enable more overscaling. Figure 10 compares error rates for the codes on a $ss2$ processor. The error rate for the code without exposed parallelism (ILP1) increases abruptly and surpasses the error rates for the more parallel codes. Table 3 shows energy results for Razor-based TS, demonstrating that TS efficiency increases *when hardware and software parallelism are balanced*. The table also demonstrates that enhancing parallelism does not provide any significant energy savings in the error-free case, motivating the need for TS-specific compiler analysis and optimization.

Table 3: Razor-based TS and error-free energy savings (%) for balancing parallelism.

CORE	ILP 1	ILP 2	ILP 4	ILP 2 error-free
ss1	13.1	5.5	0.0	1.4
ss2	5.4	9.8	9.7	0.8

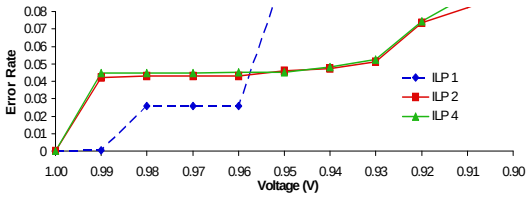


Figure 10: When hardware parallelism is available ($ss2$), exposing parallelism eliminates dependencies and reduces error rate.

```
j = N-1;
for(i=0; i<N; i++){
  sum += A[i] + A[j];
  j = i;
}
```

```
sum = A[0] + A[N-1];
for(i = 1; i < N; i++){
  sum += A[i] + A[i-1];
}
```

Figure 11: Original code (left) and code with a dependence peeled from the loop (right).

4.1.3 Loop Splitting

Loop splitting or peeling can also be used to break dependencies in code by peeling dependent instructions out of the loop body. The original code in Figure 11 contains two dependencies – a loop carried dependence for the accumulator variable (sum), and a dependence between the array indices (i, j). By peeling one of the iterations from the loop, we can eliminate one of the dependencies. This reduces the load on the CAM structure that performs dependence checking, and eliminates occurrences of forwarding. Figure 12 shows how peeling a dependence from the loop reduces the error rate for $ss1$, $ss2$, and $ss4$ processors. Table 4 compares the energy savings achieved by Razor-based TS and error-free operation before and after loop splitting is performed. In all cases, the additional overscaling enabled by loop splitting results in energy savings for Razor-based TS. For error-free operation, loop splitting actually increases energy slightly, because it causes a small reduction in performance (IPC). This divergence between the best decision for TS and error-free cases motivates the need for TS-specific compiler analysis and optimization.

4.1.4 Loop Fusion

Another technique for manipulating dependence patterns in code is loop fusion. Loop fusion merges independent instructions in separate loops into the same loop. Grouping independent instructions can help to break up long chains of dependent instructions by spreading them further apart in the binary. This can reduce the need for forwarding, since conflicting instructions are able to clear the LSQ before their dependent instructions are dispatched to the processor backend. As a side effect, loop fusion may decrease locality of access, which can degrade cache performance. In general, it is important to consider the potential performance impacts of TS-aware binary optimization along with the energy savings it enables.

Figure 13 compares code sequences with (right) and without (left) loop fusion. Note that loop fusion and loop splitting are inverse operations. I.e., the original code can be produced by performing loop splitting on the fused code. In the $ss1$ case (Figure 14), grouping independent instructions does not provide benefits, since there are not adequate hardware resources to handle the exposed ILP. In this case, the unfused (split) code has a lower error rate, because the activity of the LSU (the module that causes the most errors) is throttled by the interleaving of branches and loop index updates with the loads and stores. This activity throttling leads to increased TS energy efficiency, as shown in Table 5.

In the $ss4$ case (Figure 15). the clustering of independent instructions in the fused code spaces out dependent instructions

Table 4: Razor-based TS and error-free energy savings (%) for loop splitting.

