Branch-on-Random

Edward Lee Craig Zilles

Department of Computer Science
University of Illinois at Urbana-Champaign
{eslee3,zilles}@uiuc.edu

ABSTRACT

We propose a new instruction, `branch-on-random`*, that is like a standard conditional branch, except rather than specifying the condition on which the branch should be taken, it specifies a frequency at which the branch should be taken. We show that branch-on-random is useful for reducing the overhead of program instrumentation, via sampling. Specifically, branch-on-random provides an order-of-magnitude reduction in execution time overhead compared to previously proposed software-only frameworks for instrumentation sampling.*

Furthermore, we demonstrate that branch-on-random can be cleanly architected and implemented simply and efficiently. For simple processors, we estimate that branch-on-random can be implemented with 20 bits of state and less than 100 gates; for aggressive superscalars, this grows to less than 100 bits of state and at most a few hundred gates.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General – *Hardware/Software Interfaces*

General Terms: Performance, Measurement

Keywords: Profiling, Instrumentation, Sampling, Branch, Pseudo-random, LFSR

1. INTRODUCTION

If we could profile our applications for free, it would facilitate tools that could provide us better understanding of the software systems we build, plus expedite building systems that monitor their own health and optimize themselves based on their behavior. In current machines, however, the overhead of profiling some behaviors introduces significant overhead, which can be prohibitively expensive in the performance critical portions of code.

An effective technique for reducing the cost of collecting information that is applicable to many contexts is *sampling*: collecting information about a subset of events and using that *sample* to extrapolate the behavior of the whole. In the context of computers, sampling allows trading off accuracy for a reduction of execution time overhead, as fewer data items are collected and recorded.

Computer systems have long provided hardware support for sampling via timer interrupts and performance counters. These sampling mechanisms have been used to build effective tools for low-level performance analysis like DCPI [2] and VTune [18]. But, when trying to analyze high-level behaviors in complex systems, these hardware samples are insufficient [15]. This problem occurs because at different program locations we are concerned with collecting different information. For example, in Google's production search engine, they collect stack traces when lock contention occurs and record the size of an allocated object at a heap allocation site [10]. While it would be possible to record a map from program location to the type of information to collect, this generally is not done.

Instead, the most commonly used approach to collect high-level information is to *instrument* the code with additional code to collect the desired information. Instrumentation is powerful because one has the full flexibility of the computer to collect, filter, encode, and store architecturally-visible[1] information. Instrumentation does however add additional instructions to the execution, which leads to execution time overhead when they are executed. In some cases, this instrumentation overhead can be substantial, resulting in slowdowns as large as a factor of ten [8].

In many cases, however, sampling can also be applied to reduce the overhead of instrumentation. Two main approaches have been investigated: dynamic instrumentation and counter-based sampling. Dynamic instrumentation [17, 23, 30] achieves low-overhead sampling by dynamically inserting and removing the instrumentation from a running application. This can be an extremely effective technique, but requires significant support from a run-time system to modify executing programs as they execute. Counter-based sampling is an alternative approach where a (software) counter is decremented at every instrumentation site and only if the counter underflows is the instrumentation executed and the counter reset. While this technique is simple to implement, it results in a non-trivial amount of execution overhead (to decrement the counter) even when samples are not collected. We discuss counter-based sampling and its associated overheads in more detail in Section 2.

To almost completely eliminate the overhead of a counter-based sampling framework for instrumentation, we propose *branch-on-random*, a new instruction that is simple, useful, and efficient. Branch-on-random is much like a conditional branch instruction, except that rather than specifying a condition for the branch, the instruction encodes the frequency at which it should be taken. A single branch-on-random instruction can be substituted for the entire counter-based sampling framework at each instrumentation site, providing an approach for instrumentation that is **both** simple

[1] Instrumentation may have little or no visibility into the microarchitectural behavior of the code (*e.g.*, branch mispredictions, cache misses). Instrumentation and performance counters complement each other with neither subsuming the other.

and efficient. In fact, the sampling framework overhead is sufficiently small that programmers can exhaustively instrument their code with negligible impact on performance.

The combination of branch-on-random for profiling high-level program behavior and performance counters for profiling instruction-level behavior provides a complete, inexpensive profiling solution. Together these approaches open new opportunities for continuous profiling, so that even the most performance critical portions of the code in production environments can be profiled at all levels. For example, most Java Virtual Machines (JVMs) only collect profile information before the first optimized code is generated. By not profiling the optimized code, JVMs miss opportunities to re-optimize their code as program behavior changes [29]. Furthermore, continuous profiling mitigates the downside of aggressive speculative optimizations; we can accurately track the ratio of correct to incorrect speculations and use that to guide recompilation [34].

The focus of this paper is to demonstrate the utility, ease of implementation, and performance potential of branch-on-random. We do so by evaluating branch-on-random in the context of a JVM, an example of a software system that adaptively optimizes its execution by aggressively profiling its behavior using instrumentation. Specifically, we make the following contributions:

1. **We describe an architecture and implementation for branch-on-random that is clean and straight-forward to implement with minimal hardware complexity.** Branch-on-random can be architected without adding architecturally visible state, the implementation requires a 16-bit register and a few dozen gates, and the branch condition can be completely evaluated in the front-end of the pipeline leading to a short misprediction penalty. We describe the high-level usage then discuss implementation details in Section 3.

2. **We demonstrate that using branch-on-random in a compiler is simpler than counter-based frameworks and provides equivalent accuracy for a given sampling frequency.** When comparing the accuracy of the profile information collected by branch-on-random-based and counter-based sampling frameworks, we find the techniques are either equivalent, or branch-on-random out-performs a simple counter-based scheme, because its pseudo-randomness naturally avoids systematic correlations between when samples are collected and the code being executed. We describe our experimental method and present the accuracy results in Section 4.

3. **We demonstrate that branch-on-random provides an order-of-magnitude reduction in overhead with respect to counter-based frameworks.** Using a detailed timing simulator, we evaluate the instrumentation overhead in DaCapo benchmarks executed on the Jikes RVM and a simple micro-benchmark. We find that branch-on-random reduces the cost of each instrumentation site, and, therefore, its benefit is multiplicative with software techniques like Arnold and Ryder's "Full-Duplication" [3]. Our experimental method and results are presented in Section 5.

We conclude the paper through a discussion of related work not covered elsewhere in the paper (Section 6) and by envisioning additional applications for the branch-on-random instruction (Section 7).

```
❶ if (count == 0)
②     do_profile()
②     count = reset
❶ count--
```

Figure 1. Counter-based sampling. Instead of executing the profile code every time, a counter can be used to decide if a sample should be taken; *i.e.*, the `count` variable reaches 0 after counting down from `reset`.

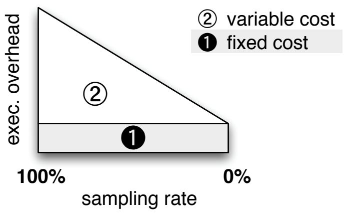


Figure 2. Components of sampling overhead. The total execution overhead from sampling is a combination of fixed and variable costs. The fixed cost comes from the instructions that need to be unconditionally executed while variable costs can be decreased by reducing the sampling rate.

