Software Canaries: Software-based Path Delay Fault Testing for Variation-aware Energy-efficient Design

John Sartori† and Rakesh Kumar‡
†University of Minnesota, ‡University of Illinois at Urbana-Champaign

ABSTRACT

Software-based path delay fault testing (SPDFT) has been used to identify faulty chips that cannot meet timing constraints due to gross delay defects. In this paper, we propose using SPDFT for a new purpose – aggressively selecting the operating point of a variation-affected design. In order to use SPDFT for this purpose, test routines must provide high coverage of potentially-critical paths and must have low dynamic performance overhead. We describe how to apply SPDFT for selecting an energy-efficient operating point for a variation-affected processor and demonstrate that our test routines achieve ample coverage and low overhead.

1. INTRODUCTION

Traditionally, processors have been designed and operated at worst case operating points determined by static critical path delays. This ensures timing safety under all circumstances, including worst case process, voltage, and temperature (PVT) variations, but also entails a significant energy overhead, since worst case conditions are rare [5] and provisioning for the worst case means operating at a much higher voltage and/or lower frequency than required on average. In response to the energy overheads of conventional worst case design, designers have sought more aggressive design styles that permit better-than-worst-case (BTWC) operation [8, 5, 1, 6, 7, 22, 28, 29, 2, 16, 12]. For example, the BTWC design technique that has found the most success in commercial processors is canary circuits [20, 7, 28]. A canary circuit is a protection circuit that attempts to mimic the static critical path delay of a design and is built to fail first in the event of an impending timing violation due to an aggressive voltage or frequency setting caused by variations or by design. Thus, failure of a canary circuit indicates the limit of safe operation (e.g., minimum voltage or maximum frequency) for the static critical path in a processor.

Whereas previous BTWC design techniques have been based on hardware mechanisms that measure slack in timing margins, we propose that testing for available timing slack can be performed with software-based techniques (that may not even require hardware changes to a processor). Software-based path delay fault testing (SPDFT) is a technique that tests for gross delay defects in a design to identify faulty chips that cannot meet timing constraints. We propose to leverage SPDFT as a software-based approach to BTWC operation by generating software routines that test for timing slack on the potentially-critical paths of a design and adapt the operating point to exploit available slack for improved energy efficiency. We call our approach *software canaries*, since our software routines use the potentially-critical paths of a processor as canary circuits.

This paper makes the following contributions.

ISLPED'14, August 11–13, 2014, La Jolla, CA, USA.

http://dx.doi.org/10.1145/2627369.2627646.

- We demonstrate the use of SPDFT to select a variation-aware energy-efficient operating point for a design. To this end, we propose a methodology for generating SPDFT routines based on microrchitectural analysis. Since the distribution of potentially-critical paths for an individual chip also depends on variations, our test generation methodology accounts for the potential impact of variations on the slack distribution. We also present microarchitectural and system support for using SPDFT to select an energy-efficient operating point.
- Using SPDFT to select an energy-efficient BTWC operating point requires software-based test routines that provide high coverage of potentially-critical paths and incur low performance overhead. We show that it is possible to generate test sequences for a processor that achieve ample coverage (e.g., 96.4%) while maintaining low performance overhead (e.g., $< 1\%$). We also present usage models for our test routines, including when they are used in conjunction with hardware canary circuits to ensure 100% coverage of potentially-critical paths.
- We show that using SPDFT to select a BTWC operating point can result in same or better energy efficiency as using canary circuits. Average energy savings are 12% compared to a hardware canary circuit-based design and 27% compared to a conventionally guardbanded worst case design.

2. RELATED WORK

The closest related work is on software-based testing for path delay faults that render a processor defective [4, 26, 11]. To the best of our knowledge, this is the first work to use SPDFT to improve energy efficiency by selecting a BTWC operating point in a variation-affected design. Also, because we perform SPDFT periodically during runtime, the test routines must have significantly lower overhead than typical delay fault testing routines.

We use SPDFT to test for timing slack on the potentially-critical paths in a processor. Fine-grained hardware-based BTWC mechanisms (e.g., timing speculation [5, 1]) can test for timing slack on the critical paths of a processor but have considerable static and dynamic overhead when all potentially-critical paths are targeted (e.g., 25% [15] to 87% [9]), especially in the context of microprocessors where a large fraction of timing paths are potentially-critical [23, 14]. Note that it is possible to use timing speculation and SPDFT synergistically to maximize system efficiency (see Section 3.6).

Another related body of work is on built-in self test (BIST) [21, 25, 30] and software-based self-test (SBST) [24, 4, 11]. A BIST routine uses on-chip hardware to check for defects in logic by exercising the logic and checking the test results. On the other hand, SBST, like SPDFT, uses a processor's instruction set to perform at-speed defect testing. BIST and SBST focus on testing for permanent defects that render a chip faulty. In contrast, we focus on using SPDFT to improve the energy efficiency of a processor through exploitation of timing slack in the presence of static and dynamic variations.