CORE	original	split	split error-free
ss1	5.8	13.8	-0.2
ss2	0.0	9.0	-0.4
ss4	3.6	13.4	-0.1

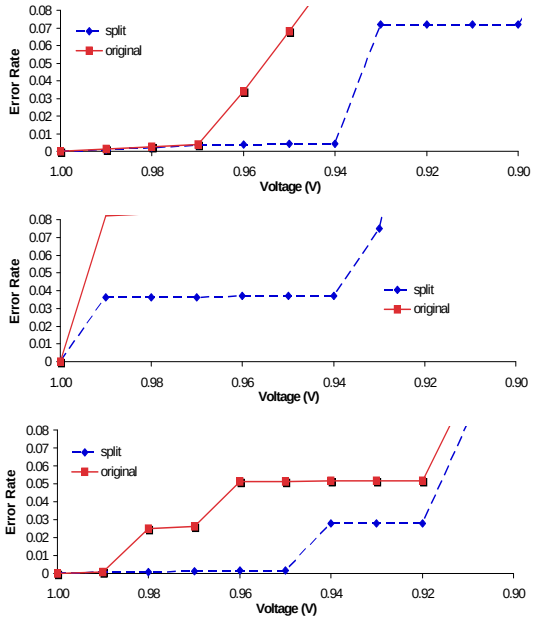


Figure 12: By removing a dependence from the loop, loop splitting reduces the error rates of the $ss1$ (top), $ss2$ (middle), and $ss4$ (bottom) processors.

in the pipeline, thus eliminating many occurrences of forwarding and reducing activity on timing critical paths in the LSU. This critical path avoidance reduces error rate and enhances TS efficiency, as shown in Table 5. Again, energy savings from loop fusion in the error-free case are only meager ($< 1\%$), motivating the need for TS-aware compiler analysis and optimization.

```
for(i=0; i<N; i++)
  sum1 += A[i];
for(i=0; i<N; i++)
  sum2 += B[i];
for(i=0; i<N; i++)
  sum3 += C[i];
for(i=0; i<N; i++)
  sum4 += D[i];
```

```
for(i = 0; i < N; i++){
  sum1 += A[i];
  sum2 += B[i];
  sum3 += C[i];
  sum4 += D[i];
}
```

Figure 13: Original code (left) and code with fused loops (right).

Several other TS-aware binary optimizations are possible. The goal of this paper is to demonstrate that significant energy benefits may be possible from TS-aware binary optimization. An exhaustive exploration of all possible binary optimizations is beyond the scope of this work.

4.2 Standard gcc Optimizations for Timing Speculative Processors

Fortunately, many standard gcc optimizations have similar goals as the targeted optimizations discussed above. For example, optimizing for a higher O level has the potential to reduce dependencies and bolster ILP. Similarly, optimizing for a lower O level may effectively restrict ILP. Below, we evaluate the TS efficiency of SPEC binaries that have been optimized at different O levels.

For architectures without available hardware parallelism (e.g., $ss1$), highly optimizing compute-limited applications can cause pipeline backend structures to fill, resulting in longer delays and higher error rates. On the other hand, for memory-bound applications with many indirect memory references, critical LSU paths are not frequently exercised. Instead, IQ contributes most substantially to the error rate, so optimizing at a higher O level, which reduces average IQ entries, and consequently, IQ delay, reduces the error rate. Thus, when hardware parallelism is limited, compute-limited applications should be optimized for a lower O level ($O0$), while memory-bound, pointer-chasing codes can be optimized for a higher O level.

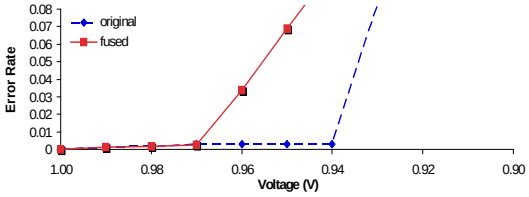


Figure 14: When hardware parallelism is limited ($ss1$), the unfused (split) code has a lower error rate, since LSU activity is throttled.

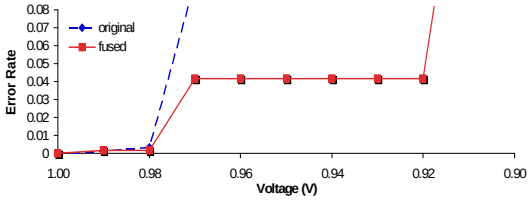


Figure 15: When hardware parallelism is available ($ss4$), the fused code spaces out dependent instructions, reducing forwarding, and consequently, error rate.

Table 5: Razor-based TS and error-free energy savings (%) for loop fusion.

CORE	original	fused	fused error-free
ss1	12.2	5.9	0.2
ss4	4.2	12.3	0.5

For architectures with available hardware parallelism (e.g., $ss2$), highly optimizing compute-limited applications can reduce dependencies, activity on critical LSU paths, and error rate. Optimizations do not have much effect on memory-bound, pointer-chasing codes, since available hardware parallelism allows average IQ entries to remain low, and critical LSU paths are not frequently exercised. Below, we test these intuitions for SPEC benchmarks with standard gcc O levels.