2. BACKGROUND: INSTRUMENTATION SAMPLING AND ITS OVERHEAD

As discussed in the previous section, sampling can be an effective technique for reducing the overhead of collecting profile information. While sampling is only viable for information that can be incomplete, it has been shown to be effective for collecting many kinds of profiles [2, 7, 8, 11, 13, 16]. While some loss of accuracy is likely to occur, for most applications of profiling we are concerned with aggregate behaviors and this loss of precision is tolerable. In particular, many performance-oriented applications are primarily concerned about the frequently-executed portions of the code (for which we will have many samples) and those behaviors that are highly-biased (whose samples will have a low variance), for which sampling allows us to accurately characterize those program behaviors that we can potentially exploit [5].

Sampling permits a reduction in overhead because it prevents the instrumentation code from being executed every time the instrumentation site is executed. In fact, we can directly control the overhead from the instrumentation by setting the *sampling rate*, the average fraction of instrumentation sites encountered at which a sample is collected. One technique to implement sampling is *counter-based sampling* (shown in Figure 1), where a variable `counter` starts at a positive value and counts down until it reaches zero, at which point, it is reset. In such an implementation, the sampling rate is controlled by adjusting the value of `reset` for the code.

As shown in Figure 2, we would expect that the overhead of the instrumentation itself (a *variable* cost) would vary directly with the sampling rate (*e.g.*, a factor of two increase in sampling rate would result in a factor of two reduction in instrumentation overhead) and this is borne out in our results and the results of previous work [3]. The sampling framework, however, introduces its own overheads, some of which are independent of sampling frequency. As a result, even when the sampling rate is reduced to zero, the overhead does not disappear. For a given set of instrumentation sites, the sampling framework will introduce a fixed, minimum overhead and previous work has shown that the overhead can be substantial (5-55%) [3].

The lower bound of overhead is purely an artifact of the sampling technique that was introduced to reduce the overhead of profiling. In the case of counter-based sampling, every time the sampling site is reached, it needs to access a counter value and check if it should branch; afterwards, the counter needs to be decremented. In addition to this fixed cost, which can be significant, counter-based sampling contributes to the variable costs by loading a reset value into the counter whenever a sample is taken.

Specifically, a software counter-based framework leads to the following architectural and microarchitectural sources of overhead:

1. Increased code size for sampling yields additional instructions fetched and increases the program's instruction working set, which can cause extra instruction-cache misses.

2. Extra instructions consume execution resources. In out-of-order processors, they must be buffered in the re-order buffer, which reduces the number of non-instrumentation instructions that can be in flight.

3. Most instructions act on registers, so additional register usage breaks up existing register live ranges. In out-of-order machines, the additional physical register requirements can stall the front-end if there are no extra rename registers available.

4. The sampling counter needs to either be stored in memory (requiring additional loads and stores) or in a register (preventing the use of that register anywhere in the instrumented code, a large cost in an instruction set architecture (ISA) with few registers).

5. Sampling requires conditional branching and at high sampling rates branch mispredictions cannot be avoided. This results in expensive penalties of flushing many instructions when the branch is resolved in the back end of the pipeline.

6. These sampling branches will also pollute branch prediction structures. Since the sampling branches are uncorrelated to the program's other conditional branches, there results both an effective reduction in the global history length, as well as a potential pollution of the global history when samples are taken. In addition, if sample branches and other program branches alias in the counter array, destructive interference can occur.

In the next section, we propose branch-on-random, which enables sampling of instrumentation with much lower overhead. We will show that it eliminates many of the sources of overhead while retaining accuracy and flexibility, making it a compelling replacement for existing counter-based frameworks.

3. BRANCH-ON-RANDOM DESIGN

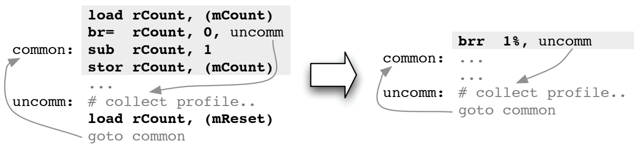
In this section, we describe how to architect and efficiently implement branch-on-random for use in low-overhead sampling. First, we demonstrate how software will use branch-on-random to implement sampling. Next, we present a candidate architecture and encoding for it. Finally, we describe one way to implement branch-on-random as part of the microarchitecture and discuss its performance relative to software counter-based sampling approaches.

3.1 Software Usage

A branch-on-random is a conditional branch. Programs can use it like any other conditional branch to jump over sections of code. The major difference is that rather than specifying the condition under which the branch will be taken, it specifies the frequency with which it should be taken. The program will not know which particular branch instances will actually be taken, because the implementation will pseudo-randomly pick them in a way that the specified frequency will be asymptotically achieved.

```
❶ if_random (1%)
②     do_profile()
```

Figure 3. Using branch-on-random. A branch-on-random is used like other conditional branches except that its condition is a frequency of how often it should be taken.



```
         load rCount, (mCount)
         br=  rCount, 0, uncomm
common:  sub  rCount, 1
         stor rCount, (mCount)
         ...
uncomm:  # collect profile..
         load rCount, (mReset)
         goto common
```

```
         brr  1%, uncomm
common:  ...
         ...
uncomm:  # collect profile..
         goto common
```

Figure 4. Usage of branch-on-random. Multiple lines of existing sampling code can be rewritten to use just a single branch-on-random, brr, while retaining the desired functionality. The bold text highlights the differences, and the shaded region is the additional code that always executes.

An example usage would be to randomly branch to profiling code as shown in the pseudo-code in Figure 3. By controlling how often this branch is taken, the program amortizes the cost of executing the additional, possibly expensive, profiling code. A cursory comparison to the counter-based sampling code in Figure 1 reveals that branch-on-random has simpler code with fewer lines of code contributing to the fixed (shaded) and variable (unshaded) costs. This discrepancy is even more obvious at the assembly level (Figure 4), where the sampling framework can be implemented with a single branch-on-random compared to many instructions for software counter-based sampling.

3.2 Encoding and Semantics

In managing the trade-off between accuracy and overhead, it is useful to be able to select from a wide range of frequencies; but, in general, the ability to exactly specify an arbitrary frequency is not necessary. Thus, we encode the branch frequency value, `freq`, in just 4 bits for 16 possible values. The mapping of bit values to frequencies is given by $(\frac{1}{2})^{freq+1}$. This provides a wide range of frequencies from 50% ($(\frac{1}{2})^1$) to .0015% ($(\frac{1}{2})^{16}$). As we will show, powers of 2 are simple to implement in hardware. Adding 1 to the encoded value, `freq`, avoids re-encoding unconditional jumps (branching 100% ($(\frac{1}{2})^0$) of the time).

The instruction format for branch-on-random ends up looking just like other conditional branches except now the condition is an encoded taken-frequency, which means individual instructions can have different frequencies. A sample instruction format is provided in Figure 5.

