# Probabilistic Counter Updates for Predictor Hysteresis and Stratification

Nicholas Riley Craig Zilles

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

## Abstract

*Hardware counters are a fundamental building block of modern high-performance processors. This paper explores two applications of probabilistic counter updates, in which the output of a pseudo-random number generator decides whether to perform a counter increment or decrement. First, we discuss a probabilistic implementation of counter hysteresis, whereby previously proposed branch confidence and criticality predictors can be reduced in size by factors of 2 and 3, respectively, with negligible impact on performance. Second, we build a frequency stratifier by making increment and decrement probabilities functions of the current counter value. The stratifier enables a 4-bit counter to classify an instruction's Likelihood of Criticality with sufficient accuracy to closely approximate the performance of an unbounded precision classifier. Because probabilistic updates are both simple and effective, we believe these ideas hold great promise for immediate use by industry, perhaps enabling the use of structures such as branch confidence predictors which may have previously been viewed as too expensive given their functionality.*

## 1. Introduction

Hardware counters are a fundamental component of modern high-performance processors. As many techniques for improving single-thread performance—especially in non-numeric programs—exploit repetition in program behavior, they require the means to characterize this behavior. Processors employ hardware counters to perform efficient on-line aggregation of multiple observations and identify dominant behavior in recent execution.

In most designs, counters are deterministically updated, *e.g.*, every observed positive feedback event triggers a counter value increment. We explore only performing counter updates in response to a subset of observations. Specifically, we propose inserting a simple pseudo-random number generator—a linear feedback shift register (LFSR)—whose output determines if a given observation should result in a counter update. LFSRs and the design of hardware counters with probabilistic updates are further discussed in Section 2.

We identify two applications of probabilistic updates. First, we show how probabilistic updates provide probabilistic hysteresis. As we discuss in Section 2, a number of proposed hardware counters use many bits per counter for hysteresis. By substituting probabilistic hysteresis, we can often significantly reduce counter sizes with little effect on predictor behavior. We demonstrate this application in two previously studied contexts: Section 3 introduces a 2-bit branch confidence predictor that performs similarly to a previously proposed 4-bit predictor, and Section 4 describes a 2-bit criticality predictor that performs similarly to a previously proposed 6-bit one.

Second, probabilistic updates can stratify instructions into different classes according to the frequency with which they exhibit a particular behavior. Not only can we make a binary prediction of whether an instruction will be critical, but we can classify an instruction based on its likelihood of being critical, allowing us to treat always-critical instructions differently than sometimes-critical instructions. The key enabler for instruction stratification is the use of different update probabilities at different counter values. Lower counter values are associated with a much higher probability of incrementing on positive feedback and a much lower probability of decrementing on negative feedback. We thereby select an operating point (ratio of positive to negative feedback) for each counter value; instructions tend to hover around the counter values that most closely approximate their operating point. In Section 5, we motivate the utility of such a stratification and demonstrate how a 4-bit counter can stratify an instruction's Likelihood of Criticality with performance comparable to an approach employing an unbounded-precision likelihood representation.

We conclude by discussing the limits of probabilistic updates. It appears at least one bit of real hysteresis, in addition to the bit(s) encoding the information being stored, is required to closely approximate a multi-bit predictor. Branch confidence and criticality predictor behavior notice-

ably changes between 2-bit and 1-bit counters, though the lost accuracy is not commensurate with the 50% storage reduction. Regardless, this limits the benefit of probabilistic updates applied to 2-bit counters, especially in aggressive branch predictors that have been scaled well past the point of linear reduction of misprediction rate with increased table size.

## 2. Hardware counter design

This section presents the organization of a hardware counter with probabilistic updates. We start by reviewing the design of traditional hardware counters and explaining why multi-bit counters can be useful. In Section 2.2, we introduce a notation for describing counters and how probabilistically-updating counters can approximate deterministic counters while requiring fewer bits of storage. Section 2.3 details the components of our probabilistically-updated hardware counter design.

### 2.1. Traditional hardware counters

Predictors of instruction behavior employ arrays of hardware counters because, while multiple executions of the same static instruction are generally similar, we want to independently track the behavior of different static instructions. To select a counter from an array, a hash function is applied to the instruction's address, perhaps including some historical information. A counter is updated by incrementing (decrementing) on positive (negative) feedback, and predictions are made by comparing the current counter value against a threshold. For example, each counter in Smith's 2-bit branch predictor [17] increments when a corresponding branch is resolved as taken and decrements when not taken; a branch is predicted as taken when the counter value exceeds a threshold of 1. Counter updates use saturating arithmetic to prevent values from wrapping.

While a branch predictor outputs a single bit of information (*taken* or *not taken*), 2-bit counters are used to provide hysteresis. One bit of hysteresis is generally sufficient for branch prediction, because it prevents a second misprediction when a branch occasionally executes against its bias direction—including the important case of a loop exit.

Some more complicated predictors can benefit from larger counters. Consider predictors used to decide whether to speculate, such as one proposed to control selective value prediction [3]. Unlike branch predictors, these predictors have inherently asymmetric misprediction penalties: the performance lost by speculating with an incorrect value (often incurring a pipeline flush) is significantly greater than the opportunity cost of not speculating if the suggested prediction would have been correct. This imbalance motivates a *biased* counter design, which conservatively prefers a cheaper false negative to a more expensive false positive.

Two attributes typify existing biased counter designs. First, the magnitudes of counter increment ($I$) and decrement ($D$) differ in order to adjust the minimum ratio of positive to negative (or negative to positive) feedback that saturates a counter in one direction. For an unbiased counter, this ratio is 50%. But by setting $I = 1$ and $D = 2$, we can require positive feedback at a rate above 66% ($\frac{D}{I+D}$) for the counter to saturate at its maximum value. Second, additional bits per counter store more history, enabling a moving average to be computed with a larger window. In this way, occasional feedback in the minority direction can be filtered out. Both of these attributes can be observed in the previously mentioned selective value prediction counter, which uses 4 bits of storage, $I = 1$ and $D = 7$.

### 2.2. Notation and probabilistic updates

We represent counters by 4-tuples, $\langle n, I, D, T \rangle$. An $n$-bit saturating counter's value increments by $I$ on a positive outcome, decrements by $D$ on a negative outcome, and predicts true if greater than the threshold value $T$, where possible values range from 0 to $2^n - 1$. Table 1 presents several previously proposed counters in this notation.