3. EXPLOITING TIMING SLACK IN SOFTWARE

In this section, we describe a framework for deriving software canaries – instruction sequences that test for path delay faults on the potentially-critical paths in a given microarchitecture. We follow with a description of system support required to use SPDFT to dynamically select an energy-efficient operating point for a design affected by static and dynamic variations.

3.1 Deriving Instruction Sequences to Test Potentially-Critical Paths

Software canaries are instruction sequences that test for timing slack on the paths in a processor and produce outputs that are checked against the known correct outputs for the instruction sequences. A valid test sequence must set up and propagate a transition on a path from start to end. We design instruction sequences in such a way that if there is insufficient timing slack at the present operating point, SPDFT will generate an incorrect output or an exception, signaling that the voltage of the processor must be increased. If correct outputs are produced, the present operating voltage is safe, and there may be additional timing slack that allows the voltage of the processor to be reduced. For our objective of using SPDFT to select a BTWC operating point, we focus testing on the set of potentially-critical paths (i.e., the paths that might become critical as the processor is affected by variations). We distinguish between *potentially-critical paths* in a design and *the critical path* in a design because as chips are affected by variations it is possible for different chips to have different critical paths or for the critical paths of a chip to change over time due to aging. If variations (e.g., local and global PVT variations) can cause the delay of a path to change by up to X%, then any path with delay that is within X% of the critical path delay is a potentially-critical path. **By testing the potentially-critical paths in a design for timing safety, SPDFT can ensure timing safety for the entire design**, even the non-architectural state, since the absence of errors on all potentially-critical paths indicates that all shorter paths are also free of errors. Stated another way, if any shorter path fails, one of the potentially-critical paths (tested exhaustively by our test routines) must have failed as well, resulting in an observable failure during testing.

Some approaches exist for generating SPDFT routines for a processor [24, 4, 11], and approaches in previous work target exhaustive path coverage. Since our objective is energy efficiency, not identification of faulty chips, we target test routines that provide ample coverage of potentially-critical paths and have low performance overhead. Our approach for deriving SPDFT routines for a given processor design involves (1) identifying the set of potentially-critical paths (i.e., all paths that may become critical due to variations) and (2) formulating instruction sequences that test those paths with high coverage. Providing coverage for all potentially-critical paths, rather than only a few static critical paths, enables adaptation to *local* as well as global variations. For example, within-die process variations caused by factors such as sub-wavelength lithographic inaccuracies can cause the critical paths on different dies to be different. By providing coverage for all potentially-critical paths, we ensure that **even if local variations change the expected delay distribution of a design, SPDFT can track available timing slack accurately**.

To identify potentially-critical paths, we perform static timing analysis (STA) to identify which paths may become critical as a result of variations. The potentially-critical paths identified by STA are the paths with delays that are within X% of the critical path delay, where X% corresponds to the delay guardband for sources of trackable variations (see Section 4).

Since the number of potentially-critical paths in modern processor designs can be large [23, 14], using conventional instruction-based path delay test generation procedures that generate instruction pairs to test specific paths for delay defects may result in very long test sequences [24, 4, 11]. Instead, we use microarchitectural analysis of potentially-critical paths to design SPDFT routines. Our approach for test routine generation consists of using microarchitectural analysis to formulate *generator templates* that characterize instruction patterns that test for path delay faults on the critical paths in a design, and expanding the generator templates to create an instruction sequence that exhibits high coverage for a particular microarchitecture.

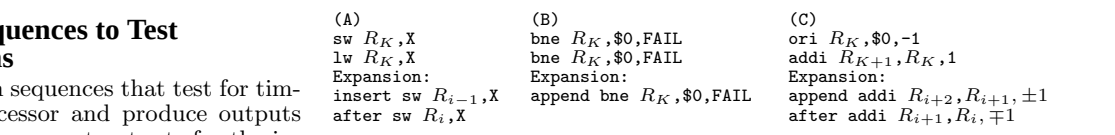

Figure 1: Generator templates are instruction patterns that excite transitions on critical paths in a logic stage and are expanded to provide adequate coverage of potentially-critical paths.

We find that this approach satisfies our goals of ample coverage and low overhead (see Section 5). Generalized automation of test routine generation is a subject of ongoing work.

3.2 Derivation of Generator Templates

In this section, we describe the derivation of instruction sequences that create transitions on critical paths in a generic superscalar processor [3]. The FabScalar processor is an open-source, typical out-of-order pipeline that supports a range of configurations, from a wide superscalar design to a narrow scalar design. Thus, our work demonstrates the viability of using SPDFT to select a BTWC operating point for a range of simple to complex cores. Detailed evaluation of applying SPDFT to set processor operating point in the context of other microarchitectures is a subject of ongoing work.

For ease of exposition, we first explain how we use microarchitectural analysis to derive a SPDFT generator template for the load-store unit, as several other pipeline stages in the FabScalar processor demonstrate similar criticality behavior.