We intentionally do not architect that branch-on-random instructions will be taken in any particular sequence, just that asymptotically the branch bias will approach the specified frequency. In doing so, we both permit flexibility of implementation, as well as prevent exposing any architecturally visible state that would need to be checkpointed during speculative execution or context switched to ensure a specific pattern.

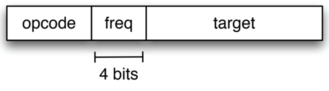


Figure 5. Branch-on-random instruction format. Instead of encoding an explicit condition for the branch, a branch-on-random specifies a frequency (a 4-bit field in our implementation), which is converted to the probability that the branch should be taken using the formula: $(\frac{1}{2})^{freq+1}$.

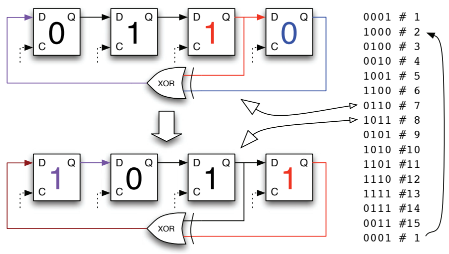


Figure 6. Updating an LFSR. A 4-bit LFSR with the right two bits selected for XOR will update from the value 0110 to 1011. All bits are shifted right on an update except the left-most bit which gets the result of the XOR. A 4-bit LFSR cycles through 15 possible values except 0.

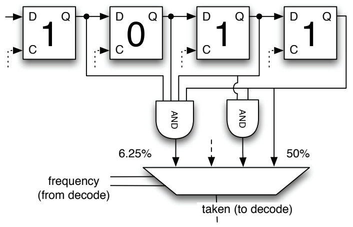


Figure 7. Generating probabilities from an LFSR. By treating each bit in the LFSR as a random value, ANDing various numbers of bits resulting in a value of 1 represents a taken branch for the corresponding frequency. A mux can be used to select the desired frequency of the branch. Only a subset of AND gates are shown to avoid a clutter of wires and gates.

3.3 Implementation

In this section, we describe a microarchitecturally simple implementation of branch-on-random. We discuss the additional hardware that is needed for the randomness, the desirable properties of branch-on-random that allow it to be fast, and the expected performance of this implementation.

Achieving Randomness: Our proposed implementation of branch-on-random uses a simple hardware pseudo-random number generator called a linear feedback shift register (LFSR) [14]. Each bit of the LFSR, implemented as a D-type flip-flop as part of a shift register, shifts to the next position on an update except for the first bit, which is computed by XORing a set of bits from the LFSR. Choosing the correct bits to XOR allows an n-bit LFSR to cycle through all 2^n values except 0. For the purpose of illustration, an example update of and sequence generated by a 4-bit LFSR is shown in Figure 6; to support frequencies of $(\frac{1}{2})^{16}$, the LFSR's used by branch-on-random will need to be at least 16 bits.

As an LFSR cycles through all possible values, individual bits can be treated as a random value of either 0 or 1, each with a probability of almost 50%.[2] This random bit is the component needed to generate the wide range of branch-on-random frequencies.

The first frequency required of the branch-on-random is 50%, and we have already explained that sourcing a single bit in the LFSR provides exactly that. If we treat each bit, which has a 50% probability of being 1, as an independent probability, the likelihood of two arbitrary bits from the LFSR both set to 1 is $50\% * 50\%$ or 25%. In other words, to get a value that is set to 1 approximately 25% of the time, we can AND any two bits from the LFSR. In general, the probability of x bits being all set to 1 is $(\frac{1}{2})^x$. With 15 AND gates we can compute all 16 supported frequencies in parallel, using the instruction's `freq` field to select the appropriate one to compute the branch outcome.

Since the random bits are not independent, some care needs to be taken in selective the bits to AND to provide some degree of independence between consecutive branch-on-random outcomes. For example, while ANDing two adjacent LFSR bits will correctly result in the branch being taken 25% of the time, the conditional probability of taking the branch given that the previous (25% frequency) branch was taken is 50%, because the one of bits is guaranteed to be one. While such a lack of independence between samples has not, in a statistically significant manner, impacted the quality of information we have been able to collect for the profiling applications we have tried (data not shown), we believe it to be an undesirable property that potentially impacts other applications. Furthermore, it can be mitigated in a straight-forward and inexpensive manner, by ANDing non-contiguous bits with varied spacing (*e.g.*, selecting bits 0, 2, 5, and 9 to compute a 6.25% probability). Providing some spacing even when many bits are ANDed (for very low probabilities) requires extending the LFSR beyond 16 bits. Given that applications requiring very low sampling rates ($(\frac{1}{2})^{16}$) are likely rare, a 20-bit LFSR may be a reasonable design point.

Pipeline Integration: In discussing the branch-on-random's implementation in the pipeline, it is useful to consider how conditional and unconditional branches are implemented. Conditional branches are taken based on value (either registers or condition codes) set by previous instructions, so they necessarily need to be resolved in the back-end of the pipeline. Unconditional direct branches do not depend on other instructions; and because of that, they can be resolved much earlier in a pipeline, specifically at decode time. While a branch-on-random has multiple potential targets (like a conditional branch), it is like an unconditional branch in that it does not depend on other instructions. As a result, it is possible to resolve them early—as soon as they are decoded. Resolving branch-on-random in decode reduces the misprediction penalty when the branch is taken.

We propose that branch-on-random be evaluated using the same datapath for handling unconditional branches. In parallel with computing the branch target, the branch-on-random condition must be evaluated. As noted above, the branch outcome is computed by ANDing a number of bits from the LFSR as shown in Figure 7. This ANDing can be performed in parallel with extracting the frequency field from a branch-on-random. The frequency field drives a 16-input mux that selects the appropriate AND gate output to determine whether the branch should be taken or not. We believe that these circuit paths will be sufficiently short that the branch outcome can be computed before the branch target has been computed, meaning that the branch-on-random hardware will not extend the cycle time of the decode stage. If necessary, the circuit can be pipelined such that the outputs of the AND gates are computed in the previous cycle and latched. To minimize the power consumption, the LFSR is only clocked on cycles in which it is used.

When implementing branch-on-random in a superscalar machine, the simplest solution is to replicate the hardware at each instruction decoder. As each branch is logically independent, it is architecturally valid to use multiple, completely decoupled LFSRs. Furthermore, given the small amount of hardware necessary to implement branch-on-random, this is not entirely unreasonable.[3]

[2] An LFSR cannot be all 0s, so an n-bit LFSR actually goes through $2^n - 1$ values, with each bit set to 1 for 2^{n-1} of the values. Thus, the likelihood for any bit to be 1 is $\frac{2^{(n-1)}}{(2^n)-1}$. With n=16, the probability is 0.5000076, which is close enough to $\frac{1}{2}$ for most practical purposes.

[3] Given that it is unlikely that multiple branch-on-random instructions will be in the same fetch packet, we can also design the machine such that multiple decoders can arbitrate (using a priority encoder based on program order) for a single LFSR. If more branch-on-randoms are present in a fetch packet than LFSRs, the fetch packet will have to be split, with the additional branch-on-randoms decoded the following cycle.