Figure 16 shows the error rates of SPEC benchmarks we evaluated at available O levels, running on the $ss1$ core. Although higher optimizations (e.g., $O2$) generally improve performance (IPC), they increase error rate and degrade TS energy efficiency for compute-limited codes (Table 6). This is because optimizing at the higher O level enhances software parallelism, but there is not sufficient hardware parallelism to handle the dispatched instructions. Thus, backend structures (LSQ and IQ) fill, and propagation delay increases, limiting overscaling. Consequently, performing no optimizations ($O0$) is preferable for compute-limited applications on the $ss1$ core when TS is used. Note that this is an interesting result, as the choice of O level would be different when compiling for the error-free case, since increasing the O level improves performance.

For pointer-chasing codes like *vortex*, which performs object-oriented database lookups, and thus contains many indirect memory references, critical LSU forwarding paths are not frequently exercised. Rather than the LSU, the IQ dominates the processor error rate for the $O0$ binary on this core. Optimizing for a higher O level results in fewer average IQ entries, reducing delay and error rate, and significantly increasing energy savings (Table 6).

For the $ss2$ core, the backend queue structures are not overly stressed. Optimizing at a higher O level reduces dependencies for compute-limited codes, and by extension, activity on the critical paths of the LSU. This reduces error rate (Figure 17) and allows more overscaling and reduced energy (Table 7). Thus, higher optimization (O) levels are beneficial, in general, for Razor-based TS when hardware parallelism is not restricted. Choosing the correct optimization level that balances hardware and software parallelism maximizes energy savings. Note that results in this section demonstrate that the best optimization level is different for TS and non-TS cases. For example, $O1$ achieves the most energy benefits for TS on the $ss2$ core, even though $O2$ has higher performance in the error-free case.

As expected, memory-bound, pointer-chasing codes see little impact from optimizations on the $ss2$ core. The many indirect memory references in *vortex* cannot be optimized at compile time, and thus, optimizations do not significantly impact LSU activity. Also, since HW parallelism is available to relieve IQ

fill pressure, optimizations do not significantly reduce the IQ error rate either.

In the error-free case, optimizing at a higher level ($O2$) can increase performance, but this performance comes with a significant increase in power consumption. Thus, energy is not significantly improved with $O2$ in the error-free case (Tables 6, 7). Distinctions between the best strategy in TS and non-TS cases further demonstrates the need for TS-aware compiler analysis and optimization.

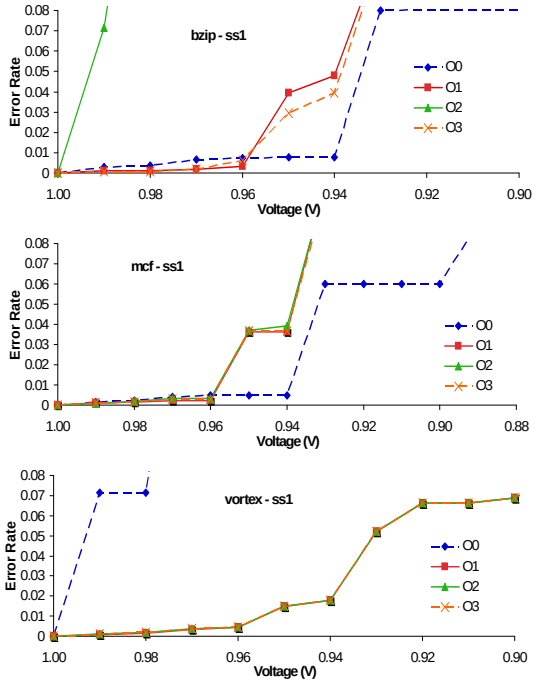


Figure 16: For the $ss1$ core, highly optimizing compute-bound code (e.g., bzip) can increase the error rate, because fill pressure increases the delays of highly utilized pipeline backend structures and limits overscaling. Optimizing memory-bound code (e.g., vortex) can reduce error rate, because critical LSU paths are not exercised, and optimizations reduce IQ fill pressure.