Load-Store: The load-store unit (LSU) performs memory disambiguation. This involves checking for dependences between loads and stores. Load disambiguation begins with a search through the store queue address CAM to determine if the load depends on any in-flight stores, followed by generation of a mask vector that indicates all preceding in-flight stores in program order. If there are matching entries from the CAM search, they are filtered by the mask vector, and the latest resulting entry accesses the store queue data RAM and forwards its data to the load. If forwarding is not required, the memory request is forwarded to the memory subsystem. Paths that perform load disambiguation involving store-to-load forwarding are critical paths in the LSU, and longer paths are exercised when the in-flight dynamic dependence chain is longer. Thus, a generator template for the LSU, shown in Figure 1(A), consists of dependent instructions that necessitate store-to-load forwarding. We observe that increasing the length of the dependence chain increases critical path coverage. Thus, the generator template is expanded by adding dependent stores to the chain until the resulting instruction sequence exhibits adequate coverage of potentially-critical paths, as described in Figure 2.

Fetch: Next PC generation logic and priority selection between multiple branch targets constitute the most critical paths in the fetch stage. Branches in the fetch group check the branch target buffer, branch predictions are generated by the branch prediction buffer, and the next PC is selected based on the predicted branch outcome of the highest priority branch. The critical paths are exercised when all the instructions in the fetch group are branches. A generator template for the fetch stage (Figure 1(B)) consists of back-to-back branches. Branch conditions are written such that the branch outcomes are never mispredicted. We observe that expanding the template by appending additional branches (up to the length of the fetch group) increases coverage of potentially-critical paths.

Decode: For the RISC ISA implemented by FabScalar, decode logic has a regular structure. Several possible instruction sequences exercise critical paths in the decode stage, including the generator template for the RegisterRead, Execute, and Writeback stages (described below, Figure 1(C)). We observe that expanding the generator template (as described in Figure 2) increases coverage of potentially-critical paths.

Rename: A large number of critical paths in the rename stage are exercised when a true dependence chain exists be-

```
// Identify potentially-critical paths
Use STA to identify potentially-critical paths P_PC
 with delay within X% of the static critical path delay
// Expand generator templates
foreach(generator template GT)
   do
      Expand length of GT by one unit in test routine
      Test coverage C_PC of P_PC
   while(C_PC increases)
// Select test with highest P_PC coverage and lowest overhead
Select canary with maximum C_PC that has minimum length
```

Figure 2: Pseudocode for test routine generation.

tween the entire group of instructions for which register renaming is being performed. When the instructions enter the rename stage, new tags are popped from the freelist for the destination registers of the instructions. Comparators indicate that one or more of the source operand(s) of the dependent instructions are the destinations of the other instruction, so instead of reading the source tags from the rename map table, the tags popped from the freelist must be selected for the dependent instructions' sources. Finally, the rename map table is also updated with the renamed register mappings. A generator template for the rename stage (Figure 1(C)) consists of back-to-back instructions that form a chain of true dependences. We observe that expanding the chain length up to N for an N-wide processor by adding dependent operations increases coverage of potentially-critical paths.

Dispatch: The dispatch stage is the gateway between the processor frontend and backend. It checks for free slots in the re-order buffer, issue queue, and load-store queue, and dispatches instructions to free slots. Like decode, dispatch has a regular structure. The generator template used for decode also works well for dispatch.

Issue: The wakeup / select loop contributes the most critical paths in the issue stage. This critical loop is exercised when there is a true dependence chain between successive instructions, such that a source operand for each dependent instruction is the result of the preceding instruction. A newly awoken instruction passes through selection logic, is read from the payload RAM, and broadcasts its destination tag, which hits in the wakeup CAM, closing the loop as the dependent instruction becomes ready. As with rename, the generator template for issue consists of back-to-back dependent instructions that can be expanded into a dependence chain. Thus, we observe that the same generator template can generate test routines that provide good coverage for both stages.

RegisterRead, Execute, Writeback: The critical paths in the register read, execute, and writeback stages are also excited by back-to-back dependent instructions. In this case, one or more source operands for each dependent instruction is obtained from the bypass network from writeback rather than from the physical register file. Critical paths in these stages consist of reading the physical register file, navigating the MUX logic joined to the bypass network, executing the instructions, and writing back the results to the writeback latches and bypass network. Again, a generator template containing back-to-back dependent instructions (Figure 1(C)) works well for these stages. For the arithmetic operations in the chain, we ensure (using gate-level simulation) that the operands selected will excite the static critical paths in the ALU.

3.3 SPDFT Test Generation

Based on the above analysis, we can generate test routines that target the potentially-critical paths in all stages in the pipeline using the three generator templates in Figure 1. The generator templates are expanded into (A) a chain of dependent memory operations that necessitate store-to-load forwarding, (B) a cluster of back-to-back branches, and (C) a chain of dependent arithmetic operations and concatenated to create a SPDFT routine for a particular processor.