**Table 1. Some proposed multi-bit hardware counters.**

| Predictor | Counter |
|---|---|
| Smith's 2-bit branch [17] | $\langle 2, 1, 1, 1 \rangle$ |
| Calder *et al.*'s biased 4-bit value confidence; low, medium and high confidence thresholds $T$, respectively [3] | $\langle 4, 1, 7, \{2, 6, 14\} \rangle$ |
| Jacobsen *et al.*'s 4-bit resetting branch confidence estimator [7] | $\langle 4, 1, 15, T \rangle$ |
| Fields *et al.*'s biased 6-bit criticality [4] | $\langle 6, 8, 1, 8 \rangle$ |
| Torres *et al.*'s Non-Forwarding Store [18] | $\langle 3, 1, 1, 6 \rangle$ |

Deterministic updates constrain the elements of the 4-tuple to integral values. With probabilistic updates, we can set fractional values for $I$ and/or $D$. For example, $D = \frac{1}{4}$ gives the counter a 25% chance of being decremented on negative feedback. We can proportionally scale down a predictor by reducing its size by one bit and halving $I$, $D$ and $T$ (*e.g.*, $\langle 4, 1, 15, 14 \rangle \rightarrow \langle 3, \frac{1}{2}, 7, 6 \rangle \rightarrow \langle 2, \frac{1}{4}, 3, 2 \rangle$). In Sections 3 and 4, we show that this proportional scaling can be applied with little impact on predictor behavior.

### 2.3. Designing with probabilistic counter updates

Multi-bit counters are updated in a read-modify-write sequence. In traditional, deterministic counters, the current value of the counter is read, incremented by $I$ or decremented by $D$ (using saturating arithmetic) depending on the type of event, and written back to the counter, as depicted in Fig. 1(a). To implement probabilistic updates, *e.g.*, a 25% chance of decrementing on negative feedback denoted by

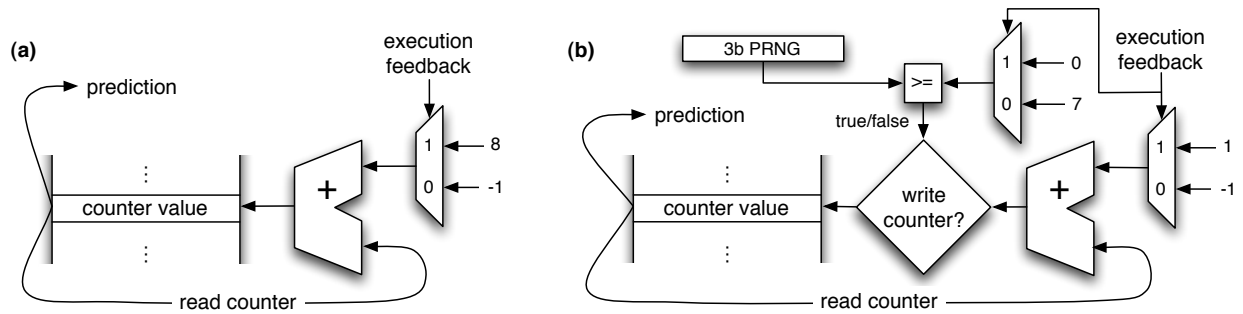


**Figure 1. The update path for deterministic and probabilistic multi-bit counters with the same increment and decrement ratio.** The deterministic counter (a) increments by eight and decrements by one; the probabilistic counter (b) increments by one and uses a 3-bit value read from a PRNG to decrement by one 12.5% of the time.

$D = \frac{1}{4}$, we introduce randomness into the counter update sequence.

Fig. 1(b) illustrates a probabilistic counter update mechanism. For fractional values of $I$ and $D$, the counter write is conditional. If the value read from a pseudo-random number generator (PRNG) exceeds a threshold value—if $I$ and $D$ differ and are both fractional, two separate threshold values are required—the counter is updated. Note that there is a single PRNG for the whole array, not one for each counter. While not shown in the figure, it may be possible to gate the entire read-modify-write process, saving power by reducing the frequency of counter reads and writes.

Our probabilistic counters require a source of pseudo-random numbers, but it does not have to be a particularly good one. Some applications may obtain a 12.5% probability by simply using a counter to select every eighth update. Using a PRNG with such a short period increases the likelihood of matching periodicity in the instruction stream, potentially always updating some instructions and never updating others. As a result, we believe a linear feedback shift register, whose period scales logarithmically with its size, represents a good cost-benefit tradeoff.

A linear feedback shift register (LFSR) [5] is a sequential shift register with logic that causes it to cycle through a pseudo-random sequence of values. The feedback in a LFSR is generated by the output of a set of its stages, called *taps*, which are XORed into a value shifted into the register. When optimally configured, an $n$-bit LFSR sequences through every bit pattern except all zeros, yielding a period of $2^n - 1$. To achieve this maximal period, an LFSR needs an even number of taps; the oldest bit is always tapped. An 8-bit LFSR with taps in a configuration with maximal period is shown in Fig. 2. If the LFSR exceeds the size of the required random number (*e.g.*, only a 3-bit number is needed in Fig. 1), any subset of the LFSR's bits can be used.

While LFSR behavior is extremely predictable, it is adequate for our purposes. A 32-bit Mersenne Twister-based PRNG [11] produced results that did not differ substantially from those obtained with a 16-bit LFSR. Given this result, we did not explore more sophisticated PRNG hardware.

## 3. Branch confidence

Our first set of experiments explores probabilistic variants of branch confidence estimators. These predictors classify a fetched branch as *high confidence* if it is likely to be correctly predicted, or *low confidence* otherwise. Applications of branch confidence predictors include pipeline gating [9]—trying to save power by reducing the number of wrong-path instructions fetched—and selecting low confidence branches at which to checkpoint processor state in large-window processors [1].

Branch confidence estimators exploit a key observation about branch predictor behavior: a branch which has been consistently predicted correctly is likely to continue to be predicted correctly. In light of this observation, Jacobsen *et al.* proposed a predictor—subsequently referred to as the JRS estimator—that tracks the number of correct branch predictions since the last misprediction [7]. The estimator's saturating counters increment on every correct prediction and *reset* on a branch misprediction; a 4-bit version is represented in our notation as $\langle 4, 1, 15, T \rangle$. In practice, the JRS estimator works quite well, isolating almost 90% of the mispredictions to just 25% of the dynamic branches.

