Exploring Circuit Timing-aware Languages and Compilation

Giang Hoang Robert Bruce Findler Russ Joseph

Electrical Engineering and Computer Science
Northwestern University
{gho705,robby,rjoseph}@eecs.northwestern.edu

Abstract

By adjusting the design of the ISA and enabling circuit timing-sensitive optimizations in a compiler, we can more effectively exploit timing speculation. While there has been growing interest in systems that leverage circuit-level timing speculation to improve the performance and power-efficiency of processors, most of the innovation has been at the microarchitectural level. We make the observation that some code sequences place greater demand on circuit timing deadlines than others. Furthermore, by selectively replacing these codes with instruction sequences which are semantically equivalent but reduce activity on timing critical circuit paths, we can trigger fewer timing errors and hence reduce recovery costs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, compilers, optimization; C.0 [*General*]: Hardware/software interfaces

General Terms Design, Languages, Performance

Keywords timing speculation, compiler, ISA design

1. Introduction

While we can expect Moore's Law to continue to provide increases in transistor density, we know that frequency scaling will not provide the same performance gains that it traditionally has. Power and thermal concerns place first-order constraints that drastically curb frequency. Challenges in semiconductor manufacturing, device reliability, design complexity all play increasingly critical roles in limiting frequency gains. Together these factors have redirected industry toward a future where chip-multiprocessors featuring an increasingly large number of simple cores will be more common. At the same time, Amdahl's law and some recent studies [14] remind us that single-thread performance cannot be neglected. We will still need to extract some gains from clock frequency.

Timing speculative architectures present an increasingly popular research direction which aims at some of the technology challenges to boost clock frequency and extract more performance from meager power budgets [6, 8, 9, 11, 17, 18, 23]. Systems that support timing speculation abandon the traditional worst-case design assumptions about circuit timing constraints and clock-frequency. They instead expect timing faults to occasionally appear during runtime and rely on runtime hardware mechanism to detect and recover from the errors. The error recovery ensures correctness and in exchange the system can operate at a greater clock-frequency or decreased supply voltage, yielding improved overall instruction throughput and energy-efficiency. Previous work has explored many ways to judiciously trade-off error rate for clock-frequency and power [23]. In particular, some recent efforts have shown the benefits for from-the-ground-up support for timing speculation [9, 17]. By considering gate-level timing characteristics and how frequently each path is exercised, circuits can be optimized to improve their error rate curves and improve their suitability for timing speculation. While these proposals view the design of timing speculative systems more holistically, they still focus almost exclusively on innovation below the ISA. They do not consider the role that other layers of the system stack may have on influencing timing errors and hence cannot exploit the capabilities of language design, compilers, binary optimizers, or system software to improve performance.

Given the promise of timing speculation and the prevailing focus downward to the lower portions of the system stack, improvements in code generation offer enticing possibilities. In particular, a compiler which considers how code selection and instruction scheduling influence timing errors could reduce the degree to which timing sensitive paths are exercised in a way analogous to techniques aimed at the circuit-level [9, 17]. By generating binaries specifically targeted for timing speculation, a timing speculation aware compiler will be able to significantly reduce incidence of timing errors. This will allow systems to operate at more aggressive clock frequencies and extend the reach of timing speculation.

In this paper, we take steps toward building systems that are holistically designed for timing speculation. We make the observation that the sequence of instructions in an application binary can have significant impact on timing error rate and introduce code transformations and ISA extensions that improve the efficacy of timing speculation while preserving program semantics. Specifically, computations that risk activating critical paths can often be replaced with those that do not, at either no cost or a small increase in instruction count. In particular, simple loop optimizations targeted specifically for timing speculative designs can have dramatic effects in increasing clock frequency for timing speculative architectures. Given that recovering from a timing error incurs a minimum penalty of a pipeline flush, and potentially more if checkpoints are involved, this is often a significant gain.

We evaluate the potential benefits of timing-aware code by executing code with several different types of optimizations on a complete gate-level timing model for a simple Alpha processor. This paper makes the following contributions:

- We introduce the concept of timing speculation-aware code generation and evaluate a number of simple yet effective code transformations that reduce timing error rate and boost overall performance.

Revised March 1, 2011 to fix typos.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.

- We evaluate a number of low cost extensions to the microarchitecture and ISA that allow for even more flexibility in avoiding critical paths during code generation.

The remainder of the paper is structured as follows: In Section 2, we walk through an example that illustrates how compilers can enhance timing speculation. Then, we describe a set of program and architecture optimizations in Section 3. In Section 4, we present an evaluation of our code transformations on a detailed processor model. Section 5 discusses related work. Finally, Sections 6 and 7 discuss some limitations in this our work and conclude.

2. Overview

To get a sense of how compiler optimizations, the design of the ISA, and the design of the programming language can all come together to make timing speculation more effective, this section explores how a compiler can generate code for the simplest loop possible in order to maximize the performance gains that timing speculation offers.

Consider this C code that counts from 0 up to 127.

```
#define N 128
for (int x=0; x<N; x++)
  /* empty loop body */ ;
```

With the exception of the **or** instruction, a typical compiler would generate this code for that loop:[1]

```
main:
    addq $31,$31,$1
$L1:
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmple $1,127,$2
    bne $2,$L1
```

The **or** operation serves as a **nop** (register 31 is always zero) corresponding to the empty loop body, but also transitions the inputs and internal nodes in the ALU so that different iterations of the loop are less likely to create happy timing coincidences.

Running this program at 2.2x[2] of the processor's normal frequency causes nearly all of the instructions in the program to fail, making the program finish at 1.28x the time of the original program.[3] Although this program is an efficient implementation of the original C program on an ordinary processor, it demonstrates a number of choices that could be improved to target an architecture with timing speculation.

As a first step to improving the program, consider the use of the **cmple** instruction. Since less than comparison logic at best entails several levels of logic and at worst is implemented via subtraction, this instruction generally requires a long time to complete. The loop termination condition for this program could have been implemented via an equality check, however, which would behave much better with the higher frequency.

Unfortunately, to compile the given code into code that uses an equality check requires a fair amount of sophistication in the compiler. While such compilers do exist, a simpler, more robust idea is to merely change the form of the looping construct in the programming language. That is, if programming language supported a looping construct like this one:[4]

```
#define N 128
for (int x in 0...N)
  /* empty loop body */ ;
```

then the compiler is free to generate a loop termination that better supports timing speculation.

So, rewriting the **cmple** to **cmpeq** yields this program:

```
main:
    addq $31,$31,$1
$L1:
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmpeq $1,128,$2
    beq $2,$L1
```

It still has a lot of failures, but the **cmpeq** instruction no longer fails, so the program runs significantly faster than the previous version, taking 0.86x the time of the original program.