Figure 2 describes the process of test generation. The test generation procedure begins by using STA to produce a path

```
ori $2,$0,-1
addi $3,$2,1
addi $4,$3,-1
addi $2,$4,1
sw $4,20($30)
sw $3,20($30)
sw $2,20($30)
lw $4,20($30)
bne $4,$0,FAIL
bne $4,$0,FAIL
bne $4,$0,FAIL
bne $4,$0,FAIL
PASS: [test passed]
FAIL: [test failed]
```

Figure 3: SPDFT routine that provides good coverage for a 4-wide superscalar processor.

```
Read test routine pointer
Begin checkpoint and set watchdog
Remove safety margin
do
   Execute test routine at pointer
   if(!FAIL)
      Decrease voltage by one step
while(!FAIL)
if(recovery or timeout)
   Revert checkpoint
Increase voltage by one step
Add safety margin
Reset watchdog
Return to normal execution
```

Figure 4: Pseudocode describing testing procedure.

timing distribution. Potentially critical paths are identified (using STA) as the subset of paths with delays that are within X% of the critical path delay, where X% corresponds to the delay guardband for sources of trackable variations. Each of the generator templates (described in Section 3.1) is expanded in turn within the test routine while recording the resulting routine's coverage of potentially-critical paths. For each generator template, the length of the instruction chain is iteratively increased by one step until the marginal increase in coverage is negligible (coverage is maximized). Among test routines that provide maximum coverage, we select the one that incurs the least overhead (i.e., the shortest instruction sequence). Figure 3 shows an example of a generated test routine that provides good coverage of potentially-critical paths for a 4-wide superscalar processor.

Compared to the traditional instruction sequences used in SPDFT, which use instruction pairs that target PDFs on individual paths, the test sequences we generate may be more efficient because they focus specifically on potentially-critical paths and target large groups of potentially-critical paths rather than individual paths. We show in Section 5 that our SPFDT routines provide high coverage (96.4%) of potentially-critical paths in our test design (FabScalar [3]), comparable to other approaches for software-based path delay fault testing [24, 4, 11]. We also propose that potentially-critical paths not covered by SPDFT can be covered using hardware canary circuits.

3.4 Microarchitectural and System Support for Selecting an Energy-efficient BTWC Operating Point Based on SPDFT

Using SPDFT to select a BTWC operating point involves using SPDFT routines to monitor availability of timing slack and adapting a processor's voltage to exploit available timing slack that exists due to guardbanding for static and dynamic variations. Since some sources of variations (e.g., temperature, aging) change dynamically, SPDFT routines should be executed periodically. The minimum interval between successive tests that guarantees timing safety can be determined based on the maximum rate of change of trackable variations. During normal operation between testing intervals, a safety margin is added to the operating voltage to protect against potential increase in delay over a single interval due to trackable variations. For example, if trackable variations can change delay at a maximum rate of 1ns per 1ms of execution time, and the length of a testing interval is $1\mu s$, then a guardband must be applied during normal operation to protect against a potential 1ps increase in delay during the interval between tests. Our results account for this overhead, as well as all other power and performance overheads introduced by our SPDFT execution framework (see Section 4). We set the interval length and safety margin such that **performance degradation from periodic testing is less than 1%**. At each testing interval, the safety margin that protects against delay drift is removed and testing is performed to determine the minimum safe operating voltage for the processor. If testing passes, there may be additional timing

slack available, and the processor may lower the supply voltage and perform testing again to check for safety at a lower voltage. When testing fails (either at the original voltage or a lower voltage), the voltage is increased by one increment plus all required margins, and program execution resumes. Proper selection of testing interval (Section 3.6) and intelligent testing patterns [19] can be used to limit the number of voltage points evaluated before arriving at the new optimal voltage. Performance overhead of testing can potentially also be reduced by scheduling testing when the processor is idle [25].

To protect the processor during testing, which may result in errors, we use a checkpointing and recovery mechanism established in prior work [25]. During testing, updates to registers are buffered in a checkpoint memory and updates to memory are buffered in the cache (marked as volatile). If a failed test necessitates recovery, updated registers are reverted, volatile cache lines are marked as invalid, the pipeline is flushed, and execution resumes from the checkpointed state after increasing the voltage to include required guardbands. We account for the overhead introduced by recovery in our evaluations (see Section 4). There are many possible frameworks that support checkpointing and recovery [25, 10], and processors that support speculative execution (like the FabScalar architecture we evaluate) already provide most of the necessary mechanisms. We use the same fault-tolerant checkpointing and recovery mechanism as Bulletproof [25], which prevents any erroneous writes made during testing from being committed to architectural state. We conservatively assume an area overhead of 1.6% for implementing BulletProof, as quoted in [25]. However, since we only require the checkpointing and recovery mechanisms of BulletProof (not the defect testing hardware), overhead in our implementation should be less.

In addition to checkpointing and recovery mechanisms, we include additional support to ensure that the processor can recover from segmentation faults and hangs that might occur due to timing violations during testing. Control errors caused by incorrect branching may cause the processor to jump to an incorrect location or to hang. We use a watchdog timer to protect against control errors and hangs. Before testing, the watchdog timer is set, and the last action of the test routine is to reset the watchdog timer. If a control fault causes the processor to hang or jump to an incorrect location, the watchdog timer expires, and recovery is initiated. Incorrect R/W or O accesses can result in segmentation faults. However, since any segmentation fault during testing can be attributed to a timing error, we suppress normal handling of R/W/O segmentation faults and instead treat them as error detections, which initiate recovery. Context switches and external interrupts are delayed during testing, as SPDFT routines are only a few instructions long. The testing routine is stored in memory as an interrupt service routine that executes at a periodic rate. Since the exact test routine is small, deterministic, and executes at a known rate, it can easily be prefetched just before testing. Figure 4 shows pseudocode for the SPDFT execution framework.