An important consideration in the design of a branch confidence estimator is the tradeoff between accuracy and coverage. Increasing the high-confidence threshold (the number of correct predictions that must be observed before a branch is considered predictable) increases the fraction of high-confidence branches predicted correctly, but reduces the number of correctly-predicted branches labeled as high

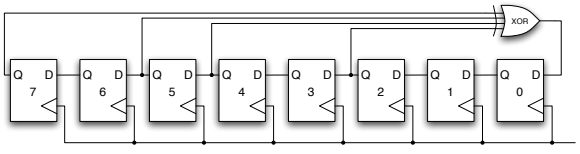


**Figure 2. An 8-bit LFSR.** Bits 2, 4, 5 and 7 are tapped.

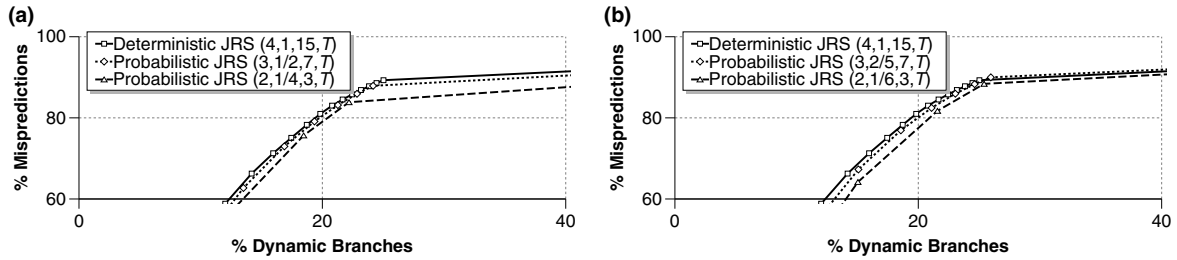


**Figure 3. Proportionally scaled probabilistic branch confidence counters have similar behavior to deterministic versions; non-proportionally scaled predictors can match their behavior.** Points correspond to different values of *T*.

confidence. With deterministic updates, increasing $T$ requires larger counters to store a larger number of correct predictions since the last misprediction. For example, the 2-bit determinstic JRS estimator $\langle 2, 1, 3, 2\rangle$ is only able to isolate about 60% of the branch mispredictions in the low confidence set.

Probabilistic updates allow us to decouple $T$ from the amount of storage allocated. We can approximate a branch confidence estimator that requires 15 correct predictions by ignoring half the correct predictions and requiring only 7 correct predictions. Results of applying this strategy are shown in Fig. 3, analagous to Jacobsen's Fig. 8 [7].

Here, we implemented branch confidence estimators for a *gshare* branch predictor with 16 bits of global history in a SimpleScalar-based functional simulator [2]. The simulated execution includes the first 4 billion instructions of each of the SPEC CINT2000 benchmarks. After each branch execution, we record the branch predictor correctness with the corresponding counter value for each estimator. From this information, we reconstruct the number of correct and incorrect high and low confidence predictions at each possible threshold $T$.

Fig. 3(a) includes linearly scaled 3-bit and 2-bit probabilistic counter variants of the 4-bit JRS estimator. While these variants closely track the deterministic estimator, the rightmost point falls somewhat short of the 4-bit version. This stems from two effects: we are forced to scale the threshold superlinearly ($\frac{14}{2} = 7$ but the highest threshold for a 3-bit counter is 6), and some imprecision is introduced by the probabilistic updates (*i.e.*, with $I = \frac{1}{2}$, a branch would need an average of 14 correct predictions, but could saturate with as few as 7). Nevertheless, nonlinearly scaled probabilistic counters can closely approximate the 4-bit predictor, as shown in Fig. 3(b). The nondeterministic versions require roughly 0.25% more dynamic branches to reach the same level of misprediction coverage.

We can also observe the behavior of probabilistic counters using the metrics introduced by Grunwald *et al.* [6]: sensitivity (SENS), the fraction of correct predictions identified as high confidence; predictive value of a positive test (PVP), the probability a high confidence estimate is correct; specificity (SPEC), the fraction of incorrect predictions identified as low confidence; and predictive value of a negative test (PVN), the probability a low confidence estimate is correct.

Different applications seek to optimize different metrics: consumers of branch confidence data use either PVP or PVN, and place varying importance on SENS and SPEC. Grunwald identifies PVN and SPEC as critical for pipeline gating, to avoid stalling the processor unnecessarily while maximizing power-saving opportunities. In contrast, selection of branches for checkpointing benefits from optimizing PVP and SENS, to minimize the likelihood that a misprediction happens at a non-checkpointed branch and to avoid running out of checkpoints (which would occur if we predicted too many branches as low confidence).

In Fig. 4(a), we plot PVP versus SENS for a range of predictors. We now use Grunwald *et al.*'s enhanced JRS estimator, which improves accuracy by appending the currently predicted branch outcome to the branch history used to index into the counter array. A point in the upper right-hand corner of this graph would correspond to an ideal predictor. Lines connect predictors of the same size, demonstrating a tradeoff between PVP and SENS. Probabilistic updates enable this tradeoff independent of predictor size. Only the topmost point on each line represents a deterministic predictor; the others represent predictors with probabilistic increments. A deterministic predictor with half the number of entries is shown for reference; it uses the same amount of space as the 2-bit predictors.

The "iso-size" curves in the figure shift slightly to the right (increased PVP) as the counter size increases from 1 to 4 bits. This trend enables estimators with 2- and 3-bit probabilistic counters to perform almost as well as the 4-bit (deterministic) predictor. Specifically, 2- and 3-bit counters with the same SENS as the 4-bit counter respectively exhibit only 0.08% and 0.02% reduction in PVP. The probabilistic 2-bit predictor significantly outperforms an equivalently-sized 4-bit deterministic predictor with half as many counters. There is a significant performance gap between the 1-bit and 2-bit counters, though both outperform deterministic counter arrays using an equivalent or lesser amount of storage.

Fig. 4(b) plots PVN versus SPEC, demonstrating the

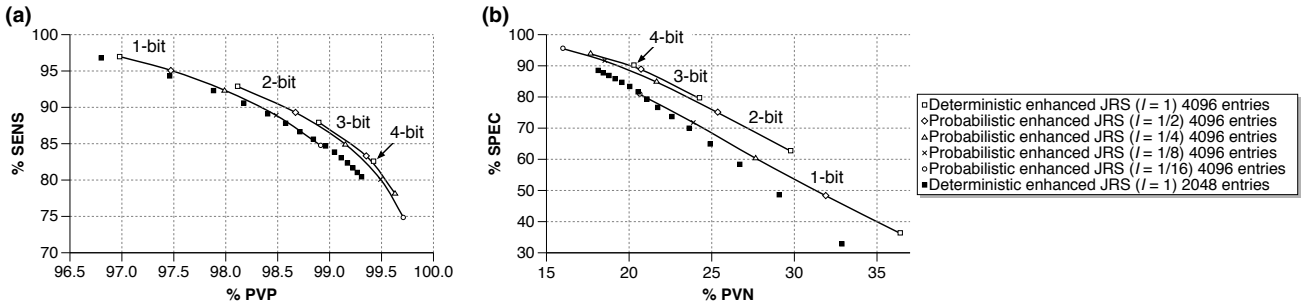


**Figure 4. Probabilistic updates endow smaller predictors with performance equivalent to that of larger deterministic predictors and decouples threshold from size.** Curves connect predictors using approximately the same area.

tradeoffs important to a class of applications including pipeline gating. Predictor behavior is similar to the PVP versus SENS data, with the relative positions of the various counter sizes reversed.