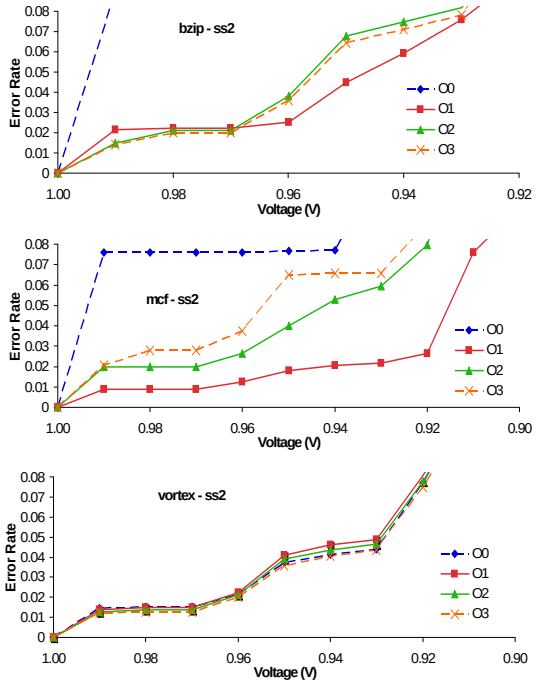


Figure 17: Optimizing compute-bound code (e.g., bzip) can reduce dependencies and activity on the critical paths of the LSU for the $ss2$ core. Choosing the right optimization level that balances HW and SW parallelism can be important. This results in lower processor error rates. The effect of optimizations is limited for memory-bound code (e.g., vortex).

Table 6: Razor-based TS and error-free energy savings (%E), performance (IPC), and binary size (MB) for SPEC benchmarks at different O levels ($ss1$).

$ss1$	bzip			mcf			vortex		
OPT	%E	IPC	MB	%E	IPC	MB	%E	IPC	MB
O0	11.8	0.45	0.32	14.7	0.56	0.31	0.0	0.55	1.70
O1	7.5	0.79	0.29	9.2	0.67	0.29	14.0	0.49	1.48
O2	0.0	0.77	0.29	9.0	0.54	0.29	14.0	0.51	1.47
O3	7.2	0.75	0.31	9.2	0.59	0.30	14.0	0.51	1.49
O2 no-error	1.2	0.77	0.29	0.1	0.56	0.29	0.0	0.55	1.47

Table 7: Razor-based TS and error-free energy savings (%E), performance (IPC), and binary size (MB) for SPEC benchmarks at different O levels ($ss2$).

$ss2$	bzip			mcf			vortex		
OPT	%E	IPC	MB	%E	IPC	MB	%E	IPC	MB
O0	0.0	0.65	0.32	0.0	0.69	0.31	10.4	0.61	1.70
O1	7.7	1.39	0.29	13.4	1.45	0.29	10.0	0.74	1.48
O2	5.7	1.32	0.29	9.1	1.37	0.29	10.2	0.75	1.47
O3	7.5	1.34	0.31	8.5	1.26	0.30	10.4	0.76	1.49
O2 no-error	1.2	1.5	0.29	1.0	1.49	0.29	0.4	0.78	1.48

5. SUMMARY AND CONCLUSIONS

Previous work on improving energy efficiency of timing speculative processors relied on code targeting conventional processors or assumed additional hardware support and instruction set extensions. In this paper, we have demonstrated that careful binary optimization can increase the energy efficiency of error resilient processors without additional hardware support. Since the program binary determines the utilization pattern of the processor, which in turn influences the error rate of the processor and the energy efficiency of timing speculation, optimizing a binary specifically for timing speculative processors can manipulate the utilization of different microarchitectural units based on their timing slack distribution to deliver energy efficiency benefits. We have demonstrated up to 39% additional energy savings with timing speculation-aware binary optimization for Razor-based processors. We expect the energy benefits to grow as more sophisticated compiler techniques are developed.

6. REFERENCES

[1] N. Choudhary, S. Wadhavkar, T. Shah, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar. In *WARP*, 2009.

[2] 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 *MICRO*, page 7, 2003.

[3] Y. Fujimura, O. Hirabayashi, T. Sasaki, A. Suzuki, A. Kawasumi, Y. Takeyama, K. Kushida, G. Fukano, A. Katayama, Y. Niki, and T. Yabe. A configurable sram with constant-negative-level write buffer for low voltage operation with $0.149\mu m^2$ cell in 32nm high-k/metal gate cmos. In *ISSCC*, 2010.

[4] B. Greskamp, L. Wan, W. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing processors for timing speculation from the ground up. *HPCA*, 2009.