3.5 The Impact of PVT Variations

Our SPDFT execution framework allows a processor to select an energy-efficient operating point in face of process, voltage, temperature (PVT), and aging-induced variations. Figure 5 estimates potential power savings for a FabScalar [3] processor implemented with 65nm technology if SPDFT can enable operating point adaptation to all sources of PVT variations. Results are nearly identical for an OpenSPARC [27] processor implemented in the same technology. Power savings are measured by synthesizing, placing, and routing the processor design at a worst case corner and evaluating the processor at the minimum safe voltage required to meet timing over the range of worst, typical, and best case corners. The figure shows that the power savings available from typical case and best case operation could be up to 22% and 47%, respectively. These savings represent benefits from idealized BTWC operation under typical case and best case conditions.

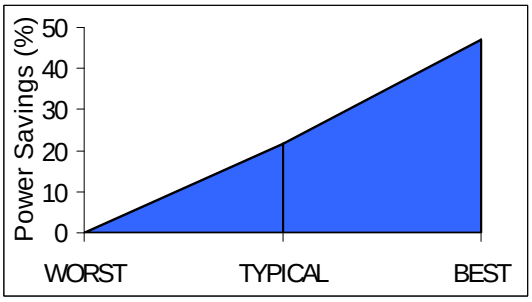

Figure 5: SPDFT can allow a processor that experiences BTWC variations to reduce power through supply voltage reduction. Additional power savings enabled by adapting to PVT variations can be up to 22% under typical case conditions and up to 47% in the best case if SPDTF allows adaptation to all sources of PVT variations.

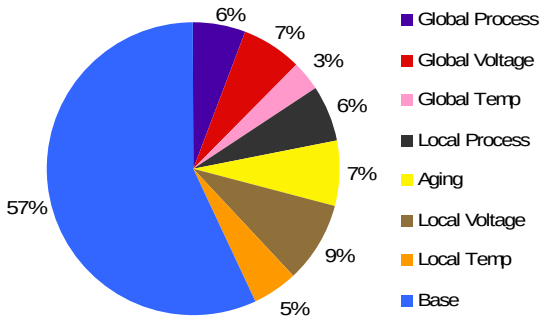

Figure 6: This figure breaks down the power consumption associated with worst case guardbands for various types of variations. If any source of variations can be tracked by SPDFT, the power associated with that guardband can be reduced under BTWC conditions.

In reality, only a subset of these benefits may be possible from SPDFT, since test routines cannot track delay fluctuations caused by all sources of variations. For example, fast-changing variations, e.g., most global and local voltage variations, cannot be tracked by SPDFT due to the latency of processor adaptation through voltage or frequency scaling and the interval between successive tests, which is long relative to the rate of change of fast-changing variations. Figure 6 breaks down the power consumption associated with worst case guardbands for various sources of variations. Note that actual design guardbands would be mostly in terms of operating frequency. However, since we use voltage scaling to target BTWC operating points in this paper, Figure 6 quantifies how much power could potentially be reduced by translating each design guardband into a voltage reduction. The size of a segment corresponding to a particular type of variations shows how much power can potentially be reduced under best case conditions (fast corner) if a hardware or software testing can track delay changes caused by that source of variations. For sources of variation that can be tracked, design margins can be partially reduced, depending on the actual amount of variations observed during operation. In Section 5, we evaluate the potential energy savings enabled by SPDFT both with and without the additional benefits enabled by adapting to a subset of PVT variations.

3.6 Other Implementation Considerations

Critical Path Coverage: SPDFT can allow adaptation to local variations only if the canary routines test all the potentially-critical paths that may become critical due to local variations [5]. Otherwise, guardbands must be used to protect against local variations. In cases where SPDFT provides high but incomplete coverage of potentially-critical paths, it may be beneficial

to use hardware canaries synergistically with SPDFT, such that hardware canaries (e.g., Razor II [5]) provide protection for any potentially-critical paths not covered by SPDFT. Such an organization allows elimination of guardbands for slow-changing local variations while keeping the hardware overhead introduced by canary circuits low (since most paths are protected by SPDFT). In Section 5, we evaluate the critical path coverage of SPDFT, as well as the potential benefits of a SPDFT + hardware canary hybrid design.

Testing Interval Size and Number of Voltage Levels: The interval between successive tests is determined by the maximum rate of change of trackable variations, to balance the performance overhead of testing (higher for a shorter interval) and the cost of providing a safety margin to protect against the maximum drift of critical path delay due to dynamically changing trackable variations in a single time interval (higher for a longer interval). Note that if the interval size is selected properly (to bound the maximum change in delay variation during an interval), SPDFT routines should not need to be run for more than a maximum of three voltages per interval (in the case when voltage step granularity is finest and the optimal voltage is lower than the present operating voltage). In Section 5, we perform analysis of SPDFT execution for different testing intervals and numbers of voltage levels in order to determine appropriate values for design parameters.