## 4. Critical path prediction

Our second set of experiments was performed in the context of Fields *et al.*'s critical path predictor [4]. In this model, a token-passing criticality analyzer updates biased 6-bit counters. The binary (*critical*, *non-critical*) output of the predictor directs optimizations. Because a relatively small number of instructions are indeed critical, predicting a few non-critical instructions as critical does not greatly affect performance, but missed critical instructions represent a significant loss of optimization opportunity. The counters are therefore biased to predict an instruction as critical if it has recently been measured as critical at least $\frac{1}{9}$ of the time.

Optimizing a predicted critical instruction might make it non-critical in the optimized dynamic instance. Nevertheless, it is desirable to continue predicting the instruction as critical, because if it is not optimized, it will remain truly critical, and the optimization can expose a secondary critical path which can also be optimized. As a result, Fields's criticality predictor is designed with a significant amount of hysteresis to ensure these instructions continue to be predicted critical. We experimented with reducing the size and/or bias of these deterministically updated counters and observed reductions in prediction accuracy that justify Fields's selection of counter parameters. For comparison, we include the best deterministic 3-bit predictor we found, $\langle 3, 4, 1, 4\rangle$.

The following experiments evaluate 3-, 2- and 1-bit probabilistic counters which represent linear scalings of Fields's 6-bit deterministic counter: $\langle 3, 1, \frac{1}{8}, 1\rangle$, $\langle 2, \frac{1}{2}, \frac{1}{16}, 0\rangle$, and $\langle 1, \frac{1}{4}, \frac{1}{32}, 0\rangle$[1]. Fractional values of $T$ were rounded down to 0, as this provided better performance. The 1-bit probabilistic counter is not suggested as a practical alternative—its behavior exhibits the graceful degradation of our method.

We describe two sets of experiments performed using critical path predictors. The first demonstrates that our proposed probabilistically-updating counter designs have the same qualitative behavior as deterministically updated counters that are two to three times larger. The second demonstrates that this behavior translates into equivalent performance for criticality-based scheduling in a clustered microarchitecture.

**Experimental method:** To perform experiments similar to those of Fields *et al.*, we simulated an 8-wide dynamically-scheduled superscalar processor with a 128-entry instruction window, 13-stage pipeline, and a *gshare* branch predictor using 16 bits of global history. We assume a perfect instruction cache, a 32 KB, 4-way L1 data cache with 2-cycle access latency, a 1 MB, 8-way L2 data cache with 10-cycle access latency, and memory with a 100-cycle latency and 4-cycle interconnect occupancy for 32-byte blocks. Simulated 16-bit LFSRs are initialized at the start of each experiment; one shift occurs per value read from the register, not per CPU cycle.

The SPEC CINT2000 benchmarks were compiled using the Digital Alpha compiler with full traditional optimization (no profile feedback). Speedups are calculated by averaging three 10 million instruction runs of the benchmarks after skipping 3, 5, and 8 billion instructions, after warming up predictors and caches for one million instructions.

### 4.1. Predictor behavior

To evaluate the behavior of alternative counter designs in isolation, we examined simulation-generated traces of Fields's model-based criticality results grouped by static instruction. We demonstrate the prediction accuracy of probabilistic counters in the context of an alias-free predictor with immediate updates—a best-case scenario for the predictors.

The accuracy of each predictor is plotted in Fig. 5. As accurate identification of critical instructions is the greater performance contributor, the reader should focus on the bottom and top segments of each column, labeled "correctly

[1] As part of this work, we explored a variety of counter parameters. Among them, we found a 2-bit configuration $\langle 2, 1, \frac{1}{16}, 0\rangle$ whose average performance slightly exceeds that of the baseline 6-bit predictor.

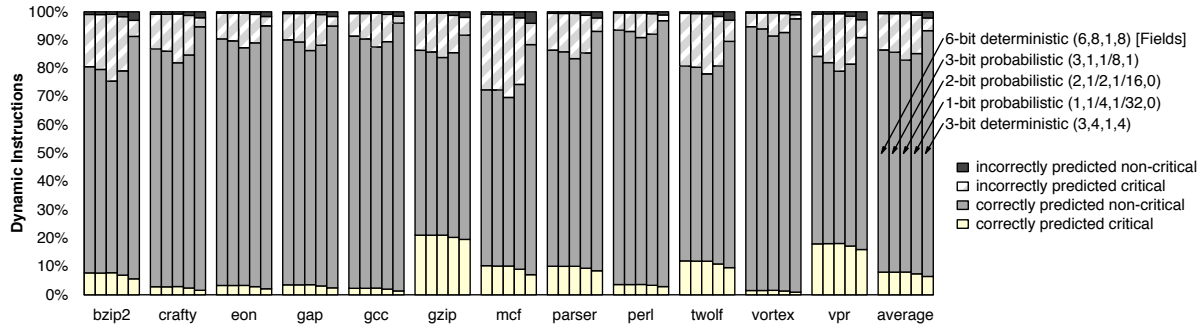


**Figure 5. The behavior of probabilistic predictors qualitiatively resembles that of larger deterministic ones.** Predictions from Fields's deterministic, three LFSR-based probabilistic, and a 3-bit deterministic counter trained on model-based criticality results, one static instruction at a time, using traces of 19 million dynamic instruction executions per benchmark.

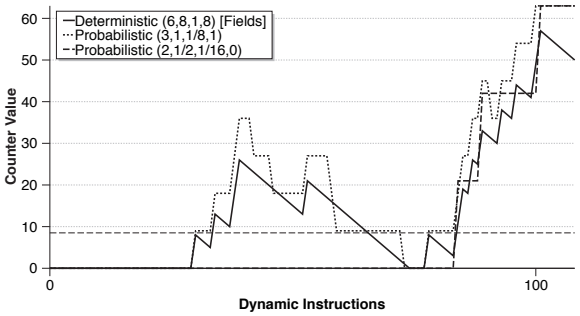


**Figure 6. Fields's deterministic and two approximating probabilistic counters, predicting model-based criticality of a single static instruction over a partial execution of *crafty*.** The dashed line indicates the threshold of Fields's counter; probabilistic counter values are scaled such that their thresholds coincide.

predicted critical" and "incorrectly predicted non-critical". There is no visibly discernible difference among the 6-bit deterministic, 3-bit and 2-bit probabilistic predictor results for these segments. The 1-bit predictor is noticeably less accurate, though its behavior is quite similar.