Prediction and Expected Performance: Branch prediction is, in general, not effective for predicting the outcome of branch-on-random instructions, because the sequence is sufficiently long that it appears truly random to a history-based predictor. As a result, our goal is to avoid mispredictions for the not-taken case and minimize the penalty for the taken case. We accomplish this by forcing these branches to always be predicted not-taken, by never entering them into the branch prediction hardware (*e.g.*, branch predictor table, branch target buffer (BTB), path history table). A beneficial side effect of this is that these structures are then not polluted by sampling branches, avoiding aliasing and conflicts in tables and wasting entries in the branch history.[4] Because branch-on-random will generally be used with low sampling rates (*e.g.*, less than 10% and often below 1%), the short misprediction penalty (discussed below) generally does not significantly impact performance.

By forcing the branch to be predicted as not-taken, a branch-on-random that is resolved to be not taken (in decode) need perform no further action and, hence, can be committed at decode time. It is safe to commit this instruction early, as it has no side-effects; it does not change the data state of architecturally-visible registers or memory. Even though speculative execution can result in unnecessary LFSR updates when resolving a branch-on-random on the wrong path, it is not necessary to checkpoint the LFSR state. Unlike a counter, losing LFSR transitions does not impact the probabilities it generates. This hardware simplification is enabled by not architecting a particular sequence of branch outcomes.

When a branch-on-random *is* taken, fetch has to be re-directed to the branch target, resulting in a small performance overhead. When a branch is taken, the wrong path instructions fetched between when the branch was fetched and when it is resolved need to be flushed. By resolving the branch in the decode stage, this "front-end misprediction" will generally have significantly less impact than the "back-end misprediction" associated with standard conditional and indirect branches. Notably, these short mispredictions are detected before register renaming, so there is no need for recovering register rename state.

Looking back at the sources of overhead of software counter-based sampling presented in Section 2, many of them (items 2 through 4 and 6) have been eliminated with a branch-on-random-based implementation, and those that remain have been reduced. The first source of overhead is reduced, as branch-on-random sampling framework consists of a single instruction. The second, third, and fourth are eliminated, as this instruction is completed at decode and uses no (architecturally visible) registers or memory. The fifth source of overhead is reduced because, when a sample is to be collected, only the penalty of a front-end misprediction is paid, not a full pipeline squash. Finally, the sixth source of overhead is eliminated, as branch-on-random does not pollute the predictors.

Summary: Implementing branch-on-random requires: 1) extending the existing decoder to recognize the new instruction, 2) using the existing branch target computation datapath to compute its target, 3) adding an LFSR that is clocked at the end of a cycle when a branch-on-random is decoded, 4) 15 AND gates, one of each size from 2 to 16 inputs, 5) a 16-input multiplexer controlled directly from a field in the instruction, 6) overloading the existing front-end misprediction path used for unconditional branches, and 7) adding control logic to prevent inserting a BTB entry on a branch-on-random misprediction. Thus, for a single-issue machine, we estimate branch-on-random can be implemented with roughly 20 bits of state (for the LFSR) and less than 100 gates. For a superscalar machine where all decoders support unconditional branches, this logic scales linearly with the decode width; for a 4-wide superscalar, branch-on-random should contribute less than 100 bits of state and less than 400 gates. Given that branch-on-random provides the opportunity to add profiling code to any executable with almost no overhead, we believe that this small amount of hardware is justifiable in future microprocessors.

3.4 Deterministic Implementations

During actual program execution, it is generally not important that branch-on-randoms follow any proscribed sequence; the non-deterministic implementation described above exploits this freedom to simplify the hardware. For testing purposes, however, it can be beneficial to have deterministic behavior, even for branch-on-random. In particular, hardware vendors have found that building completely observable, deterministic processor implementations enable them to greatly accelerate the process of post-silicon validation [28]. In this section, we describe the minor additions required for a deterministic implementation of branch-on-random.

The main source of non-determinism for branch-on-random derives from the fact that the LFSR is speculatively updated (in the decode stage), which cause LFSR transitions to be lost when branch-on-random instructions are squashed due to branch mispredictions or exceptions. This can be avoided by "checkpointing" the LFSR and restoring it after a squash. Because the LFSR updates are shifts, we can recover its previous state by simply allocating additional storage for the bits that would have shifted off the end of the LFSR (one additional bit per speculative branch-on-random allowed) and shifting back. Obviously the system must track how many bits to shift back and this can be performed either by maintaining a counter of number of branch-on-randoms that have been decoded — this counter needs precision only to the number of branch-on-randoms that can be in flight — that is checkpointed with other rename state that gets checkpointed; alternatively, where a "reverse-execution" technique is used to recover rename tables (*e.g.*, for exceptions in the MIPS R10000 [32]), we can shift back once each time a branch-on-random is removed from the ROB.

For the purposes of hardware testing it is not necessary to expose the LFSR in the architecture. The LFSR can be hooked up to an existing scan chain present for enabling chip testers to read/write the contents of all of a processor's registers; the only cost would be extending the scan chain.

Determinism can facilitate software debugging as well, but there are a number of existing challenges facing deterministic application execution, at least in the context of multithreaded applications. Nevertheless, deterministic branch-on-random behavior for applications has different requirements than for hardware debugging; in particular, the LFSR state must be readable and writable by software, so that it can be initialized by the application to a known value and saved/restored on context switches. While this does have some additional cost, if the LFSR can be read efficiently by application software it can be used as a very fast pseudo-random number generator by randomized algorithms [25].

Alternatively, if full-speed execution is not required during software testing, a fully-deterministic branch-on-random can be emulated in software, irrespective of whether the hardware implementation is deterministic or not. One approach would be to substitute an invalid opcode when compiling branch-on-random instructions during testing.[5] Then a signal handler for invalid opcodes could be registered that would emulate the branch-on-random in software, using LFSR state stored in thread-local storage. Alterna-

[4]For infrequently-executed unconditional jumps, such as those in Figure 4, branch-on-random could encode a 100%-taken frequency that avoids interfering with the BTB.

[5]This approach is actually how we collect the accuracy results presented in Section 4 on real machines.

tively, if re-compiling was thought to be too tedious, a mode could be provided (as part of processor state) that would treat branch-on-random instructions as invalid opcodes (invoking an exception handler, as above).

4. ACCURACY

In this section, we validate that the branch-on-random's behavior is suitable for instrumentation sampling. We consider the branch-on-random's sampling suitable if it achieves accuracy similar to or better than software counter-based sampling. In particular, we demonstrate that no idiosyncrasies of the LFSR's behavior prevent it from being suitable for use in sampling program behaviors. We modify Jikes RVM [1], a Java-in-Java virtual machine, to emit branch-on-random instructions and functionally execute them. We measure accuracy by comparing a method invocation profile collected via each sampling method with a full profile.

The results from the functional accuracy tests for branch-on-random provide three interesting results. The first shows that using an LFSR-based branch-on-random provides similar accuracy compared to existing counter-based sampling that samples exactly at every fixed interval. The second is that the LFSR's pattern naturally avoids the problem of falling into a pattern with the code. Finally, the process of modifying Jikes for these tests was straightforward, reflecting the simplicity of using the branch-on-random ISA extension.