We can further improve the code if we augment standard compiler optimizations with timing speculation smarts. One such optimization, loop unrolling, duplicates the loop body in order to get better pipeline behavior and to expose opportunities for other optimizations that cross iterations of a loop. In addition to that, we can also take advantage of a more efficient increment operation once the loop body has been duplicated. Specifically, in even iterations of the loop, we know that adding one to the index operation can be implemented with **or**, a much faster operation. Usually, in an ALU, **or** and addition are implemented separately, where **or** could be computed using an array of OR gates, while an addition operation is implemented with a high performance adder. While the adder is limited by a potentially long carry chain, an **or** operation is very fast because the result bits can be computed in parallel.

Making these transformations to our example yields this:

```
main:
    addq $31,$31,$1
$L1:
    or $31,$31,$31 # nop
    or $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmpeq $1,128,$2
    beq $2,$L1
$L2:
```

which clocks in at 0.81x the time of the original program. Applying the loop unrolling optimization alone, i.e., still using the **addq** instruction to increment the loop results in negligible gains. (Our processor model predicts branches are always taken, so the unrolling improves the pipeline behavior, but these savings are swamped by the recovery costs for timing errors.)

Unfortunately, more aggressive loop unrolling does not help us improve performance of the loop index computation when using a standard ISA because implementing the increment operation in the

[1] Technically, a naive compiler would likely generate worse code than this code and a sophisticated compiler would probably notice that the loop has no effect on the computation and thus eliminate it. Nevertheless, if the loop's body had some interesting computation, a sophisticated compiler would likely generate code like the given code to implement the loop's iteration.

[2] We acknowledge that many practical constraints might prevent one from operating at such an extreme frequency in a real system. Since our example loop is very small and simple, it does not exercise the most critical paths in the processor. Running the simulation at 2.2x frequency helps better demonstrate the failure behavior of this simple loop benchmark as well as the effectiveness of our code transformation.

[3] Our precise model and the associated cost of failure are discussed in Section 4.

[4] Using iterators and generators à la Python or Racket [7]

newly exposed cases would require multiple logical operations and thus multiple cycles.

Instead, we introduce the **brinc** instruction, short for "broken increment". Its operands are just like the **addq** instruction, but it only performs the addition in the lowest 4 bits of the operand. For the remaining bits, the first operand is just carried forward to the result. Thus, when adding a big number to a small one which does not create a carry bit from the lower 4 bits, **brinc** behaves just like **addq**, making **brinc** a good candidate for use in an index computation in an unrolled loop.[5]

Exploiting that in our example produces this program:

```
main:
    addq $31,$31,$1
$L1:
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmpeq $1,128,$2
    beq $2,$L1
$L2:
```

which completes in 0.74x the time of the original program. This time, performing the optimization without the timing speculation enhancements (unrolling without adding **brinc**) improves the performance over the original somewhat (to 0.82x), but still represents a slowdown as compared to the simpler unrolling with the **or**.

At this point, most of the failing instructions are the branch instructions, due to the arithmetic they execute to compute the target address from the current PC. A backward branch with **beq** involves subtracting the PC with an offset, which usually results in a long carry chain. The compiler can generate a **bne** instruction instead of the final **beq** instruction, turning a subtraction into an addition, which is more likely to complete at the higher frequency and thus better exploit timing speculation. We call this optimization "branch rerouting".

Applying branch rerouting to our code yields this program

```
main:
    addq $31,$31,$1
    lda $3,8200($31)
$L1:
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    brinc $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    jmp $31,($3)
$L2:
```

which runs in 0.76x the time of the original program. Interestingly, branch rerouting alone actually slows the program down slightly when we omit the **brinc** instructions, clocking in a 0.83x the time of the original program.

Finally, if the compiler can prove that the loop index variable is not mutated and that the loop has a number of iterations that is an even multiple of 4[6], it can remove the intermediate compare and branch instructions in the unrolled version, making our example run much faster. Of course, most of the speedup here comes from the fact that the compiler can eliminate most of the loop itself and thus this speedup will happen only when the loop body is very simple (e.g., initializing an array or similar).

Making that transformation here produces this code

```
main:
    addq $31,$31,$1
    lda $3,8200($31)
$L1:
    or $31,$31,$31 # nop
    brinc $1,1,$1
    or $31,$31,$31 # nop
    brinc $1,1,$1
    or $31,$31,$31 # nop
    brinc $1,1,$1
    or $31,$31,$31 # nop
    addq $1,1,$1
    cmpeq $1,128,$2
    bne $2,$L2
    jmp $31,($3)
$L2:
```

which runs at 0.37x the time of the original loop.

In short, combining conventional compiler optimizations and judiciously chosen changes to the programming language and the ISA yields results that take advantage of timing speculation much effectively than possible without them.

3. Code transformations

In addition to the transformations given in Section 2, there are a number of other speculation-specific enhancements that compilers can take advantage of.

3.1 NOP Padding

When a timing error occurs in execution units, it has two immediate consequences on the instructions in the pipeline. First, it produces

[5] One might think that when adding a small number to a large number, the delay of the operation is proportional to the length of the carry chain. That is not always the case because the the delay of the current addition depends not only on the current input values but also on the state of intermediate logic gates that were set by the previous addition. In particular, if a previous addition induces a long carry chain but the current one does not have any carries, the intermediate logic gates representing the propagate and generate signals must change to hold the correct values for the current addition. For example, with an 8-bit Kogge-Stone adder, the delay for the addition of 1 to 11111100, which does not have any carries, can vary greatly depending on the previous inputs. If the previous operation is adding 1 to 11111111, the delay of this current addition can be three times as long as if the previous addition is adding 1 to 11111110.

[6] In theory, the compiler can generate a loop header that runs a small number of iterations to guarantee that the main body of the loop is always an even multiple of 4.