All the predictors can trivially classify instructions which exhibit consistent behavior over the duration of program execution—those which are always critical or never critical. More interesting cases, where instructions are occasionally critical or go through phases of criticality, distinguish the predictors' performance.

Fig. 6 illustrates probabilistic counters approximating Fields's deterministic counter design in such a case. The scales of the counters have been normalized to that of the baseline counter, from 0 to 63 ($2^6 - 1$). Plateaus indicate periods when criticality feedback is being ignored by probabilistic counters.

As the 3-bit counter is deterministically incremented, its upward movements match those of the 6-bit counter. It has proportionally fewer intermediate steps than the 6-bit counter and so must periodically—one-eighth of the time—make correspondingly larger jumps downward.

To demonstrate differing probabilistic and deterministic counter behavior, the trace in Fig. 6 was selected such that the 2-bit counter—which, unlike the 3-bit counter, has a fractional value for $I$—experiences bad luck. It effectively receives "tails" on flips of five coins in a row, completely missing the initial critical behavior of the instruction. These occurrences are rare, as demonstrated by Fig. 5, but lead to a slight reduction in accuracy relative to the 6-bit predictor.

## 4.2. Performance

We measure the performance impact of using a reduced-size probabilistic counter on Fields's applications of criticality-based scheduling in clustered architectures. Our baseline machine includes eight 1-wide clusters, each with 16-entry instruction windows, and uses a dependence-based steering policy. The criticality predictors employ 16K-entry tables trained using Fields's criticality model.

Fig. 7 plots speedups over executions without critical scheduling. Consistent with the idealized predictor accuracy, performance degrades slowly with reduced predictor storage. On average, the 2- and 3-bit counters achieve speedups only slightly smaller than the 6-bit counter: 7.75% and 7.91% compared to 7.94%. For some benchmarks, the probabilistic counter "gets lucky" and outperforms the 6-bit deterministic predictor: see *bzip2*, *gzip* and *twolf*. Even the 1-bit counter is competitive in several benchmarks, achieving 6.4% speedup on average. Notably, the 1-bit probabilistic predictor outperformed the deterministic 3-bit predictor, which achieved an average speedup of 5.1%.

Alternatively, keeping the predictor size constant, nondeterminism can be used to increase performance. For any predictor size, a probabilistic predictor exists which outperforms Fields's 6-bit deterministic one (Fig. 8). The best probabilistic counter varies with size: the 2-bit predictor performs better at smaller sizes where conflicts from aliasing are more significant than the loss in resolution.

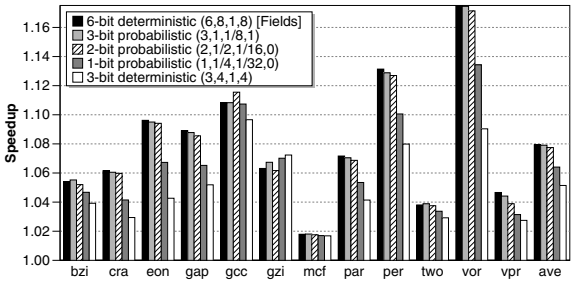


**Figure 7. The performance achieved by 3- and 2-bit probabilistic predictors closely approximates that of a 6-bit deterministic predictor.** Speedups over a clustered microarchitecture without critical scheduling, using predictions from model-based criticality results.

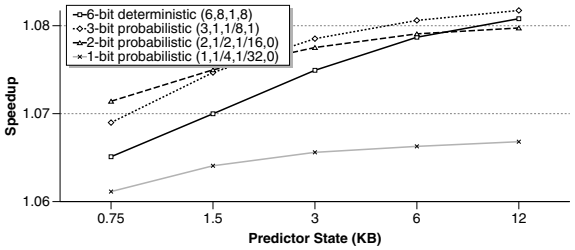


**Figure 8. Probabilistic predictors outperform deterministic ones of the same size.** Probabilistic predictors are less sensitive to size. The 3-bit deterministic predictor (not shown) always offers less than 6% speedup.

## 5. A Likelihood of Criticality tracker

The previous section demonstrated how probabilistic updates can increase the storage efficiency of a binary predictor whose outputs are (*critical*, *non-critical*). We now show that it can be useful to go beyond a binary notion of criticality to a *Likelihood of Criticality* (described in Section 5.1). We demonstrate a counter design in Section 5.2 that can stratify instructions into groups based on the frequency at which they are critical. This design includes probabilistic updates where the probabilities of incrementing and decrementing depend on the current counter state. As few as four bits of storage can thereby be used to predict Likelihood of Criticality almost as well as a predictor with unbounded precision (Section 5.3). While demonstrated in the context of criticality, the proposed counter design is equally applicable for tracking other program behaviors.

### 5.1. Likelihood of Criticality

Previous work [15] has shown, when giving preferential scheduling priority to critical instructions, it can be desirable to have more than a binary notion of criticality. With multiple levels of criticality, we can distinguish between two predicted critical instructions, and two predicted non-critical instructions. A motivating example is the fragment

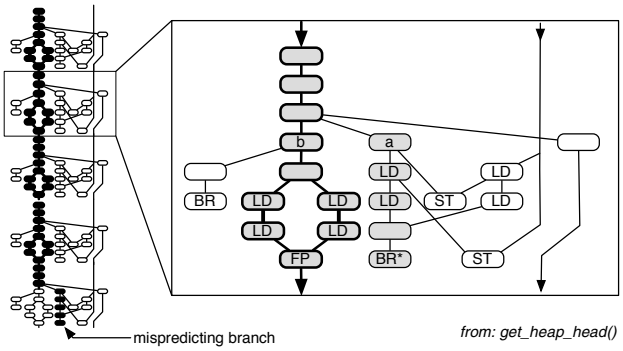


**Figure 9. A code example demonstrating the source of contention-related stalls.** The critical path, ending in a mispredicted branch (**BR***), is highlighted. Both instructions **a** (on the rib) and **b** (on the spine) are predicted critical, but instructions on the spine are more often critical.

from the benchmark *vpr* shown in Fig. 9, which exhibits a "spine and ribs" structure commonplace in programs. In this loop, the dominant spine (including instruction **b**) computes a loop-carried dependence used by ribs which periodically diverge from the spine and do not reconverge.