4.1 Experimental Method

The Arnold-Ryder framework is a compile-time transformation that takes Java code previously instrumented by an earlier compiler phase and converts the instrumentation into profile sampling. It does this by adding a decrementing global counter that then branches to instrumentation code when the counter reaches zero, at which point the counter is reset to the sampling interval. An example of this was shown earlier in Figure 1.

It was straightforward to extend the Arnold-Ryder framework (already implemented in Jikes) to use branch-on-random sampling. To sample the instrumentation, instead of adding a counter load, check, reset, and decrement as well as allocating memory for the counter and reset value, it only needs to add a single branch-on-random instruction. The only difficulty in using branch-on-random, is that it required us to change the code layout for the instrumentation. As implemented in Jikes, the Arnold-Ryder framework places the instrumentation code in line and jumps over it when samples are not taken. In contrast, usage of branch-on-random necessitates that the instrumentation code is place out of line, because a low overhead implementation of branch-on-random necessitates that the common case branch outcome be fall through. Thus, we modified Jikes's Arnold-Ryder framework to place the instrumentation code at the end of the method. After executing the profiling code, the code unconditionally jumps back to the normal code order. We illustrate this change with a sample control-flow graph in Figure 8.

To evaluate the quality of the profile information collected via sampling, it is necessary to run the benchmarks sufficiently long that enough profile information is collected to make a reasonable comparison between methods. As a result, we wanted to run the DaCapo benchmarks (2006-10-MR2) [4] to completion using their "default" data set size. To achieve these long simulation runs using an instruction that our machines do not natively support, we ran the benchmarks on real machines, functionally emulating branch-on-random using signal handlers. For these experiments, we had Jikes emit an invalid opcode for the branch-on-random followed by 4 bytes for a branch offset. We registered a signal handler for

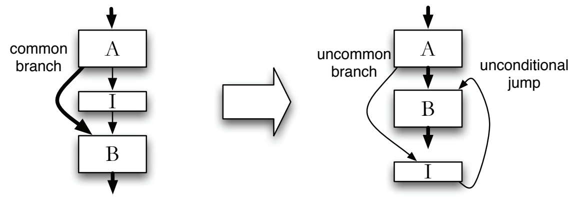


Figure 8. Rearranging code blocks. Instead of branching over the instrumentation code (taking the thick line) in the common case, we flip the condition, so fetching in order will only incur misprediction stalls and unconditionally jump back in the uncommon case.

SIGILL, the illegal instruction exception, in the C program that Jikes uses to boot the rest of the virtual machine (written in Java). When our invalid opcode is encountered, the O/S calls our signal handler which functionally emulates a branch-on-random by simulating an LFSR in software; based on the LFSR state, the signal handler either updates the PC to the fall-through instruction or adds the branch offset to the PC to redirect control flow to the branch target. This implementation runs at near full speed on a real machine and is transparent to the JVM above.

We ran the DaCapo benchmarks on Jikes to collect a profile of method invocations for accuracy comparisons between branch-on-random sampling and counter-based sampling. To compare accuracy of each sampling technique, we calculate the *overlap percentage* [3] by first collecting the full profile of method invocations and recording the relative method percentages. We repeat the same process with each sampling method. To compute the accuracy, we use the following expression:

$$accuracy = \sum_{i=1}^{N} min(f_{full}(i), f_{sampled}(i))$$

where $f_{full}(i)$ and $f_{sampled}(i)$ are the fraction of all of the collected samples that were for method i in the full and sampled profiles, respectively. For example, if `method1` is accounts for 50% of the method calls in the full profile while sampling reports it accounts for 60%, the method contributes 50% to the profile's accuracy. Because the sampling overcounted `method1`, other method percentages must be under-counted. A perfect sampling would result in an accuracy of 100%.

To compare the differences between deterministic sampling and random sampling, we also implemented the branch-on-random to be taken at defined intervals; for example, take every 1024th branch. This is essentially a hardware counter triggered by the branch-on-random instruction, so we will refer to it as a hardware counter in the results.

4.2 Results

In this section, we show that branch-on-random is accurate. We first compare it to both hardware and software counters of the same sampling frequencies and result in comparable and sometimes better accuracy. We then test various configurations of branch-on-random implemented with LFSRs to look at the sensitivity of changing various properties of the LFSR.

Because the accuracy of sampling is directly tied to the number of items being sampled, we sort the DaCapo benchmarks based on their total number of method invocations at size "default." The following is the order of benchmarks with the invocation counts in millions: `fop` (7), `antlr` (17), `bloat` (93), `lusearch` (108), `xalan` (109), `jython` (170), `pmd` (195), `luindex` (212).[6]

[6] We do not include results for `chart`, `eclipse`, and `hsqldb`, as these benchmarks would not run on Jikes for our machines (with or without the sampling framework), resulting from failure to access dependencies in the classpath or out-of-memory errors.

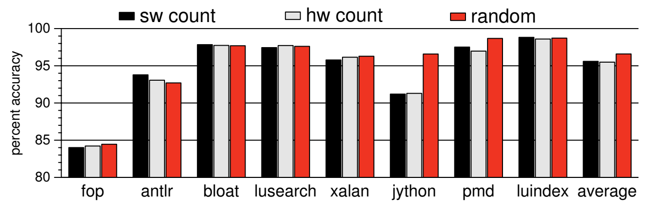


Figure 9. Sampling accuracy at 2^{10}. Random sampling of method invocations has comparable accuracy to those with software or hardware counters. The randomness avoids matching the program behavior in `jython` where counter accuracy suffers.

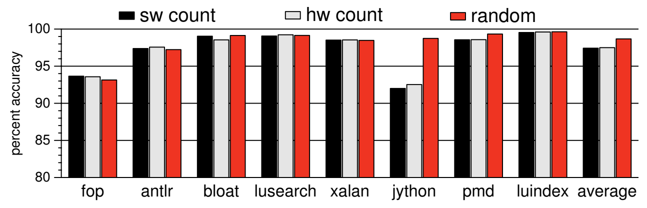


Figure 10. Sampling accuracy at 2^{13}. Decreasing the number of samples by a factor of 8 results in similar accuracy trends as those in Figure 9 except that everything is lower. Branch-on-random accuracy shows its resonance-avoiding ability in `jython` and `pmd`.

Frequency analysis

We first show the accuracies of the branch-on-random with a frequency of $(\frac{1}{2})^{10}$ and counter-based samplings with an interval of 1024 (2^{10}) in Figure 9. Overall, we find the profile quality achieved by all of the techniques to be pretty similar. The universally lower accuracies of `fop` and `antlr` are the result of a small number of samples being collected for these benchmarks by all three approaches.

`Jython` is a noticeable outlier, where branch-on-random's accuracy is almost 7% higher than the two counter methods. In this benchmark, the counter-based methods resonate with the test program. For example, a loop body containing calls to two leaf methods will result in only one of the two methods getting sampled for a counter-based sampling interval that is a multiple of two.[7] The pseudo-randomness of branch-on-random, however, automatically varies the intervals between samples. While software techniques have been employed to avoid this problem [2] in counter-based sampling, branch-on-random inherently eliminates this problem, so users do not even need to think about it in the first place.