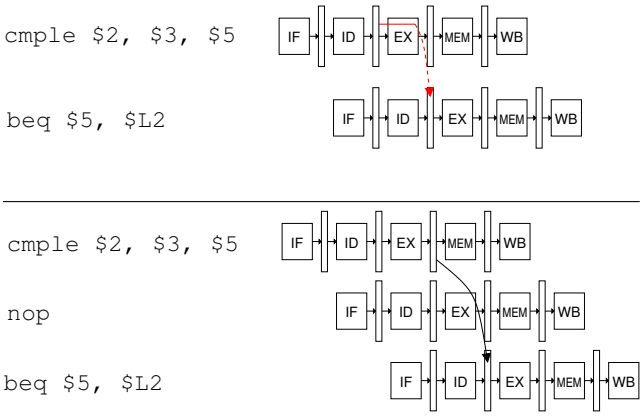


Figure 1. Pipeline timing diagram for two code sequences with RAW register dependencies. For the first sequence (above) the `cmple` instruction completes its execution but cannot forward its result to the following `beq` leading to a timing error for the branch. For the second sequence (below), a NOP has been inserted between the two instructions, padding the timing error. Now the result is forwarded a cycle later from a pipeline register. The padded version of this code eliminates timing errors on the forwarding paths.

an incorrect result for the instruction slotted to that execution unit which must be corrected. Under some error recovery mechanisms, this may require replay and has the likely effect of delaying other instructions in the pipeline. Second, errant results are forwarded to dependent instructions elsewhere in the pipeline. To limit the impact of timing errors, both effects need to be addressed.

By padding the instruction sequence with NOPs, we can mitigate the later concern – incorrect forwarded results with no hardware modifications. However, with some small microarchitectural improvements, we can extend pipeline control logic to improve the pipeline error recovery as well. Figure 1 shows a code sequence with a RAW register dependence. At a frequency scaling factor of 2.15x, the clock cycle would be sufficient for the execution hardware to produce the result and capture the value in the stage pipeline registers but due to the time necessary to pass through forwarding logic, it would violate the setup time needed to successfully forward the value to the dependent instruction. As seen in Figure 1, after inserting the NOP, the result is no longer forwarded directly from the execution unit in the EX stage, it instead is forwarded from a pipeline register in the MEM stage. Consequently, all input values arrive at pipeline registers in sufficient time and no timing errors occur.

If the frequency factor is increased to 2.2x then a simple NOP operation is no longer sufficient. Instead, the compare operation itself cannot produce a result. Under timing speculative architectures that apply stage-level checking granularity (e.g. Razor [6]), the pipeline would likely have to recompute the compare operation and then replay the subsequent instructions, a potentially significant recovery penalty. If on the other hand, the microarchitecture could recognize cases where replay was not necessary, it would significantly decrease the error recovery penalty. One way to achieve this would be to add a special NOP operation which we call PAD to the ISA that tells the hardware that replay is unnecessary. The compiler could then insert this PAD instruction in cases where it can statically determine that replay is unnecessary. Figure 2 shows how the hardware to support this might be incorporated into recovery control logic. The key elements are the PAD detection logic and the AND gate which suppress the flush signal.

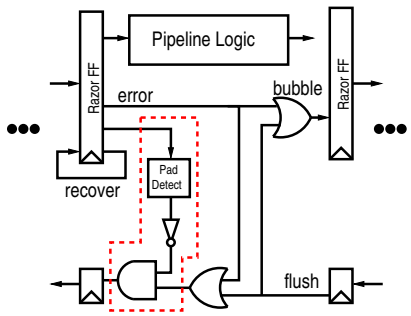


Figure 2. Control logic needed to support the PAD operation. The outline shows the modifications to the original Razor control mechanism [6].

3.2 Broken Operations

Many of the operations in a standard ISA have hidden addition operations as part of their function. For example, computing the effective address of a load involves an addition of an offset to a register and, in some cases, these additions are amenable to **brinc**.

Consider this snippet of a C program implementing a game with a small number of players. Each player tracks which of the other players are adjacent to in a neighbors struct:

```
#define set int

// recording (as a set of booleans)
// who is in the neighborhood
// of a given player
typedef struct {
  set left;
  set up;
  set right;
  set down;
} neighbors;
```

Using this data structure we can write a function that iterates over all of the neighbors structs and determines the set of players that are adjacent to at least one other player (i.e., the players that are not hiding).

```
int all_vis(neighbors *ns, int size) {
  int i;
  set r=0;
  for (i=0; i<size; i++)
    r |= (ns[i].left  |
          ns[i].up    |
          ns[i].right |
          ns[i].down);
  return r;
}
```

When we compile this code, we get this loop.[7]

```
$L5:
        ldl  $1,0($16)
        ldl  $2,4($16)
        ldl  $3,8($16)
        ldl  $4,12($16)
        addl $5,1,$5
        bis  $0,$1,$0
```

[7] Indeed, `gcc -O2` produces a loop almost identical to this one when given the above function, but with the operations in a slightly different order.

```
bis  $0,$2,$0
bis  $0,$3,$0
bis  $0,$4,$0
cmpeq  $5,$17,$2
addl  $31,$1,$0
lda  $16,16($16)
beq  $2,$L5
```

If we can arrange the neighbor array such that it begins at a virtual address that is a multiple of 16, then we know that all of the effective address computations in the **ldl** instructions are safe for use with **brinc**.

3.3 Multi-instruction Code Substitution

In some cases, it may be possible to replace a multi-instruction sequence with an equivalent set of instructions (i.e., that produce an equivalent result), but place less stress on critical path structures. Consider the following two-instruction sequence that performs multiplication by a constant factor of 6. Note that this code was generated by a circuit timing agnostic compiler (`gcc-4.2`):

```
s8subq  $2,  $2,  $4
subq    $4,  $2,  $4
```

The first instruction, `s8subq`, scales r2 by 8 and then subtracts r2 from that result (r4 $\leftarrow$ 8 $\times$ r2 - r2 = 7 $\times$ r2). The second instruction performs one more subtraction to produce the result r2 $\times$ 6. By performing constant multiplication with shifts and add/sub instructions, the compiler can produce multiplies that may be faster than some true dedicated multiplier components. Unfortunately, this code begins to generate timing errors at 1.18x of the base frequency when a small or medium value is present in r2 register. This is due to the way that the `subq` sensitizes the adder carry chain. Note that an instruction sequence like this inside an inner loop would severely limit frequency scaling.

However, the above instruction sequence is not the only combination of shifts and arithmetic instructions that can produce r2 $\times$ 6. We could instead produce the same result using:

```
addq    $2,  $2,  $4
s4addq  $2,  $4,  $4
```

The key benefit of this alternate sequence is that it places much less stress on critical path circuits in the ALU and can tolerate significantly higher clock frequencies.

4. Evaluation

In this section, we describe our processor model and application workload. We then present experimental results and evaluate the code optimizations presented in Section 2 and 3.

4.1 Models and Methodology

We evaluate the potential of code optimization for timing speculative architectures using the following benchmarks: `bitcount`, `dijkstra`, and `strsearch` from MiBench [12], and a sequential version of `integer sort` (`is`) from NAS benchmark suite [2]. Instead of directly implementing these code optimizations in a compiler backend, we apply the code transformations by hand. We compile the benchmarks with `gcc-4.2` with the `-O2` optimization level and hand tune the assembly code with our techniques before assembling into binary.