The rib that starts with instruction **a** includes a hard-to-predict branch. As a result, both the static instructions labeled **a** and **b** are frequently predicted as critical: they are regularly on the critical backward slice of the mispredicted branch. If we need to make a choice between these instructions, as we have to do on a clustered microarchitecture with single-issue clusters—where only one of the two instructions can issue immediately after their predecessor, and the other has to be steered away or stalled—binary predictions of criticality are insufficient.

If we choose to break ties by preferring the older instruction, in this case instruction **a**, we make the wrong choice for every iteration but the last, because only in the last iteration is **a** critical. If we instead predict *how likely an instruction is to be critical*—**b** is much more likely to be critical than **a**—and prioritize instructions based on their likelihood to be critical, we can achieve a better schedule than with a prediction of critical/not-critical. In practice, an instruction's *Likelihood of Criticality* (LoC) is well predicted by the fraction of occurrences—a real number between 0 and 1—that an instruction has been critical in the past.

While each instance where LoC improves performance only saves a few cycles, such instances are common enough to significantly impact overall performance. In the 8-wide single-issue clustered machine described in Section 4, a LoC-based scheduler offers a 5% speedup over binary criticality scheduling, as shown in Fig. 10. These results include an "unlimited precision" or "infinite" LoC tracker which records the number of critical and non-critical instances of a given instruction. When a prediction is needed, it computes the fraction of critical instances and assigns the instruction

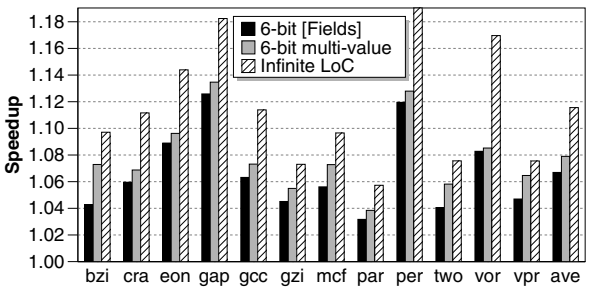


**Figure 10. True Likelihood of Criticality (LoC) information enables effective resource allocation.**

one of eight priority levels for scheduling (rather than criticality scheduling's two [4]).

Empirically, we find little benefit in providing more than eight priority levels. With eight levels, we can correctly prioritize instructions unless they have very similar LoCs, in which case the penalty of incorrectly prioritizing them—which only happens half the time—is negligible. As a result, the results in the remainder of this section use eight priority levels.

## 5.2. A LoC estimator

Clearly, tracking LoC by keeping a count of critical and total instances is not feasible, as it requires too many bits of state per instruction. In this section, we describe a feasible approach to estimating LoC, but first demonstrate that merely reinterpreting the existing counter states of Fields's predictor poorly approximates a true LoC tracker.

Fig. 10 shows that interpreting a Fields multi-bit criticality predictor's value to create a continuum of criticality, encoding eight criticality levels in the top three bits, provides only a 1% average speedup over binary criticality. This traditional predictor organization does not improve performance much because it tends to saturate at the ends of the spectrum. Fig. 11 shows a Fields predictor's behavior when fed random bit streams with a variable fraction of 1's, from 0 to 100% on the $x$ axis. For each stream, we plot the fraction of time spent in each of the eight priority levels. While the Fields predictor is effective at distinguishing between instructions with LoC above and below 11%, as the predictor increments by 8 and decrements by 1 ($\frac{1}{8+1} \approx 11\%$), it is really only able to distinguish whether an instruction is above or below that threshold.

The key to distinguishing multiple thresholds throughout the LoC continuum is a collection of increment and decrement ratios, each of which distinguishes one stratum from the next. We now describe how this can be implemented. The first step is to associate each counter encoding with a LoC. In demonstrating this idea, we consider a 3-bit counter and initially distribute the encodings equally throughout the range of LoCs (Table 2). More bits provide better resolution. The goal is for our predictor to be in a state that closely approximates the LoC of the instruction being tracked (*i.e.*, an instruction that is critical 40% of the time should have a counter value of 2 or 3 most of the time).

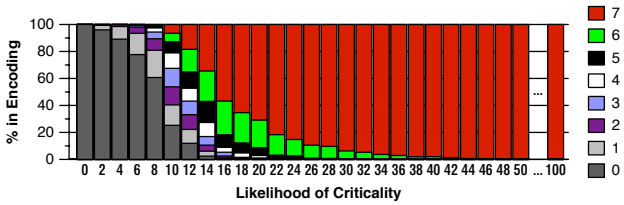


**Figure 11. Traditional saturating counters effectively encode a binary threshold.** The Fields predictor only uses the intermediate encodings for LoCs in a small transition region around 11%.

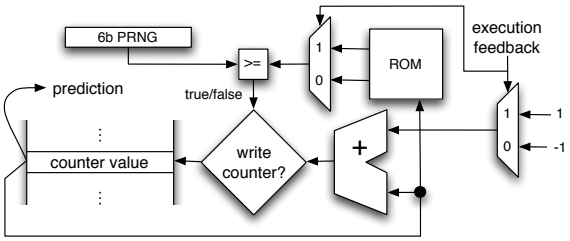


**Figure 12. A probabilistic stratifier.**

Counters track likelihoods by probabilistically incrementing (decrementing) on positive (negative) feedback with different probabilities for each value. Fig. 12 depicts a design where the current counter value indexes into a ROM, retrieving a threshold applied to the PRNG output. LoCs above the threshold are more likely to increment, and those below it are more likely to decrement.

Using different probabilities at each state lets us partition the range of likelihoods into multiple segments. Specifically, we decrease the probability of incrementing with increasing counter values (and correspondingly increase the probability of decrementing), as shown in Table 2, to create the desired operating points. These probabilities will create a tendency for an instruction to move toward an encoding that matches its LoC.