4. METHODOLOGY

To quantify the potential benefits of software canaries, we use a detailed methodology to measure power, performance, timing, and activity. Designs are implemented with the TSMC 65GP library (65nm), using Synopsys Design Compiler for synthesis and Cadence SoC Encounter for layout. To evaluate the power and performance of designs at different voltages and design corners, Cadence Library Characterizer was used to generate libraries at each voltage (V_{dd}) between $1.0V$ and $0.5V$ at $0.01V$ intervals for worst, typical, and best case corners. 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 test coverage measurement. Evaluation of potential energy savings is performed by implementing the processor at the worst case corner (conventional design) and evaluating the design at a BTWC corner (e.g., typical, best).

To measure the coverage achieved by our test routines, we use PrimeTime STA to determine the set of paths that can become critical when affected by worst case variations (P_{PC}). We use Cadence NC-Verilog to execute our SPDFT routines on the synthesized, placed, and routed netlist for the processor and produce a VCD file which is used to trace toggled paths from destination to source to determine the paths that are tested by the test routines (P_{SPDFT}). Coverage is computed as the cardinality of the set intersection between P_{SPDFT} and P_{PC} divided by the cardinality of P_{PC}.

We perform evaluations for a collection of benchmarks from the SPEC and EEMBC benchmark suites. Benchmarks are executed on a synthesized, placed, and routed processor. SPEC benchmarks are fast-forwarded to their Simpoints [13], while EEMBC embedded benchmarks are run in their entirety.

4.1 Energy Impact of Dynamic Adaptation

To calculate the expected power of the processor, we use parameters from the technology library to characterize the distribution of variations from best to worst, where the range from best to worst covers six standard deviations (6σ). We take the number of discrete voltage levels as an input and use PrimeTime to measure the power of the placed and routed processor at each voltage level. We then use the cumulative distribution function ($\Phi(x) = \frac{1}{2}[1 + erf(\frac{x-\mu}{\sqrt{2}\sigma})]$), along with the delay vs. voltage relationship of the processor to calculate the probability that the variations are in the range corresponding to each discrete voltage level (erf is the error function). Let us denote the power at the voltage level corresponding to a certain deviation of variations as $Power(x)$. We then calculate the expected value of power as $E[Power] = \sum_{i=1}^{N} \Phi(x)|_{x=x_{i-1}}^{x_i} \cdot Power(x_i)$.

We measure performance (IPC) for each benchmark by executing benchmarks on our processor RTL using NC-Verilog and derate the performance based on the overheads imposed by the dynamic execution framework. First, we calculate the maximum number of voltage level changes per interval using the maximum rate of change of trackable variations, the interval length, and the number of voltage levels. We calculate the maximum expected overhead of switching between voltage levels based on the maximum number of level switches per interval and the cost of switching voltage levels. The cost of switching voltage levels depends on the size of the voltage step, which in turn depends on the number of voltage levels. We perform evaluations for two different voltage regulators – an on-chip voltage regulator that can scale the voltage at a rate of 1 V / 100 ns [18, 17] and an off-chip voltage regulator that is 1000 times slower (1 V / 100 μs) [18]. We calculate the performance overhead for each execution of the test routines as the product of the maximum number of times testing is applied per interval and the number of cycles required to execute the test routines. The maximum number of tests per interval is one more than the maximum number of level switches. Recovery, when required, incurs performance and power overheads for performing the recovery as well as performance lost due to flushing the pipeline. To obtain the derated performance of a benchmark for SPDFT-based execution, we derate the IPC observed during execution of the benchmark based on the total number of overhead cycles devoted to executing SPDFT routines, context switching, switching voltage levels, and recovery. We calculate energy as power divided by performance (W/IPC).

5. RESULTS

Using SPDFT as described in Section 3 can result in improved energy efficiency for a variation-affected design. In practice, available energy savings depends on the extent of variations observed as well as the ability of SPDFT routines to track delay changes due to those variations. Tracking of local variations is enabled only if test routines provide coverage for all potentially-critical paths. Coverage analysis (described in Section 4) reveals that our SPDFT routines provide coverage for *96.4% of potentially-critical paths*. For the remaining 3.6% of the potentially-critical timing paths, we have the option of using Razor II [5] as a canary circuit to provide coverage. Razor II is modeled following the methodology described in [15]. (Note that Razor II [5] is different than Razor-based timing speculation [8] and can be used as a canary circuit.) Due to the high coverage of our SPDFT routines, adding canary circuits for unprotected potentially-critical paths only adds 0.2% power overhead. When Razor II is used as a canary circuit without SPDFT, the overhead introduced by canary circuits is 7.5%.