We construct a Verilog processor model for a simple in-order five-stage Alpha pipeline. Alpha is an example of a clean, classic RISC architecture, and a simple in-order pipeline is representive of some low power-embedded processors. Our processor implements the vast majority of user-mode integer instructions and supports all forwarding paths. We synthesize the processor model using Synopsys Design Compiler with the FreePDK 45nm gate library [25] and target the design for a 350MHz clock speed suitable for a low-power embedded processor. The optimization effort is set to "high" to produce a design as balanced as possible. The critical paths in this design include the integer ALU, some forwarding paths, and NextPC logic. We do not implement floating-point data paths.

We use the M5 simulator [3] to fast-forward the benchmarks to the start of the main execution loop, then transfer the register and memory states to our gate-level Verilog model. The detailed gate-level simulation allows us to evaluate all of our code transformations and generate error rates that reflect the gate-level transitions inside a real design. Due to the extremely long run-times associated with gate-level simulation, we reduced the size of the input sets. We acknowledge that this may have some impact on data working set size and memory stall time. However, it would not have an impact on factors like instruction mix and control flow which would be more closely related to the optimizations proposed in this paper.

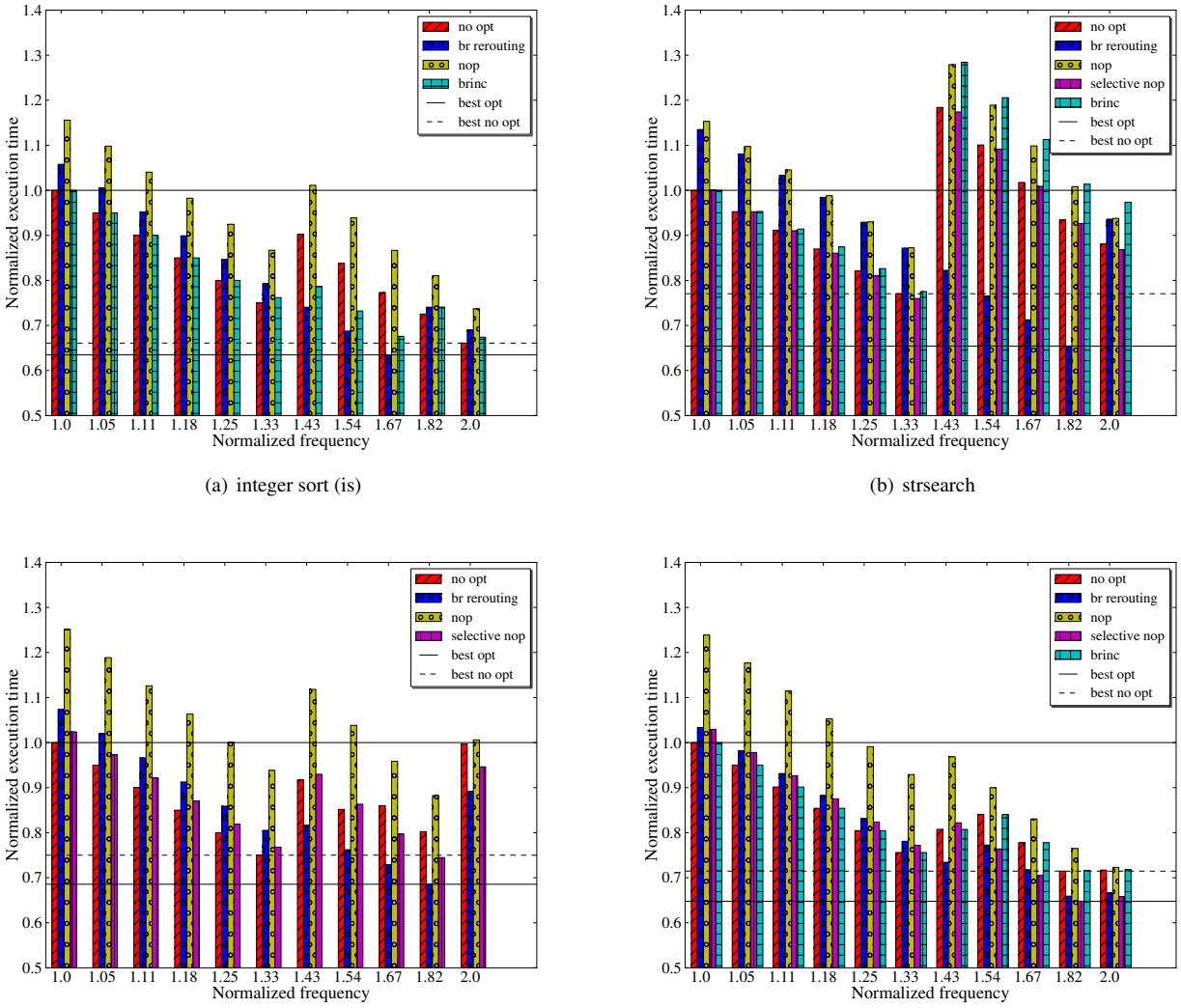
We model a Razor-like per stage dynamic error detection and recovery mechanism. We model a five cycle error recovery penalty which would be appropriate for a shallow in-order pipeline. However, we believe that many of the techniques presented in this paper would also be applicable to architectures which support different checking granularities and correction strategies with alternative recovery latencies. In architectures that feature more punitive recovery latencies, many of our techniques would have a more pronounced benefit.

Our modeled design features an 8KB instruction L1, an 8KB data L1 cache, and a 256 KB unified L2 cache. We do not model timing errors in SRAM structures including the caches and register file. There has been a large amount of recent work on technology-aware cache designs that support overclocking, counter the effects of parameter variation, and tolerate ultra-low voltage scaling [20, 26]. We believe that some of these techniques could be applied to bolster the caches in our design. In addition, the techniques that we apply are orthogonal to any improvements in cache and register file design.

4.2 Experimental Results

We apply three of the proposed code optimizations: *branch rerouting*, *nop padding*, and, *loop unrolling w/ brinc conversion*. To evaluate the impact that modifying the ISA can have on timing speculation, we synthesize execution hardware which implements the **brinc** operation described in Section 2. This operation is intended to accelerate addition with small values. We incorporate the additional hardware into our baseline design. We apply all optimizations to each benchmark, with two exceptions. First, we were unable to apply **brinc** conversion in the `bitcount` benchmark because most of the dominant loops were not suitable for unrolling. Second, for `is`, we did not identify enough candidates for selective nop padding. Even when the frequency is doubled, most of the faults observed in the pipeline are due to backward branch address calculation, for which our nop operation does not help. For *nop padding*, we use two methods of insertion. A naive method blindly inserts nop after any instruction that uses the 64 bit adder. This method is very simple, but it doubles the CPI for those instructions padded even when they are not affected by timing speculation. A better method is to compile the application and profile the application with timing speculation to find out the most frequent failing instructions and selectively pad only those instructions with nops.