Equation 1 shows how to compute the bias direction for a given LoC at a particular counter value.

$$\text{bias direction} = \text{LoC} \times (\text{increment probability}) - (1 - \text{LoC}) \times (\text{decrement probability}) \quad (1)$$

The bias direction is positive when the LoC exceeds the counter value's operating point and negative otherwise. Thus, instructions tend to find an equilibrium point, oscillating between the operating points immediately above and below the LoC. In our example, an instruction with an LoC of 40% will tend to oscillate between counter values 2 and

**Table 2. A 3-bit LoC tracker associates a different operating point with each encoding.** Increment probabilities are (100%−operating point); decrement probabilities match the corresponding operating points.

| counter value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| operating point | 11% | 22% | 33% | 44% | 55% | 66% | 77% | 88% |
| increment probability | 89% | 78% | 67% | 56% | 45% | 34% | 23% | 0% |
| decrement probability | 0% | 22% | 33% | 44% | 55% | 66% | 77% | 88% |

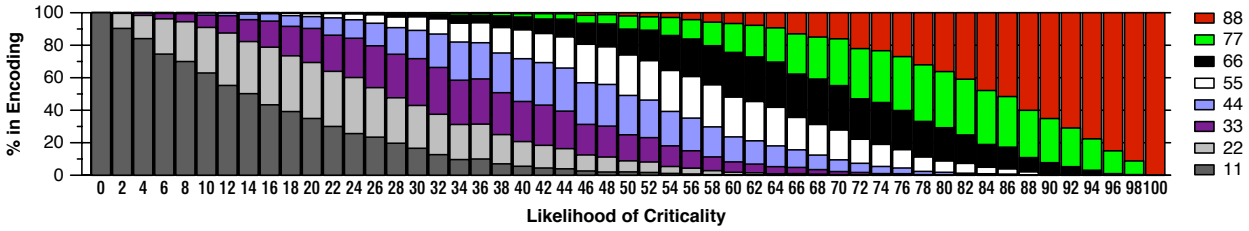


**Figure 13. The probabilistic stratifier more evenly distributes encodings.** We again use randomly generated streams with fractions of ones between 0 and 100%.

3, as shown by equations 2 and 3.

$$\textbf{when counter = 2:}$$
$$0.4 \times 0.67 - 0.6 \times 0.33 = 0.268 - 0.198 = +0.07 \quad (2)$$
$$\textbf{when counter = 3:}$$
$$0.4 \times 0.56 - 0.6 \times 0.44 = 0.224 - 0.264 = -0.04 \quad (3)$$

Using this style of counters, we can more evenly distribute encodings across the space of LoCs to construct a *probabilistic stratifier*. Figure 13 shows how long each LoC spends in each of the 3-bit counter encodings. Instructions in the middle of the LoC range spend non-trivial amounts of time in as many as six encodings. While we achieve the correct encodings on average, the statistical nature of the counters leads to some "jitter" in the value. The amount of jitter does not scale with the size of the counter, but does present a lower bound on the size of effective counters, as we show below.

Despite the jitter, the counter values can effectively distinguish two instructions of different criticality. Fig. 14 shows the probability of correctly selecting the more critical instruction as a function of the difference in criticality. For a 6-bit counter, instructions that differ in criticality by 20% can be distinguished almost 100% of the time, and even differences of only 10% can be distinguished 90% of the time. This is encouraging, as our previous experiments showed that it was only necessary to distinguish 8 levels of LoC.

We can tailor our probabilistic stratifier to the LoC domain by considering the distribution of LoC values. In the above examples, we distributed counter encodings evenly throughout the LoC space. Observed instruction LoCs are not evenly distributed, however: as shown in Fig. 15, more than half of dynamic instructions correspond to static instructions that are on average critical less than one percent of the time. We have found empirically that the best performance is achieved by distributing encodings in proportion

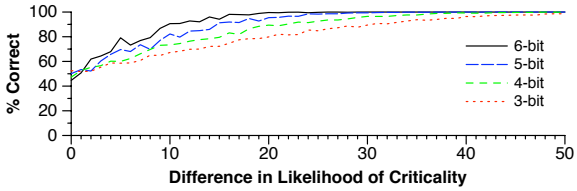


**Figure 14. Likelihood trackers can consistently distinguish instructions with differing criticality.**

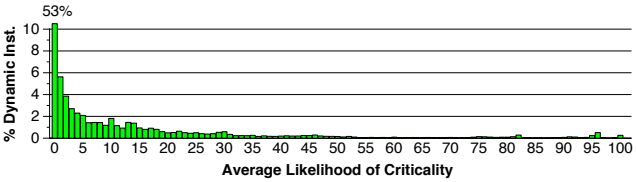


**Figure 15. Likelihoods of criticality are not equally distributed.** 53% of instructions are critical less than 1%; almost 75% are critical less than 10% of the time.

to the distribution of LoCs, *i.e.*, allocating more encodings to lower criticality values[2]. While it may seem counter-intuitive to allocate many encodings to differentiate many levels of "non-criticalness," it is important to correctly prioritize between rarely critical instructions because of the overwhelming frequency with which such comparisons are made.

## 5.3. Results

With this approach, we find that as small as a 4-bit counter, mapped down to 8 priority levels, can closely approximate the performance of the unbounded precision LoC tracker. In Fig. 16, we compare the performance of, from

[2]For example, our 4-bit counters use the following set points: 0, 0.002, 0.005, 0.01, 0.02, 0.03, 0.05, 0.07, 0.10, 0.13, 0.16, 0.21, 0.29, 0.40, 0.50, 0.60. Thus the 0000 encoding is meant to encode instructions with LoC between 0% and 0.2%.

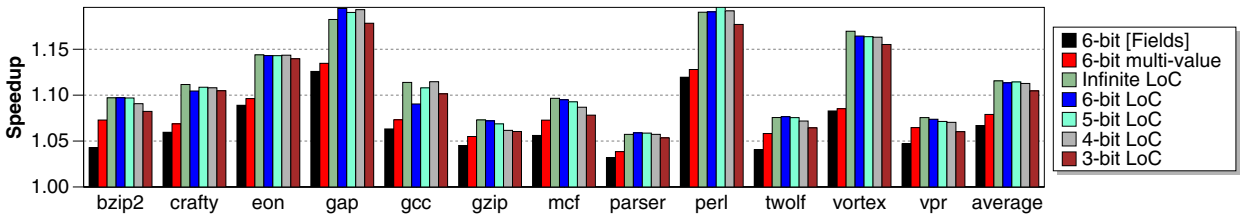


**Figure 16. A 4-bit Likelihood of Criticality (LoC) predictor approximates one of infinite size.**

left to right in each group, Fields's binary criticality predictor (using 6-bit counters), a multi-level criticality predictor implemented by using the top 3 bits of the Fields predictor, a LoC tracker with unbounded precision, and LoC trackers with 6, 5, 4, and 3 bits of precision.

The LoC trackers significantly outperform the traditional saturating counter implementations. Four bits provide sufficient precision; with fewer than four bits, there is a perceptible loss of performance as the jitter becomes non-negligible relative to the number of available encodings. Nevertheless, this probabilistic scheme, like those shown in previous sections, degrades gracefully; the 3-bit version loses only 1% of performance with respect to the ideal of unbounded precision.

Interestingly, in some benchmarks the probabilistic stratifiers occasionally outperform the infinite precision one. We do not believe this to be an important phenomenon, just "dumb luck" where elided updates happen to yield better predictions than the deteriministically updated version.