[5] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *JILP*, 2005.

[6] G. Hoang, R. Findler, and R. Joseph. Exploring circuit timing-aware language and compilation. In *ASPLOS*, pages 345–356, 2011.

[7] Intel Corporation. Intel atom processor z5xx series, 2008.

[8] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing processors from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[9] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Recovery-driven design: A methodology for power minimization for error tolerant processor modules. In *DAC*, 2010.

[10] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *ASPDAC*, 2010.

[11] J. Patel. CMOS process variations: A critical operation point hypothesis, 2008.

[12] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. Eval: Utilizing processors with variation-induced timing errors. *MICRO*, pages 423–434, 2008.

[13] J. Sartori and R. Kumar. Architecting processors to allow voltage/reliability tradeoffs. *CASES*, 2011.

S1. Understanding How Slack and Activity Distributions Determine Error Rate

In this supplemental section, we explain in greater detail how slack and activity distributions determine the error rate of a processor. The extent of energy benefits gained from exploiting timing error resilience depends on the error rate of a processor. In the context of voltage overscaling-based timing speculation, for example, benefits depend on how the error rate changes as voltage decreases. Likewise, in the context of frequency overscaling, benefits depend on how the error rate changes as frequency increases. If the error rate increases steeply, only meager benefits are possible [8]. If the error rate increases gradually, greater benefits are possible. In this paper, we have considered voltage overscaling-based timing speculation, though our conclusions should also be applicable to other forms of timing speculation.

The timing error rate of a processor in the context of voltage overscaling depends on the timing slack and activity distributions of the paths of the processor. Figure 18 shows an example slack distribution. The slack distribution is a histogram that shows the number of paths in a design at each value of timing slack. As voltage scales down, path delay increases, and path slack decreases. The slack distribution shows how many paths can potentially cause errors because they have negative slack (shaded region). Negative slack means that path delay is longer than the clock period.

From the slack distribution, it is clear which paths can cause errors (timing violations) at a given voltage and frequency. In order to determine the error rate of a processor, however, the activity of the negative slack paths must be known. A negative slack path causes a timing error when it toggles. Therefore, knowing the cardinality of the set of cycles in which any negative slack path toggles reveals the number of cycles in which a timing error occurs.

For example, consider the circuit in Figure 19 consisting of two timing paths. P_1 toggles in cycles 2 and 4, and P_2 toggles in cycles 4 and 7. At voltage V_1, P_1 is at critical slack, and P_2 has 3ns of timing slack. Scaling down the voltage to V_2 causes P_1 to have negative slack. Since P_1 toggles in 2 of 10 cycles, the error rate of the circuit is 20%. At V_3, the negative slack paths (now P_1 and P_2) toggle in 3 of 10 cycles (cycles 2,4,7), and the error rate is 30%.

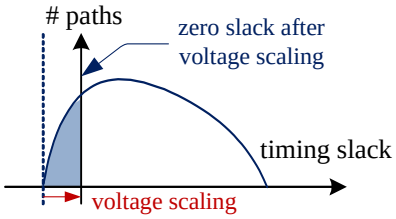
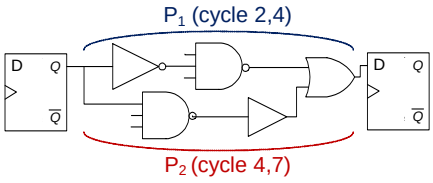


Figure 18: Voltage scaling shifts the point of critical slack. Paths in the shaded region have negative slack and cause errors when toggled.



	Slack		
	P_1	P_2	ER
V_1	0ns	3ns	0%
V_2	-1ns	1ns	20%
V_3	-2ns	-1ns	30%

Total cycles = 10

Figure 19: Slack and activity distributions determine the error rate.

S2. Details of Activity-based Error Rate Calculation

This supplemental section provides additional details on how we calculate error rate in our design flow. To calculate the error rate produced by a binary running on a processor implementation, we run a gate-level simulation of the binary on the synthesized, placed, and routed processor RTL, and we capture switching information for the nets in the design in the form of a value change dump (VCD) file. To calculate the error rate of a design at a particular voltage, toggled nets from the VCD file are traced to find the paths that have toggled in each cycle. The delays of toggled paths are measured using PrimeTime, and any cycle in which a negative slack path toggles is counted as an error cycle. The error rate (ER) of the design is equivalent to the cardinality of the set of error cycles, divided by the total number of simulation cycles (X_{tot}), as shown in Equation 1,

$$ER = \frac{|\bigcup_{p \in P_n} \chi_{toggle}(p)|}{X_{tot}} \quad (1)$$

where P_n is the set of negative slack paths and $\chi_{toggle}(p)$ is the set of cycles in which path p toggles.