Figure 3 presents the normalized execution times for all four of our benchmarks as we scale the clock frequency. The individual sub-figures demonstrate that timing speculation performs well on baseline code. For the `is` benchmark, the performance is maximized at around double the frequency. For the others, the baseline



(a) integer sort (is)

(b) strsearch

(c) bitcount

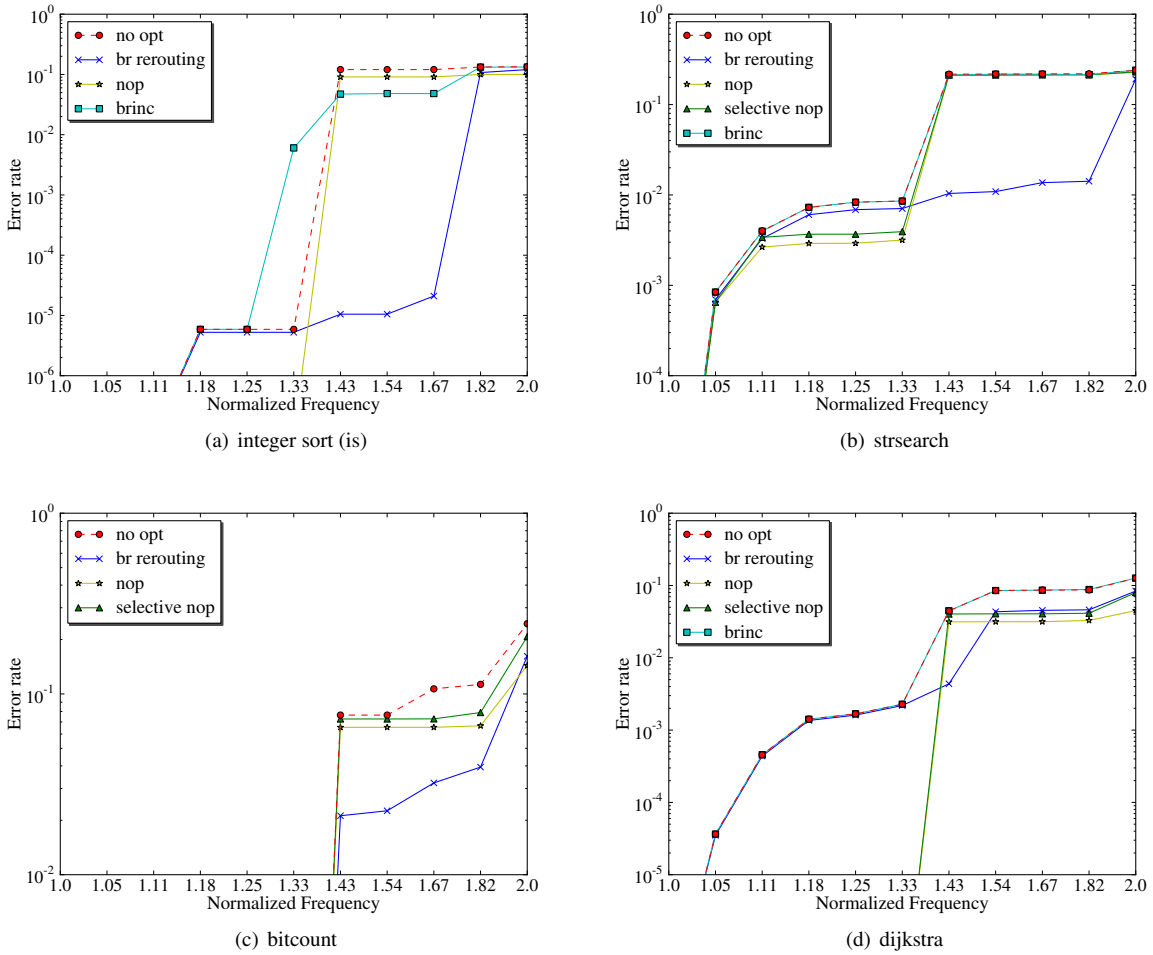
(d) dijkstra

Figure 3. Normalized execution time for `is`, `strsearch`, `bitcount`, and `dijkstra`; showing the baseline performance and with three code optimizations: branch-rerouting, nop-padding, and loop unrolling plus brinc conversion. For nop-padding we examine two varieties: *nop* - which applies nop padding following any instruction capable of stressing critical path circuits versus *selective nop*, which inserts nops more judiciously.

achieves peak performance around a frequency of 1.3x the baseline clock (although `dijkstra` reaches its best performance at both points). While this result confirms some of the benefits of timing speculation these figures more importantly show that by specifically optimizing code for timing speculation, the performance can be significantly boosted. Collectively the code optimizations can shift the optimum frequency upwards and elevate performance. However, it is clear from these figures that not all the code optimizations are successful in every case. Branch rerouting is clearly the most consistent as it is capable of increasing the optimum frequency and noticeably improving performance over the baseline in three of the four benchmarks.

As shown in Figures 3(a) and 3(b), compiler optimizations offer the most promising gains for `is` and `strsearch`. For those benchmarks, branch rerouting is extremely effective and is capable of reducing execution time by **37%** and **35%** respectively. In both cases, at higher frequencies a large number of errors in the baseline code can be attributed to calculating PC-relative targets in the backwards branches within inner loops. By eliminating these errors, the optimized code can effectively reshape the error rate curve and tolerate higher frequencies. This effect is especially pronounced in Figure 3(b) because the error rates for non-optimized code increase very rapidly at higher frequencies.

For the remaining benchmarks, `bitcount` and `dijkstra`, there are performance gains from some of the optimizations, but the overall impact is not as strong. In these two cases, the small performance gains stem from very different reasons. For `bitcount`, the error rates become very large for high frequencies. More importantly, the high error rates can be attributed to instructions which we cannot directly address with our current techniques. This leads



(a) integer sort (is)

(b) strsearch

(c) bitcount

(d) dijkstra

Figure 4. Error rates for the same runs as shown in Figure 3

to limited efficacy. In the case of `dijkstra`, the baseline code achieves excellent performance under extreme frequency scaling. Consequently, it is very difficult for the timing speculative optimizations to contribute additional performance gains.