## 6. Related work

Probabilistic event counter updates and nonuniform update probabilities were previously proposed by Morris [12]. Morris's counter encoding was logarithmic, as opposed to the linear and discrete encodings we use.

Loh *et al.* observe that 2-bit counters remain in the *strongly taken* or *strongly not taken* states for 90% of the predictions made by an 8K entry *gshare* branch predictor on the SPEC CINT2000 benchmarks [8]. Instead of fractional values for $I$ and $D$, Loh *et al.* implement fractional values for $n$ by sharing a hysteresis bit among multiple counters. Reductions in area of 25% or 37.5% can be achieved with only 2.8% or 8.3% increase in mispredictions with a single hysteresis bit shared among two or four counters, respectively.

The Alpha EV8 branch predictor design [16] employed four sets of separated prediction and hysteresis arrays. In two of these hysteresis arrays, implementing the meta-predictor and part of an *e-gskew* predictor, each hysteresis bit is shared between two counters. A partial update strategy, in which the hysteresis table is not written on every prediction, somewhat alleviates the aliasing effects caused by the shared hysteresis.

The Bandwidth Adaptive Snooping Hybrid (BASH) cache coherence protocol [10] uses a saturating policy counter $\langle 8, 1, 1, T \rangle$, where $T$ is a randomly generated 8-bit integer, to determine whether to unicast or broadcast based on the bandwidth utilization. Pseudo-random numbers are generated by an LFSR and compared to the policy counter off the critical path, though the amount of logic required to implement a LFSR (as shown earlier) is negligible. RegionScout [13] is an cache coherence optimization which uses an array of LFSRs to record locally-cached regions.

A portion of this work appeared in [14].

## 7. Conclusion

We have demonstrated how probabilistic updates—implemented by introducing a pseudo-random number generator into the counter update path—can reduce hardware predictor sizes by providing probabilistic hysteresis. Branch confidence and criticality predictors' counters can be reduced to 2 bits from 4 and 6 bits, respectively, with little impact on performance. Probabilistic updates can also be used to build probabilistic stratifiers: predictors that classify an instruction based on the frequency a given behavior is observed. A 4-bit stratifier can represent eight levels of Likelihood of Criticality (LoC) with sufficient accuracy to achieve performance closely approximating that of an unbounded-precision mechanism.

While these results are significant, it is important to observe the limitations of probabilistic updates. Specifically, it should be noted that all of the above cases required at least one bit of "real" hysteresis provided by counter state to approximate the behavior of a larger counter. For branch confidence and criticality, which make binary predictions, 2-bit predictors were required; for LoC with 3-bit resolution, a 4-bit predictor was required. Below this point, with 1-bit binary predictors and 3-bit LoC stratifiers, we observed a more noticeable drop in performance. Some real hysteresis appears necessary to smooth out the randomness-introduced jitter. Demonstrating that one bit of hysteresis is both necessary and sufficient for approximating any multi-bit counter would be interesting future work.

More practically, this requirement for real hysteresis limits the applicability of probabilistic updates. It appears that probabilistic updates have little to offer counters of two or

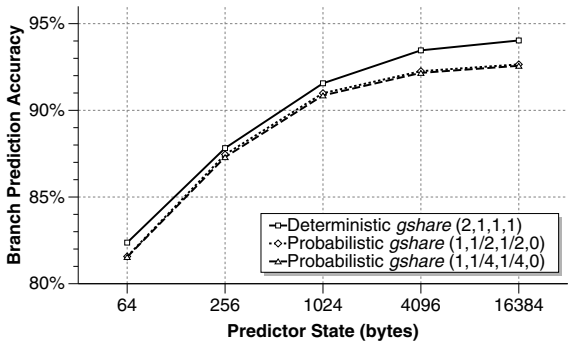


**Figure 17. Probabilistic updates are not effective for 2-bit counter-based branch predictors.**

fewer bits, including those used in most branch predictors. We confirmed this intuition with experiments comparing equal-resource *gshare* branch predictors using probabilistically updated 1-bit counters with those using deterministically updated 2-bit counters, as shown in Fig. 17. With 1-bit probabilistic counters, we can use twice as many with similar hardware cost, but the benefit of using additional counters (and increasing the history length by one) always failed to balance the penalty introduced by probabilistic updates. This is especially true in large branch predictors where additional history bits achieve diminishing returns.

Nevertheless, probabilistic updates can significantly reduce the cost—both in area and power—of other proposed predictors. By reducing the barrier to entry, probabilistic updates potentially enable inclusion of predictors whose function could not otherwise be justified. In particular, 1-bit probabilistic predictors present an interesting design point, as they are even simpler than deterministic predictors: they do not require a read-modify-write update sequence. Overall, we are optimistic that the simplicity and effectiveness of probabilistic updates will lead to their inclusion in future microprocessors.

## Acknowledgments

This research was supported in part by NSF CAREER award CCR-03047260 and a gift from the Intel corporation. We thank Pierre Salverda, Luis Ceze, Paul Sack, Naveen Neelakantam and the anonymous reviewers for their feedback on this work.

## References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[3] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proc. 26th International Symposium on Computer Architecture*, pages 64–74, 1999.

[4] B. A. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proc. 28th International Symposium on Computer Architecture*, pages 74–85, 2001.

[5] S. W. Golomb. *Shift Register Sequences*. Aegean Park Press, Laguna Hills, CA, 2nd edition, 1982.

[6] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proc. 25th International Symposium on Computer Architecture*, pages 122–131, 1998.

[7] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 142–152, 1996.

[8] G. H. Loh, D. S. Henry, and A. Krishnamurthy. Exploiting bias in the hysteresis bit of 2-bit saturating counters in branch predictors. *The Journal of Instruction-Level Parallelism*, 5, June 2003.

[9] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proc. 25th International Symposium on Computer Architecture*, pages 132–141, 1998.

[10] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. In *Proc. 8th Annual International Symposium on High-Performance Computer Architecture*, pages 251–262, 2002.

[11] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[12] R. Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[13] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proc. 32nd International Symposium on Computer Architecture*, pages 234–245, 2005.

[14] N. Riley and C. Zilles. Probabilistic counter updates for predictor hysteresis and bias. *Computer Architecture Letters*, August 2005.

[15] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *Proc. 38th Annual International Symposium on Microarchitecture*, 2005.

[16] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proc. 29th Annual International Symposium on Computer Architecture*, pages 295–306, 2002.

[17] J. E. Smith. A study of branch prediction strategies. In *Proc. 8th Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[18] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia. Store buffer design in first-level multibanked data caches. In *Proc. 32nd Annual International Symposium on Computer Architecture*, pages 469–480, 2005.