Figure 20 shows an example VCD file and illustrates the path extraction method. The VCD file contains a list of toggled nets in each cycle, as well as their new values. Toggled nets in each cycle are marked, and these nets are traversed to find toggled paths. A toggled path is identified when toggled nets compose a connected path of toggled cells from a primary input or flip-flop to a primary output or flip-flop. In Figure 20, nets a, b, and c have toggled in the first and fourth cycles (#1, #4), and nets d and c have toggled in the second and fourth cycles (#2, #4). Two toggled paths are extracted: $a-b-c$ and $d-c$. Paths $a-b-c$ and $d-c$ both have toggle rates of 40% ($|\chi_{toggle}(p)| = 2$ and $X_{tot} = 5$). Therefore, if only one of the paths has negative slack, the error rate is 40% in this example. If both paths have negative slack, then timing errors will occur in cycles #1, #2, and #4, for an error rate of 60%.

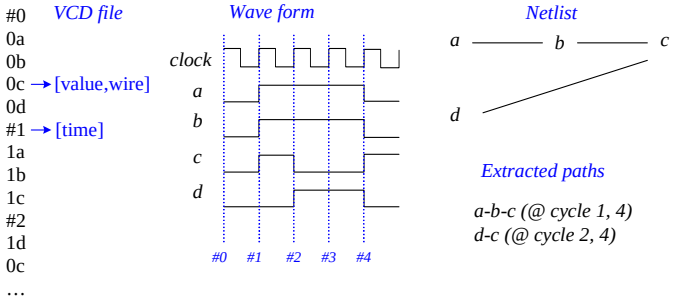


Figure 20: VCD file format and path extraction from the VCD file.

In addition to inducing timing errors by increasing logic delays, voltage scaling may prompt reliability concerns for SRAM structures, such as insufficient Static Noise Margin (SNM). Fortunately, the minimum energy voltage for our processors is around 750mV, while production-grade SRAMs have been reported to operate reliably at voltages as low as 700mV [3]. Research prototypes have been reported to work for even lower voltages. In any case, modern processors typically employ a "split rail" design approach, with SRAMs operating at the lowest safe voltage for a given frequency [7].

S3. Related Work

Previous work on TS-aware design has focused on optimizing *hardware* to improve the efficiency of TS. Work has been done primarily at the design level [4, 8–10, 12] and the architecture level [13] to reshape the slack distribution of a processor to enhance the energy efficiency benefits of TS. These optimizations primarily focus on making the static slack distribution of a processor more amenable to overscaling.

This work, however, focuses on optimizations at the *software* level that influence the activity and *dynamic* slack distributions of a processor (see Section 2). Since the error rate of a timing speculative processor depends on both slack and activity (see Section S1), TS-aware compilation has just as much potential to optimize processor error rate as hardware-based techniques. A promising direction of work involves co-optimization of software and hardware to reshape the dynamic slack distribution and maximize the energy efficiency benefits of exploiting TS.

The closest related work [6] focuses on extending the instruction set to include instructions for which the circuit implementation has a shorter critical path. Replacing instructions with these new instructions increases timing slack and enables more overscaling. The instruction set extensions proposed by [6] primarily focus on reduced-complexity arithmetic operations. We optimize program binaries to improve energy efficiency for TS processors without requiring hardware support.

In a typical ASIC design flow, all paths with excess timing slack are optimized to remove the timing slack, thus reducing power consumption and area. This design style produces a design with a critical slack wall [11], so that the vast majority of timing paths have near-critical slack. Since all circuit modules in our designs have many critical paths, as we would expect in a processor implemented by a typical CAD flow, we are unable to utilize optimizations that redirect instructions to units with more timing slack [6]. Instead, our optimizations focus on avoiding activation of the critical paths in a hardware unit and throttling the activity of units that cause the most errors. Additionally, we focus on binary optimizations that do not require instruction set extensions, and thus, may be more generally applicable. Finally, since the architectures that we evaluate are different than the architecture studied in [6], the modules that cause the most errors are different. Therefore, our architecture-specific optimizations focus on different regions of the processor.