Architectural Support for Probabilistic Branches

Almutaz Adileh
Ghent University
Ghent, Belgium
almutaz.adileh@ugent.be

David J. Lilja
University of Minnesota
Minneapolis, MN, USA
lilja@umn.edu

Lieven Eeckhout
Ghent University
Ghent, Belgium
lieven.eeckhout@ugent.be

***Abstract*—A plethora of research efforts have focused on fine-tuning branch predictors to increasingly higher levels of accuracy. However, several important optimization, financial, and statistical data analysis algorithms rely on probabilistic computation. These applications draw random values from a distribution and steer control flow based on those values. Such probabilistic branches are challenging to predict because of their inherent probabilistic nature. As a result, probabilistic codes significantly suffer from branch mispredictions.**

This paper proposes *Probabilistic Branch Support (PBS)*, a hardware/software cooperative technique that leverages the observation that the outcome of probabilistic branches needs to be correct only in a statistical sense. PBS stores the outcome and the probabilistic values that lead to the outcome of the current execution to direct the next execution of the probabilistic branch, thereby completely removing the penalty for mispredicted probabilistic branches. PBS relies on marking probabilistic branches in software for hardware to exploit. Our evaluation shows that PBS improves MPKI by 45% on average (and up to 99%) and IPC by 6.7% (up to 17%) over the TAGE-SC-L predictor. PBS requires 193 bytes of hardware overhead and introduces statistically negligible algorithmic inaccuracy.

I. Introduction

Branch prediction is fundamental to modern-day superscalar processors to keep the pipeline full with useful instructions. When fetching a branch instruction, the processor predicts the outcome of the branch and speculatively fetches and executes instructions along the predicted path. When the branch turns out to be mispredicted, the processor needs to squash the wrong-path instructions and re-direct fetch along the correct path, which results in a significant performance penalty. Branch prediction accuracy is more critical for pipelines that are deeper and wider. Since the introduction of dynamic branch prediction [1], a large body of work has been devoted to improve branch prediction accuracy, see for example [2], [3], [4], [5], [6], [7], [8], [9], [10], [11].

Even though modern-day branch predictors are highly accurate, we observe that a number of branches depend on probabilistic values, i.e., the direction of the branch is determined based on a random value drawn from some distribution. These branches are inherently difficult to predict because of their probabilistic nature. We therefore call them *probabilistic branches*. We find that various algorithms in emerging application domains such as machine learning, statistical data analysis and financial optimization rely on probabilistic control flow. One typical example of a probabilistic branch appears in evolutionary optimization algorithms where

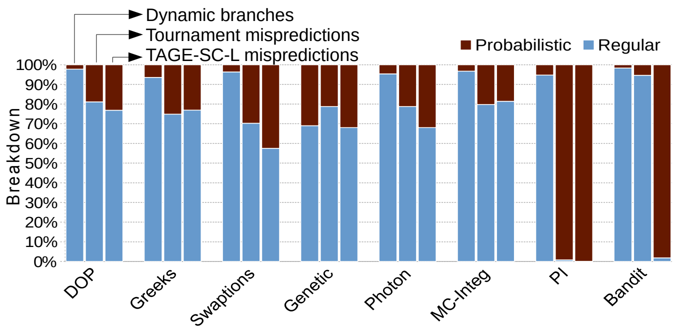

Fig. 1. Breaking down probabilistic versus regular branches: left bar shows relative execution frequency; middle and right bars show fraction of mispredictions. *In spite of their relatively small occurrence, probabilistic branches lead to a disproportionally large fraction of mispredictions.*

a random value is generated between 0 and 1, and if the value is larger than a particular threshold, a new solution is generated (e.g., through mutation or crossover) to explore the large solution space.

We find probabilistic branches to have significant impact, as shown in Figure 1. Although probabilistic branches are less frequent compared to regular (non-probabilistic) branches, see the leftmost bars, they lead to a disproportionally large number of mispredictions, see the middle and rightmost bars for a 1 KB tournament predictor [12] and 8 KB TAGE-SC-L predictor [13]. (Details about the setup are provided in Section VI.) For example, for DOP, probabilistic branches account for approximately 2% of the dynamically executed branches, however, this leads to 19% and 23% of the branch mispredictions for the tournament and TAGE-SC-L predictors, respectively. Note also that the misprediction rate for the probabilistic branches tends to be higher for the more sophisticated TAGE-SC-L predictor. This suggests that as modern predictors improve the prediction accuracy for regular branches, probabilistic branches become even more critical.