As discussed in Section 3.5, SPDFT routines cannot track changes in timing slack due to all types of variations. Local and global voltage variations (such as Ldi/dt) can change very quickly, and likely do not allow enough response time for SPDFT routines to adapt the processor's voltage. SPDFT for fast-changing variations is a subject of ongoing work.

Since SPDFT cannot track all sources of variations (e.g., fast-changing voltage variations), SPDFT-based designs still use worst case guardbands for untrackable variations. Table 1 quantifies the power savings afforded by SPDFT-based designs that adapt to various sources of trackable variations and use worst case guardbands for untrackable variations (denoted in the first column). Results are shown for typical and best case corners. The first row corresponds to a SPDFT-based design that does not provide coverage for all potentially-critical paths.

Table 1: Power savings (%) for adapting to trackable variations.

Worst-Case Guardbands	TYPICAL	BEST
Full Voltage, Local Temp and Aging	19.5	25.4
Full Voltage	27.4	38.1

Note that such a design cannot adapt to voltage, local temperature, or aging variations because the coverage of potentially-critical paths is less than 100%. The design can, however, adapt to slow-changing and static global variations. The second row of the table quantifies power savings for a synergistic SPDFT + hardware canary-based design where adaptation to slow-changing local aging and temperature variations is possible because the paths uncovered by SPDFT are covered using Razor II as a canary circuit. The results show that using SPDFT to select a BTWC operating point may significantly improve energy efficiency, with or without the synergistic use of hardware canary circuits. Benefits are significantly higher when a synergistic SPDFT + hardware canary-based approach is used to provide coverage for all potentially-critical paths.

The next set of results quantifies the energy savings enabled by SPDFT for a real execution framework that allows dynamic adaptation to variations (details in Section 4.1). These results consider all performance and power overheads for voltage scaling, test routine execution, and error recovery. We quantify energy savings with respect to both conventional and canary circuit-based baselines. We explore the design space by varying the testing interval size and number of voltage levels.

Figure 7 shows energy savings over conventional and canary circuit-based designs averaged over our benchmark suite. SPDFT achieves energy savings of up to 28% over conventional design and 12% over canary circuit-based design. Energy savings enabled by our dynamic adaptation framework are 96% of the ideal energy savings that could be achieved without any power or performance overheads. Performance overhead introduced by periodic testing and adaptation is low, ranging from around 1% to 3% for different testing interval lengths.

From the results, we observe that testing interval size does not affect energy savings much, though savings are greater for a shorter interval because the cost of providing a larger safety margin to protect against delay drift due to dynamically changing variations during a longer interval outweighs the cost of additional testing incurred with a shorter interval. Energy savings are 5% closer to ideal when 4 voltage levels are used instead of 2. There is a small (2%) boost in energy savings from increasing the number of voltage levels from 4 to 8, however, we use 4 voltage levels for the remaining evaluations, since several conventional voltage scaling designs have up to 4 voltage levels.

The results in Figure 7 assume an on-chip voltage regulator that can scale the voltage at a rate of 1 V / 100 ns [18, 17], as described in Section 4.1. We also performed evaluations for an off-chip voltage regulator that is 1000 times slower (1 V / 100 μs) [18] than the on-chip regulator. In this case, energy savings are 24% over conventional design, 10% over canary circuit-based design, and 83% of the ideal energy savings that assume no overheads.

6. CONCLUSION

In this paper, we propose using SPDFT to select an energy-efficient operating point for a variation-affected design. We describe a procedure for generating software canaries – software-based PDF tests with low performance overhead that provide ample coverage of potentially-critical paths in a processor. We also describe microarchitectural and system support for SPDFT-based execution and show the potential for energy reduction from using SPDFT to select an operating point in a variation-affected design. Average energy reduction at a typical case corner is 12% compared to a hardware canary circuit-based design and 27% compared to a conventionally guardbanded worst case design.

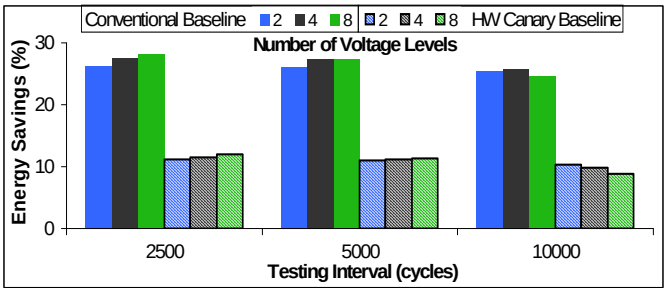

Figure 7: SPDFT in the context of a dynamic adaptation framework achieve up to 28% energy savings over a conventional worst case design and 12% energy savings over a canary circuit-based design. To make results more conservative, we do not account any overhead for canary circuits in the baseline design. Benefits are averaged over all benchmarks in our suite.

7. REFERENCES

[1] K. Bowman, J. Tschanz, C. Wilkerson, S. Lu, T. Karnik, V. De, and S. Borkar. Circuit techniques for dynamic variation tolerance. In *DAC*, pages 4–7, 2009.

[2] T. Burd, S. Member, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, 11(35):1571–1580, 2000.