We find a similar set of results (shown in Figure 10) when reducing the sampling rate to 1 per 8192 (2^{13}). Once again, `jython` performs poorly with the counters, but now it is easier to see that `pmd` also shows some of this pathological behavior.

Sensitivity analysis

We performed sensitivity analyses with respect to two aspects of the LFSR design: (1) the selection of bits to `XOR` for generating the sequence of pseudo-random numbers and (2) the selection of bits to `AND` for computing each probability (as discussed in Section 3.3). For this profiling application, we found the branch-on-random-based sampling framework to be robust to both the particular LFSR sequence generated and to which bits of the LFSR register are sampled, allowing the LFSR implementation to be selected for implementation ease. In particular, the variation that we observed in each of these experiments was not statistically significant relative to the distribution of results achieved from initializing the LFSR with different values. Having already discussed the latter result, we briefly relate the former.

For a particular length LFSR, there can be many sequences of pseudo-random numbers that cycle through all the possible values. The produced sequence depends on which bits of the LFSR are chosen to be `XOR`ed to compute the input to the LFSR (bit 0). We compared the profile information for four configurations of a 32-bit LFSR—two with four "taps" at bits (32, 31, 30, 10) and (32, 19, 18, 13), and two with six "taps" at bits (32, 31, 30, 29, 28, 22) and (32, 22, 16, 15, 12, 11)—and found variation in the profile quality below the level of significance.

5. OVERHEAD

In this section, we compare the overhead of a branch-on-random-based sampling framework to a counter-based one using timing simulation. We use DaCapo benchmarks running on Jikes RVM to explore these overheads in real applications and a microbenchmark to perform detailed analysis of branch-on-random's performance characteristics. We find that branch-on-random can achieve an order-of-magnitude reduction in overhead from software-only frameworks, with the overhead of each instrumentation site assymptotically approaching .1 cycle (on a 4-wide out-of-order machine) as the sampling rate is decreased.

5.1 Experimental Method

To support the JVM-based workloads in timing simulation, we have developed an x86-based timing simulation integrated into the Simics full-system simulator. This simulator uses the timing-first approach [22], where the timing simulator runs ahead and uses a "golden" functional model (Simics [31]) to verify the results produced by instructions as they commit. This approach to simulation permits full-system simulation without having to build a 100% functionally-correct timing simulator, while preserving the ability to perform arbitrary speculative execution. The functional model included in our timing simulator is derived from PTLsim [33], using PTLsim's decomposition of x86 instructions into micro-ops.

We configured our simulator to be a 4-wide (decode, execute, retire) out-of-order processor with a 80-entry reorder-buffer. The front end can fetch up to three x86 instruction per cycle, but stops fetch at a predicted taken branch. Its branch predictor is a tournament predictor with a 16-bit gshare and a 64k-entry bimodal predictor, and it includes a 32-entry RAS and a 1024-entry branch target buffer (BTB). The minimum (back-end) misprediction penalty is 11 cycles. The L1 caches are 32KB, 4-way set-associative with 64-byte blocks. The shared L2 cache is 1MB, 8-way set-associative and responds in 8 cycles, and memory responds in 140 cycles.

We extended our simulator to support branch-on-random by overloading the Simics *magic instruction*. Jikes's code generator emits a branch-on-random as a magic instruction followed by a 4-byte branch offset. Our Simics extension module supports this instruction both in pure-functional simulation mode (so that we can fast forward our workloads to the point we want to sample) and in the timing simulator. In timing simulation mode, the timing simulator (as the leading simulator) is responsible for functionally simulating the branch-on-random and communicating its computed outcome to Simics so that both simulators compute the same outcome. Branch-on-random instructions are resolved in the decode stage, the 5th stage of the pipeline.

As we need to compare different versions of the same code (a baseline version with no instrumentation and versions with branch-

[7] For example, for an interval of 2, if the first method is sampled, the second method will not sample and instead decrement the counter; however, now the counter indicates that the next method will be sampled, which happens to be the first method again. In general, this affects any fixed interval that is a multiple of a recurring pattern of sampling sites such as chains of nested method calls in a loop or sequences of method calls.

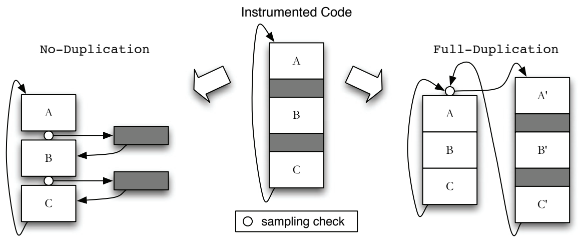


Figure 11. Two transformations provided by the Arnold-Ryder framework. `No-Duplication` adds a sampling site (the circle) before each shaded instrumentation block. `Full-Duplication` duplicates the code with backedges pointing to the original, removes the instrumentation from the original, and adds a check at every method entry and loop back-edge to determine which version of code should be executed.

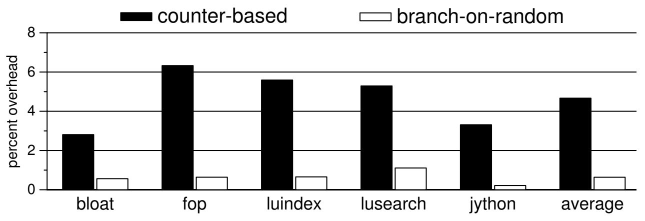


Figure 12. Overhead of two sampling frameworks for DaCapo benchmarks running on Jikes. Software **counter-based** sampling (using `Full-Duplication`) averages almost a 5% overhead on these weakly-optimized benchmarks, while the **branch-on-random**-based framework achieves a 0.64% overhead. Performance is normalized to a non-instrumented version of the code, and both experiments use a sampling period of 1024.

on-random and counter-based sampling frameworks), we need a method for identifying an equivalent region of the execution for simulation in each of these executables. To support this in our Java-based workloads, we have implemented the ability in Jikes to insert markers in specified methods [26]. Again, we use the Simics magic instruction (this time with a zero branch offset to distinguish it from a branch-on-random) for these markers. Our Simics extension module keeps track of the number of these marker instructions executed, beginning warmup, beginning simulation, and ending simulation at specified marker counts.

5.2 Application Results

Our goal in this evaluation is to understand the relative performance of the branch-on-random framework relative to counter-based sampling. As the overhead of the instrumentation code is orthogonal to the framework overhead, we simply configured Jikes to instrument method execution frequencies for these experiments, a profiling technique that introduces a relatively small number of instrumentation sites into the code. By using a sampling rate of one sample per 1024 executions of an instrumentation site, we largely eliminate the overhead of the instrumentation itself, exposing the overhead of the sampling framework. In the next section, we consider how the overhead changes with sampling rate.