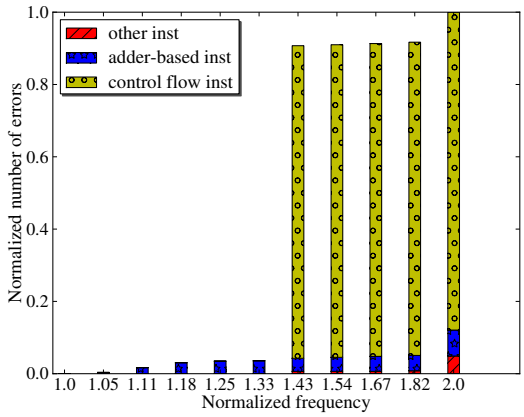
While branch rerouting works well overall, the remaining techniques have more irregular success. These large differences show that proper selection of nop pad locations is essential. The non-selective nop padding consistently achieves poor performance while the selective nop padding achieves some small wins in `bitcount`and `dijkstra`. In contrast, as Figure 3(d) shows, the indiscriminate nop padding severely hurts performance for `dijkstra`. This is a benchmark which scales well with frequency, so by definition, there should be few places to apply padding. Once added, the superfluous nops significantly increase runtime.

4.3 Error Rates

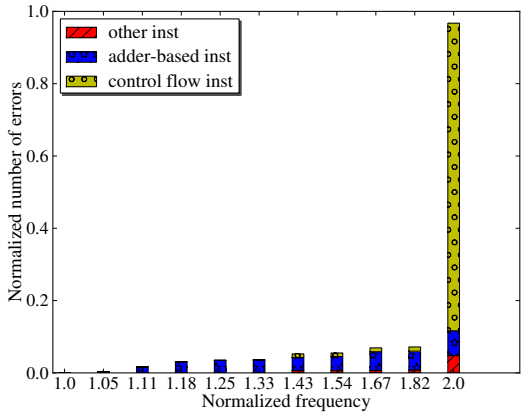
In many cases, the execution time graphs follow a shape like a reverse sawtooth, slowly improving and then suddenly jumping back up. This behavior is due to the way timing errors suddenly appear at particular clock frequencies. Specifically, consider Figure 4, which shows the error count as a function of the clock frequency. In general, there are several plateaus as various thresholds are crossed, which trigger waves of failures. Comparing these graphs with those in Figure 3, the points where the execution time jumps back up correspond to the beginning of the plateaus. This is where a new class of instructions has failed, inducing a sudden jump in error rate. After that point, the execution time gradually decreases along the plateau of the error rate curves, because no new instructions are failing while frequency is decreasing.

The error rate curves in Figure 4 demonstrate how code optimizations can reshape the error rate curve. Furthermore, they help to explain why some of the optimizations perform better than others. First, branch rerouting is capable of maintaining a dominant error rate (i.e. a curve which is no worse than any of the other optimizations) throughout the frequency scaling window for `is` and `bitcount` as well as large portions of `strsearch` and `dijkstra`. In particular, there are some frequency ranges for `is` and `strsearch` where branch rerouting maintains two or three orders of magnitude lower error rates than the unoptimized code. This is a dramatic difference, which shows the potential for circuit timing aware compilation. Referring back to Figures 3(a) and 3(b), these also correspond to frequency ranges where branch rerouting offers superior performance.

Secondly, the two versions of nop padding follow very similar error rate curves as the frequency scales. The indiscriminant nop padding keeps the error rate slightly lower, but there are no large



(a)



(b)

Figure 5. Failure counts categorized by type of failing instruction for the original `strsearch` benchmark and the `strsearch` benchmark with the jump rerouting optimization

gaps between the two. This suggests that the selective nop padding is not too selective i.e. it is not missing many opportunities which are exploited by the indiscriminant padding. In addition, there is a small window (1.1x-1.33x frequency scaling factor) in `strsearch` where nop padding does better than the other techniques. Unfortunately, this does not translate into large performance gains as witnessed in Figure 3. This is because the penalties of superfluous nops outweigh of most of the gains offered by the useful nops. Another way to view this is to say that the degree of nop padding is too high for this part of the error rate curve. In short, nop padding can help reduce error rates in some instances, but it needs to be very selective.

We choose to investigate further the behavior of the branch rerouting technique, the most effective in reducing timing errors. We examine the number of timing errors for `strsearch` before and after branch rerouting is applied. Each error is classified into one of the three categories. Adder-based instructions are arithmetic instructions that rely mainly on the 64-bit adder. These include additions, subtractions, scale-and-add operations. Control flow instructions include conditional branches as well as direct-address jumps. Instructions that do not fit into these two categories are classified as "others". Figure 5(a) shows the error counts for the `strsearch` benchmark, broken down by the type of instruction that fails. The graph shows that 1.33x of the original frequency is the critical point for conditional branches. Beyond that point, a majority of failures are due to conditional branches, and these failures hinder further frequency increase.

Figure 5(b) shows the error count for the same program, but after the branch rerouting optimization has been applied. The conditional branches have been replaced by direct-address jumps. These direct address jumps load the next address directly from a register rather than computing the next address using an adder. Therefore, the delay of a direct-address jump is much shorter than a conditional jump, allowing the processor to operate with very small number of control flow failures until the frequency reaches 2x. At 2x the normal frequency, these direct-address jumps start failing, resulting in massive number of errors due to control flow instructions. These errors put a limit on frequency scaling, however, we can see that the lower latency of direct-address jump already allows significant improvement in frequency scaling of the processor compared to the usual conditional branch.

4.4 Processor Model Discussion

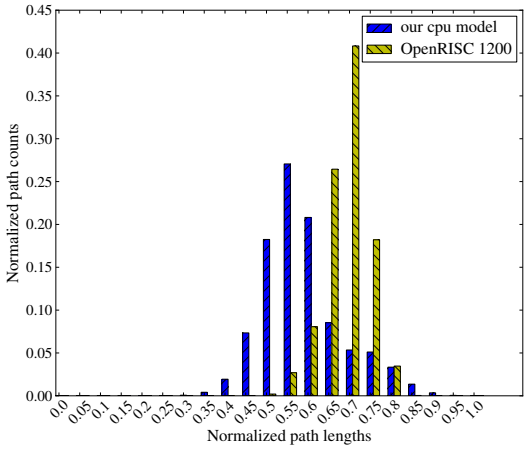
The high frequency that we are able to achieve with our processor model does raise some questions. To make sure that our processor model is reasonable, we analyze the path distribution of our design and compare it with other freely available designs. Figure 6(a) shows the distribution of all paths in our CPU model and an OpenRISC 1200 core. The OpenRISC 1200 is a simple in-order pipelined IP core, which is very close to the design of our processor model. We synthesized the OpenRISC 1200 with the same gate library and target it for 250MHz frequency. The synthesized gate-level model is used for comparison with our design. We wrote a script which enumerates all paths in the synthesized design to create a path distribution histogram.