[3] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *ISCA*, 2011.

[4] K. Christou, M. K. Michael, P. Bernardi, M. Grosso, E. Sanchez, and M. Sonza Reorda. A novel sbst generation technique for path-delay faults in microprocessors exploiting gate- and rt-level descriptions. In *VTS*, pages 389–394, 2008.

[5] S. Das, C. Tokunaga, S. Pant, W. Ma, S. Kalaiselvan, K. Lai, D. Bull, and D. Blaauw. Razor II: In situ error detection and correction for PVT and SER tolerance. *Proc. ISSCC*, pages 400–622, 2008.

[6] S. Dhar, D. Maksimovic, and B. Kranzen. Closed-loop adaptive voltage scaling controller for standard-cell ASICs. *ISLPED*, 2002.

[7] A. Drake, R. Senger, H. Deogun, G. Carpenter, S. Ghiasi, T. Nguyen, N. James, M. Floyd, and V. Pokala. A distributed critical-path timing monitor for a 65nm high-performance microprocessor. In *ISSCC*, pages 398–399, 2007.

[8] D. Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, pages 7–18, 2003.

[9] M. Fojtik, D. Fick, Y. Kim, N. Pinckney, D. Harris, D. Blaauw, and D. Sylvester. Bubble razor: An architecture-independent approach to timing-error detection and correction. In *ISSCC*, pages 488–490, 2012.

[10] M.S. Gupta, K.K. Rangan, M.D. Smith, Gu-Yeon Wei, and D. Brooks. Decor: A delayed commit and rollback mechanism for handling inductive noise in processors. In *HPCA*, pages 381–392, 2008.

[11] Sankar Gurumurthy, Ramtilak Vemu, Jacob A. Abraham, and Daniel G. Saab. Automatic generation of instructions to robustly test delay defects in processors. In *ETS*, pages 173–178, 2007.

[12] V. Gutnik and A. Chandrakasan. An e¡cient controller for variable supply-voltage low power processing. *IEEE Proc. Symposium on VLSI Circuits*, pages 158–159, 1996.

[13] Greg Hamerly, Erez Perelman, J. Lau, and Brad Calder. Simpoint 3.0: Faster and more 'exible program analysis. In *JILP*, 2005.

[14] Andrew Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Designing processors from the ground up to allow voltage/reliability tradeo↑s. In *IEEE HPCA*, pages 119–129, 2010.

[15] Andrew B. Kahng, Seokhyeong Kang, Rakesh Kumar, and John Sartori. Recovery-driven design: Exploiting error resilience in design of energy-e¡cient processors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(3):404–417, 2012.

[16] T. Kehl. Hardware self-tuning and circuit performance monitoring. *ICCD*, pages 188–192, 1993.

[17] Wonyoung Kim, D.M. Brooks, and Gu-Yeon Wei. A fully-integrated 3-level dc/dc converter for nanosecond-scale dvs with fast shunt regulation. In *ISSCC*, pages 268 –270, 2011.

[18] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *HPCA*, pages 123 –134, 2008.

[19] S. Lee, S. Das, T. Pham, T. Austin, D. Blaauw, and T. Mudge. Reducing pipeline energy demands with local dvs and dynamic retiming. In *ISLPED*, pages 319–324, 2004.

[20] Charles R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno, and John B. Carter. Active management of timing guardband to save energy in power7. In *MICRO*, pages 1–11, 2011.

[21] Edward McCluskey. Built-in self-test techniques. *IEEE Des. Test*, 2(2):21–28, March 1985.

[22] M. Najibi, M. Salehi, A. Afzali Kusha, M. Pedram, S. M. Fakhraie, and H. Pedram. Dynamic voltage and frequency management based on variable update intervals for frequency setting. In *ICCAD*, pages 755–760, 2006.

[23] Janak Patel. Cmos process variations: A critical operation point hypothesis, 2008.

[24] M. Psarakis, D. Gizopoulos, E. Sanchez, and M.S. Reorda. Microprocessor software-based self-testing. *Design Test of Computers, IEEE*, 27(3):4–19, 2010.

[25] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. In *ASPLOS*, pages 73–82, 2006.

[26] Virendra Singh, Michiko Inoue, Kewal K. Saluja, and Hideo Fujiwara. Instruction-based self-testing of delay faults in pipelined processors. *IEEE TVLSI*, 14(11):1203–1215, November 2006.

[27] Sun. *Sun OpenSPARC Project*, 2010.

[28] James Tschanz, Keith Bowman, Chris Wilkerson, Shih-Lien Lu, and Tanay Karnik. Resilient circuits: enabling energy-e¡cient performance and reliability. In *ICCAD*, pages 71–73, 2009.

[29] A.K. Uht. Going beyond worst-case specs with teatime. *IEEE Micro Top Picks*, pages 51–56, 2004.

[30] Bardia Zandian, Waleed Dweik, Suk Hun Kang, Thomas Punihaole, and Murali Annavaram. Wearmon: Reliability monitoring using adaptive critical path testing. In *DSN*, pages 151–160, 2010.