To demonstrate that branch-on-random can improve performance with respect to the best software sampling techniques, we ran these experiments using Arnold-Ryder's `Full-Duplication` approach [3]. As shown in Figure 11, `Full-Duplication` amortizes the cost of the sampling framework across all of the instrumentation sites within a given acyclic region of the program. Specifically, it replicates every instrumented code region to produce one version that contains instrumentation and one that does not, using counter-based sampling to select the appropriate version on entry to the region. To ease the measurement of the profiling overhead, we turn Jikes's adaptive optimization off, so that all code runs using the baseline compiler with instrumentation for the full run. Since the application code is less highly optimized, our measurements may underestimate the absolute overhead, but the relative overheads should be representative.

We find that across the benchmarks,[8] branch-on-random provides almost an order-of-magnitude reduction in the overhead of the sampling framework, on average (as shown in Figure 12). The main contributors to the counter-based sampling frameworks overhead not present in the branch-on-random are largely accounted for by two factors: 1) the overhead of fetching and executing the additional instructions in the counter-based approach, and 2) additional branch mispredictions resulting from the counter-based approach. Interestingly, the branch mispredictions result from two sources: 1) sampling branches that are incorrectly predicted as taken due to aliasing in the predictor, and 2) non-sampling branches whose prediction accuracy is impacted by the effective reduction in branch history due to the (low-entropy) sampling branches.

While useful for understanding branch-on-random's behavior in applications, the small non-determinisms present from simulating differently-compiled versions of this multi-threaded workload make it difficult to use these applications to do detailed analysis of the two frameworks. In the next section, we use a single-threaded microbenchmark to cleanly isolate the overhead of branch-on-random on a per-instrumentation site basis.

5.3 Microbenchmark Results

In an attempt to precisely measure the overhead of sampling frameworks, we created a microbenchmark that computes checksums and character distributions of half a million characters stored in memory. There are multiple execution paths in the character processing loop that conditionally update the checksum for upper-case, lower-case, and the other remaining characters. By adding instrumentation, we can collect edge profiles to compute branch biases.

To minimize the perturbations on the generated code, we compile the benchmark once with `gcc -O2` to an x86 assembly file and post-process the assembly to insert the different instrumentation implementations. With that file, we modify the instructions to generate code that implements `no-instrumentation`, `full-instrumentation`, and several versions of `sampled-instrumentation` for both branch-on-random and counter-based sampling with a wide range of sampling intervals. Directly modifying the assembly in this way ensures that all the benchmark binaries are generated with the same instructions, register usage, stack allocations, and code layout. With the assembler as a constant, we are able to constrain the execution time variations to just the differences between sampling techniques.

To provide the reader some context of this microbenchmark, we collected some statistics on the character processing loop in the non-instrumented version (for all of our experiments we exclude the program's prologue and epilogue from timing simulation). In the baseline code, the branch-prediction accuracy is 84.5%, which results from the frequent data-dependent branches: the character

[8] We were unable to get results for `antlr` and `xalan` due to the non-determinism of the testing framework preventing us from reliably sampling the same portion of execution in the different versions. Because the JVM is a multithreaded program (as are most of the DaCapo benchmarks), variations in the instruction path lengths among the benchmarks cause interrupts to be taken at different program points, leading to different thread interleavings and different heap allocations (as garbage collection is performed at different times). `pmd` would not run at all for us on Jikes/Simics.

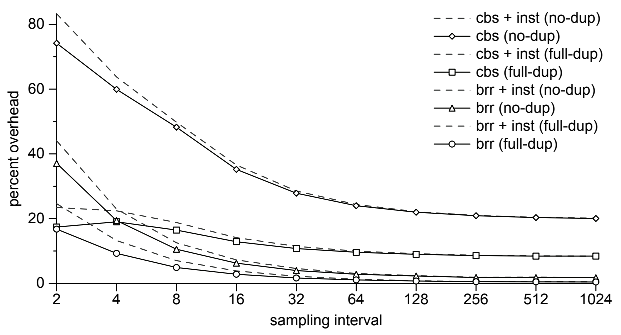


Figure 13. Overhead of different frameworks. The overheads for branch-on-random ends up lower than that of counter-based for the more interesting frequencies. Both implementations benefit from using `Full-Duplication` over `No-Duplication`. The solid lines come from running simulations with only the framework while the dashed lines also include the instrumentation.

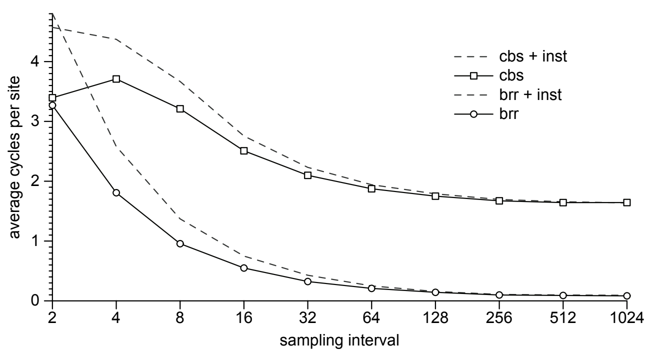


Figure 14. Average cost of a sampling site. Branch-on-random averages 10 to 20 times fewer cycles than counter-based sampling for frequencies above 64. We only plot the results of `Full-Duplication` to avoid the clutter of showing `No-Duplication`, which has qualitatively similar results.

stream of Shakespearian plays has words that are all upper-case or all lower-case. In addition, both data and instruction caches hit over 99.5% of the time. As a result, the pipeline is fetching its maximum of 4 instructions 67% of the time and is handling branch mispredictions 29.5% of the time. The front-end is idle 1.3% due to a full reorder buffer and 6.1% for an empty fetch-queue flushes, but the remaining time, it is running at full speed.

For sampling intervals above 64, we find the sampling overhead from using branch-on-random is an order of magnitude less than the overhead from using counter-based sampling. The curves in Figure 13 show the percent overhead compared to the baseline for the four combinations of branch-on-random and counter-based sampling with `No-Duplication` and `Full-Duplication`. The lines show the overhead of branch-on-random decreasing much faster and further than counter-based as we increase the sampling interval. Branch-on-random's framework overhead decreases because fewer taken branch-on-randoms result in less time redirecting on early branch mispredictions.

These results also show that using `Full-Duplication` lowers the framework overhead of not only the counter-based but also the branch-on-random sampling techniques. Relative to `No-Duplication`, `Full-Duplication` reduces both the number of sampling sites as well as the number of branches taken to instrumentation, so the total number of branch mispredictions also decreases. Even though counter-based sampling benefits a lot from moving to `Full-Duplication`, branch-on-random also benefits and maintains its order-of-magnitude overhead reduction relative to counter-based sampling.

As the overhead of the sampling framework is a function of how often instrumentation sites are encountered, we report the overhead in cycles per instrumentation site. We do this by computing the net increase of simulation cycles (due to instrumentation and sampling) by subtracting the cycles it takes to execute the baseline from each simulation. This allows us to compute an average number of cycles for each dynamically-encountered sampling site.