The figure shows that the path distribution of our processor model skews more towards the center, meaning that our processor has many paths with medium length. The OpenRISC, being an optimized design, has a majority of path lengths closer to the critical path. This explains why we were able to experiment with higher frequency than what we would expect to achieve with a normal processor. However, the fact that our processor model is not as optimized as the OpenRISC 1200 core does not disqualify the merit of this work. The path distribution of OpenRISC 1200 shows that there is still a large gap between the critical path length and the length of the majority of paths in the core (less than 5% the total number of paths has length greater than 75% of the longest path). We believe that the even though the distribution of paths are not the same, the bottleneck in the designs are the same, which are the adder, nextPC logic. Therefore, our compiler techniques will have positive impacts on a processor such as OpenRISC 1200, although the frequency achieved will not be as high.

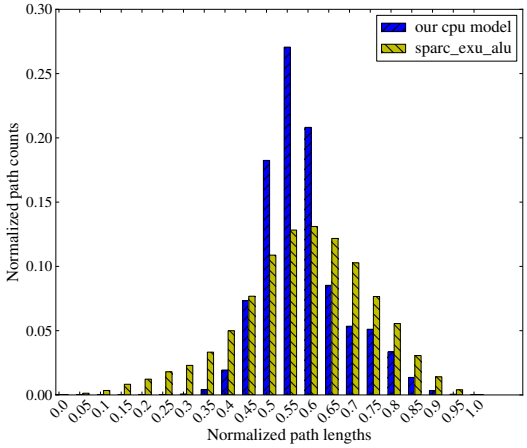
Figure 6(b) shows a comparison of our processor model versus a synthesized OpenSPARC T1 ALU. The OpenSPARC T1 ALU also has a majority of paths distributed to the medium length path, which is not too different from our design. This shows that even though our processor design is not as optimized as OpenRISC 1200 core, it is still reasonable to merit the results of our work.

5. Related Work

In this section, we discuss the proposed techniques in relation to previous work.



(a) Our CPU model vs. OpenRISC 1200



(b) Our CPU model vs. OpenSPARC ALU

Figure 6. Path distributions comparison between our CPU model and other existing models.

5.1 Improving hard fault robustness with the compiler and system software

In recent years, a number of researchers have examined software support to improve system resiliency in the presence of hard faults. These proposals have considered optimizations at the compiler or hypervisor level that either directly prevent architecturally incorrect results from being written to committed state or serve as an assist to hardware which maintains this invariant. The code transformation techniques examined in this work fall into the later category.

Meixner and Sorin introduce *detouring* [19], a software solution which applies code transformations to avoid use of faulty processor components. The authors target this technique for multi-core designs which feature many cores with narrow and shallow pipelines, lacking component-level redundancy. Through a series of code transformations and substitutions applied inside a compiler or binary translator, application software is modified to maintain correct functionality given the presence of known hardware faults. Overall, this technique obtains good fault coverage with zero hardware cost. Consequently, this approach has no performance impact on fault-free cores and may squeeze some performance out of partially functional cores that would otherwise have to be decommissioned.

Gupta *et al.* [10], Hazelwood *et al.* [13], and Reddi *et al.* [27] apply compiler transformations to reduce stress on the power delivery system. The overall goal is to reduce the likelihood of instruction execution patterns [16] that could lead to excitations in the RLC networks which connect the on-chip circuits with the off-chip power supply. In so doing, the system avoids severe droops and ringing on the PWR/GND that would take the circuits out of safe operating range and potentially lead to errors.

System software level has also been focus of investigation for hard-fault detection and recovery mechanisms. In particular, migration and emulation capable virtualization software has been proposed as one means to help reclaim performance from from partially functional cores in a CMP [15]. Recent work has also proposed system-software guided detection of hard faults during processor lifetime [5, 28]. We focus our work at the compiler level and evaluate static code optimization. However, a binary translator or JIT compiler in a level of system software (e.g. virtual machine monitor) may be able to offer many of the same capabilities for generating timing-aware code.

5.2 Hardware support for timing speculation and hard fault resilience

Recent work has examined hardware solutions capable of (i) increasingly the utility of processors under marginal operating conditions, (ii) tolerating hard faults in a single core through demapping and redundancy, and (iii) boosting throughput in CMPs with faulty-cores.

First, there has been increasing interest in hardware that can extract additional performance out of real silicon and eliminate dependence on worst-case design margins [1, 6]. Razor introduced a timing speculative microarchitecture that included dynamic detection and recovery for timing errors [6]. Key innovations included the concept of normal computation tolerating a small non-zero error rate, design of double sampling state elements to detect timing violations, and a low-complexity recovery mechanism based on counter-flow pipelining. In exchange for a small non-zero error rate, Razor designs are capable of significantly reduced supply voltage and can yield dramatic power savings.

Recent trends in silicon manufacturing and processor design complexity have fueled continued interest in architectures that can tolerate timing errors [8, 9, 11, 17, 18, 23]. There have been several important innovations in timing speculative architectures. The Circuit-level Timing Speculation scheme [18] speeds up critical blocks such as adder, rename and issue logic by implementing a faster approximation version of each unit. The approximated results are checked in the next cycle using fully implemented blocks and recovery is possible by reissuing the instruction. Paceline introduced dynamic error detection and correction suitable for aggressive out-of-order processors using a dual core leader-checker strategy and checkpoint-based recovery [8]. The EVAL framework applies microarchitectural techniques and run-time control techniques to influence the error rate curve via high-dimensional adaptation [23]. This work demonstrated the idea that error rate curves could be manipulated to tradeoff error rate versus power and processor frequency. Gupta *et al.* developed a micro-architecture capable of globally adapting voltage and frequency to address process, voltage, temperature (PVT), variations [11]. The fine-grained, high-frequency variations that they tackle necessitate local recovery mechanisms which they introduce. Blueshift applies a ground-up based design strategy to maximize the potential of timing speculation [9]. The approach includes gate-level optimizations to decrease the delay of the most frequently sensitized paths at the expense of less frequently utilized paths. Also taking the ground-up

design approach, Kahng *et al.* proposes a power-aware slack redistribution algorithm that allows the processor to fail more gracefully and extends the range over which voltage/reliability tradeoff is possible [17].