The average number of cycles used for each sampling site for both branch-on-random sampling and counter-based sampling using `Full-Duplication` is shown in Figure 14. For a 50% branch-on-random, 3.19 cycles are spent per site, which intuitively makes sense as the cost corresponds to 50% of the 5-cycle pipeline flush plus the overhead of having 2 extra instructions in the program stream (the branch-on-random and the unconditional jump back from instrumentation).

The counter-based sampling sites include a couple of memory accesses and a backend-resolved branch, which flushes the whole pipeline if mispredicted; as a result its lower-bound is much higher than branch-on-random's. The reason that the counter-based implementation actually has lower overhead sampling at a rate of 2 than 4 comes from the hardware correctly predicting the counter branches for the former, but larger intervals prevents the pattern from fitting in the branch predictor's global history. Comparing the two sampling techniques at an interval of 1024, we see branch-on-random going 20 times faster; but in general, it performs 10 to 20 times faster in the interesting interval ranges.

In both Figure 13 and Figure 14, we show the overhead including the instrumentation costs with dashed lines. Obviously, the overhead of the instrumentation is a direct function of the complexity of the instrumentation; for this simple microbenchmark, we use a relatively low overhead instrumentation, but it can still be clearly seen how decreasing the sampling rate directly reduces this source of overhead. As a reference, `full-instrumentation` without sampling adds an average 4.3 cycles to the baseline per instrumentation site.

6. RELATED WORK

In this section, we briefly describe previous approaches for hardware support for profiling and descriptions of existing uses of LFSRs.

Many sampling techniques make use of hardware to some extent to reduce the profiling overhead. DIGITAL Continuous Profiling Infrastructure (DCPI) leverages simple event counters to diagnose the presence and source of performance problems [2] and to seed the invocation of an emulation-based value profiler [7]. ProfileMe propose longitudinal profiling, which captures all of the events relating to a single instruction's execution, in part to enable DCPI-like tools for out-of-order microarchitectures [11]. The relational profiling architecture (RPA) enabled annotating instructions with the type of profile information that should be collected, and the off-loading of the profile processing to special- and general-purpose cores [16]. Programmable profiling co-processors have even been proposed to provide flexible profile collection without slowing down program execution [9].

In addition to their applications in cryptography and communications, LFSRs have been previously proposed in computer architecture, microarchitecture, and testing. Pseudo-random values can probabilistically update counters to reduce the size of predictors [27]. Cache coherence can use a dynamic approach as in Bandwidth Adaptive Snooping Hybrid (BASH) to leverage the LFSR as a source of randomness to choose between unicast or broadcast [12]. Region-Scout is another cache coherence optimization that uses an array of LFSRs to record locally-cached regions [24]. LFSRs also help compress the inputs of built-in self-tests by allowing designers to choose seed values used to functionally generate test vectors [20].

7. CONCLUSION

As the complexity of software systems continues to grow and society's use and reliance on these systems grows as well, it is desirable to develop techniques to permit software systems to configure, monitor, optimize, and repair themselves, without human intervention [19]. In order to achieve any such "autonomic" behaviors, a software system will need to continuously collect information about its activities, and this information collection will have its costs.

Branch-on-random, however, brings us a step closer to building autonomic systems by providing an order-of-magnitude reduction in the cost of implementing sampling frameworks for instrumentation. We have described how to architect and implement branch-on-random with minimal hardware complexity and demonstrated that branch-on-random is simple to use. Additionally, compared to existing frameworks, it achieves equivalent accuracy yet provides significant reductions in overhead. Together, branch-on-random and performance counters provide a complete, inexpensive solution that allows profiling both high-level and instruction-level behavior in even the most performance critical parts of production code.

In addition, because each branch-on-random instruction encodes its own frequency, it is possible to efficiently implement convergent profiling [8], by modifying the sampling frequency as information is collected. In convergent profiling, a high sampling rate is used initially, but as the profile "converges" the sampling rate can be reduced, as we merely need to validate that program behavior continues as we have characterized it. If the low frequency samples appear out of line with the characterization, sampling rates can be increased to re-characterize the behavior.

Finally, branch-on-random can potentially be used in non-profiling situations, anywhere a choice needs to be made and statistical behavior is acceptable. One example is in the CPython's cooperative multithreading; to support multithreaded programs on a non-thread safe interpreter, CPython releases the "global interpreter lock" after executing a specified number of bytecodes. A branch-on-random with a suitable frequency could replace the overhead of this counting. Another example is using branch-on-random to efficiently select among functionally-equivalent code versions to determine which is fastest [21]. Usage of branch-on-random could even assist in extraction of parallelism from sequential code as shown with `Y-branch` [6]. Invariably, it is impossible to anticipate all of the potential uses of a simple, but powerful mechanism like branch-on-random.

8. REFERENCES

[1] B. Alpern et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. 16th Symposium on Operating System Principles*, Oct. 1997.

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179, 2001.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[5] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for Experimenters*. John Wiley and Sons, 1978.

[6] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture*, Dec. 2007.

[7] M. Burrows, U. Erlingson, S.-T. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–167, Nov. 2000.

[8] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.

[9] Y. Chou and J. P. Shen. Instruction path coprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[10] J. Dean. Personal communication, Aug. 2007.

[11] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–302, Dec. 1997.

[12] K. Diefendorff. Power4 focuses on memory bandwidth. *Microprocessor Report*, 13(13):1–8, Oct. 1999.

[13] B. A. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[14] S. W. Golumb. *Shift Register Sequences*. Aegean Park Press, revised edition, 1982.

[15] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA)*, Oct. 2004.

[16] T. H. Heil and J. E. Smith. Relational profiling: Enabling thread level parallelism in virtual machines. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 281–290, Dec. 2000.

[17] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. Technical Report CS-TR-1994-1207, University of Wisconsin, Madison, 1994.

[18] Intel Corporation. VTune Performance Analyzer.

[19] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.

[20] B. Konemann. LFSR-coded test patterns for scan designs. In *Proceedings of European Test Conference*, pages 237–242, 1991.

[21] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: using hot optimizations without getting burned. *ACM SIGPLAN Notices*, 41(6):239–251, 2006.

[22] C. J. Mauer, M. D. Hill, and D. A. Wood. Full system timing-first simulation. In *Proceedings of the 2002 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 108–116, June 2002.

[23] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[24] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 234–245, 2005.

[25] R. Motwani and P. Raghavan. Randomized algorithms. *ACM Comput. Surv.*, 28(1):33–37, 1996.

[26] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 174–185, New York, NY, USA, 2007. ACM.

[27] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and stratification. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, Feb. 2006.

[28] I. Silas et al. System level validation of the Intel Pentium-M processor. *Intel Technology Journal*, 7(2):37–43, 2003.

[29] M. Smith. Overcoming the Challenges of Feedback-Directged Optiization. In *Proc. Proc. ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00)*, Jan. 2000.

[30] O. Traub, S. Schechter, , and M. D. Smith. Ephemeral instrumentation for lightweight program proling. Technical report, Harvard University, 1999.

[31] Virtutech AB. Simics full system simulator. http://www.simics.com/.

[32] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[33] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2007.

[34] C. B. Zilles and N. Neelakantam. Reactive techniques for controlling software speculation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.