The second category, includes fine-grained techniques that isolate faulty subcomponents and restrict computation to functional portions of the core. Bower *et al.* describe microarchitectural self-repair techniques that tolerate failures in a variety of array structures. Srinivasan *et al.* demonstrate that microarchitectural redundancy can significantly improve mean time to failure under a variety of lifetime wear mechanisms. Rescue applies very fine-grained logic-level fault-isolation techniques as well as de-mapping of faulty components and leverage of functionally redundant resources to obtain high-fault coverage [24]. As a whole, these techniques target primarily stuck-at-faults and are aimed well-below the ISA. In this paper, we consider joint extensions to the ISA and microarchitecture that create more opportunities for functional isolation and limited duplication of execution hardware. This is useful in cases where the faster hardware can be used to avoid timing faults.

Finally, many techniques attempt to improve the performance of multi-core systems with one or more cores that have some permanent component failures. A common insight is that when a given core has lost functionality due to failure, the lost functionality can be provided by some set of remaining cores. Core Cannibalization [22] leverages cross-core redundancy at the granularity of a pipeline stage. A special interconnect between neighboring cores allows them to share functional resources and bypass faulty components. Powell *et al.* examined more coarse-grained hardware driven core salvaging for CMPs [21]. They advocate use of hardware mechanisms to save and transfer state. This permits computational migration in the presence of failure as an alternative to fine-grained resource sharing. In this work, we focus on core-level timing faults. We believe that in some cases thread migration and cross pipeline resource sharing may offer additional ways to reduce timing violations especially when considering parameter variations [4]. In particular, compile-time knowledge may be useful determining how to schedule these operations. This maybe an interesting avenue for future work.

6. Limitations and Future Work

Process variations will be an increasingly significant problem for future processor generations. Prominent process variations could affect our compiler techniques adversely. The variations of path lengths from core to core make the performance of these technique less predictable. First, the difference between a fast path and an equivalent slow path will vary, making the improvement from our techniques more pronounced in one core while less significant in others. It will become harder to decide an optimal set of techniques for each benchmark. We acknowledge that this is a problem but do not address this issue in this work. It will be one of the focuses of our future work.

Our results show that branch rerouting consistently improve performance on all benchmarks. There is not a clear winner, however, among the rest of the techniques. This calls for a methodology to decide where and what techniques should be applied to obtain maximum benefits. We have not addressed this issue in this paper. This is an important item for future work. It will likely involve repeated profile-guided transformations using a compiler.

The introduction of BRINC breaks the ISA abstraction and possibly has consequences with regard to software compatibility. However, we believe that to improve performance, this is a worthwhile tradeoff. With shrinking process technology, future processors will be more likely to be affected by process variations and hard faults. To address these problems effectively, the solutions should involve many layers of the system from software down to the gate level.

7. Conclusion

Timing speculation is a promising technique to further improve single-thread performance. In this paper, we have demonstrated that to unleash more potential of timing speculation, we should involve the design at multiple levels from gate level, micro-architectural level, the ISA, the compiler and all the way up to programming language. In particular, we have proposed and demonstrated a few simple compiler transformation techniques that improve code resilience under timing errors. These techniques use simple compiler transformation to replace long delay operations with faster ones, and reduce the number of timing errors significantly. Although the performance of these techniques varies depending on the benchmark, there is always one that outperforms baseline timing speculation by up to 15%.

Acknowledgments

We would like to thank anonymous reviewers and Antonio Gonzalez for their helpful comments. This work is in part supported by NSF CAREER CCF-0644332 and NSF CNS-0720820.

References

[1] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, page 7. ACM, 2005.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2006.82.

[4] S. Borkar et al. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Design Automation Conference (DAC-40)*, 2003.

[5] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 97–108. IEEE Computer Society, 2007.

[6] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *The 36th International Symposium on Microarchitecture (MICRO-36)*, November 2003.

[7] M. Flatt and PLT. Reference: Racket. `http://www.racket-lang.org/tr1/`.

[8] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 213–224, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: http://dx.doi.org/10.1109/PACT.2007.52. URL `http://dx.doi.org/10.1109/PACT.2007.52`.

[9] B. Greskamp, L. Wan, U. Karpuzcu, J. Cook, J. Torrellas, D. Chen, and C. Zilles. BlueShift: Designing Processors for Timing Speculation from the Ground Up. In *IEEE 15th International Symposium on High Performance Computer Architecture, 2009. HPCA 2009*, pages 213–224, 2009.

[10] M. Gupta, K. Rangan, M. Smith, G. Wei, and D. Brooks. Towards a software approach to mitigate voltage emergencies. In *Proceedings of the 2007 international symposium on Low power electronics and design*, page 128. ACM, 2007.

[11] M. Gupta, J. Rivers, P. Bose, G. Wei, and D. Brooks. Tribeca: design for PVT variations with local recovery and fine-grained adaptation. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 435–446. ACM, 2009.

[12] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th annual Workshop on Workload Characterization*, 2001.

[13] K. Hazelwood and D. Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, August 2004.

[14] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[15] R. Joseph. Exploring salvage techniques for multi-core architectures. In *Workshop on High Performance Computing Reliability Issues (HPCRI) 2005*. Citeseer, 2005.

[16] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003.

[17] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a Processor From the Ground Up to Allow Voltage/Reliability Tradeoffs. In *IEEE 16th International Symposium on High Performance Computer Architecture, 2010. HPCA 2010*, 2010.

[18] T. Liu and S.-L. Lu. Performance improvement with circuit-level speculation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 348–355, New York, NY, USA, 2000. ACM. ISBN 1-58113-196-8. doi: http://doi.acm.org/10.1145/360128.360166. URL `http://doi.acm.org/10.1145/360128.360166`.

[19] A. Meixner and D. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008*, pages 80–89, 2008.

[20] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou. Yield-aware cache architectures. In *roceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)*, December 2006.

[21] M. Powell, A. Biswas, S. Gupta, and S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 93–104. ACM, 2009.

[22] B. Romanescu and D. Sorin. Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 43–51. ACM, 2008.

[23] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. EVAL: Utilizing processors with variation-induced timing errors. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture-Volume*, 2008.

[24] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.

[25] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. Davis, P. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, et al. Freepdk: An open-source variation-aware design kit. 2007.

[26] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu. Trading off cache capacity for reliability to enable low voltage operation. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 203–214, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: http://dx.doi.org/10.1109/ISCA.2008.22. URL `http://dx.doi.org/10.1109/ISCA.2008.22`.

[27] V. J. Reddi, S. Campanoni, M. S. Gupta, K. Hazelwood, M. D. Smith, G.-Y. Wei, and D. Brooks. Eliminating Voltage Emergencies via Software-Guided Code Transformations. *Transactions on Architecture and Code Optimization (TACO)*, 7(2), 2010.

[28] S. Sastry Hari, M. Li, P. Ramachandran, B. Choi, and S. Adve. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 122–132. ACM, 2009.