In this paper, we propose *Probabilistic Branch Support (PBS)*, a cooperative hardware/software approach to completely eliminate mispredicted probabilistic branches. PBS leverages a unique characteristic of probabilistic branches: unlike regular branches, the outcome of probabilistic branches needs to be correct only in a statistical sense. Whether the outcome of a particular execution of a probabilistic branch is taken or not is irrelevant, as long as the probabilistic characteristics of the branch (and the algorithm as a whole) are respected. This property enables relaxing what is considered a correct prediction of the probabilistic branch, i.e., any outcome of the probabilistic branch is correct, as long as we maintain the statistical properties of the branch across the entire execution of the workload.

PBS does not predict the outcome of probabilistic branches and does not speculatively fetch and execute instructions along the predicted path, as is done for regular branches. Instead, PBS stores the outcome of a particular execution of a probabilistic branch to then direct the next execution. This completely eliminates the need to predict the outcome of the probabilistic branch. Since the outcome is known at fetch time, we always fetch correct-path instructions. To guarantee correctness of execution, we need to store and retrieve the probabilistic values that are used in the control-dependent execution path. To this end, PBS stores the probabilistic values and the respective branch outcome in a special hardware table upon the first (few) execution(s) of a probabilistic branch. Subsequent executions are then steered using the stored branch decision(s). The probabilistic values generated in the code that precede the probabilistic branch are replaced with the old ones, and so is the branch direction. The probabilistic values that precede the current execution as well as the branch outcome are stored for the next execution. Because the time at which the probabilistic values and branch outcome are known may happen out of sync with the fetch stage, a careful design is needed to seamlessly integrate the proposed modifications in an out-of-order processor pipeline.

PBS involves software/hardware cooperation, i.e., the software marks probabilistic branches for the hardware to exploit. Marking probabilistic branches is done by either the compiler or the programmer. At the ISA level, we propose two new probabilistic compare and jump instructions, which can be added to the ISA or can be encoded by leveraging unused bits in existing ISAs, thereby guaranteeing backward compatibility. We augment the hardware with new tables to store probabilistic values and branch outcomes for future reference. The hardware overhead is limited to 193 bytes.

We evaluate PBS through execution-driven simulation across a set of probabilistic applications. Our experimental results report that PBS reduces the number of mispredicted branches per 1K instructions (MPKI) by 45% on average, and up to 99%, for an 8 KB TAGE-SC-L branch predictor. This leads to an average performance (IPC) improvement of 6.7%, and up to 17%, for a 4-wide out-of-order processor.

Because PBS involves a bootstrap phase in which the first few executions of a probabilistic branch are recorded to direct subsequent instances, the final result of the algorithm may deviate from the original code. This will not lead to a program crash because the probabilistic algorithms, by construction, are developed to be robust to variations in the generated random numbers. It is expected that the inaccuracy is (very) small though, because PBS only affects the first few executions of a probabilistic branch. We experimentally verify that the output is in most cases identical to the original code or falls within acceptable bounds.

II. Motivation

We first survey how several types of algorithms rely on probabilistic values in their branch conditions. We subsequently show how previously proposed techniques fail as a general solution for probabilistic branches.

A. Probabilistic Algorithms

There exist quite a few algorithms that rely on probabilistic values. These applications span numerous important domains, e.g., scientific, engineering, biological, financial and learning applications. What follows is not an exhaustive enumeration, but a coverage of applications from these important and emerging application domains.

1) Evolutionary Optimization: Evolutionary algorithms are inspired by nature in their approach to reach an optimization objective. These algorithms typically represent the problem as a stream of bits (similar to a chromosome in biological terms). The algorithms rely on a couple key functions, such as *mutate* and *crossover* in a genetic algorithm [14], to keep modifying the sequence of bits until it converges to the optimum solution. The functionality of the mutate and crossover functions relies on probabilistic values typically drawn from a uniform distribution. For example, to decide whether to mutate (or flip) a certain bit in the chromosome, a uniformly distributed random number is generated and compared against a predefined mutation rate. This probabilistic branch is not easy to predict. The code snippet below shows an example for the mutate function:

```
void mutate(string &bits){
    for (int i=0; i<bits.length(); i++)
        if (RANDOM_NUM < MUTATION_RATE)
            if (bits.at(i) == '1')
                 bits.at(i) = '0';
            else bits.at(i) = '1';
```

2) Financial Optimization: Several financial applications rely on stochastic computation, as shown in the following code example (an excerpt from the Greeks application [15]):

```
double monte_carlo_call_price(args){
  double S_adjust = S * exp(T*(r-0.5*v*v));
  double S_cur = 0.0, payoff_sum = 0.0;
  for (int i=0; i<num_sims; i++)
    double gauss_bm = gaussian_box_muller();
    S_cur=S_adjust*exp(sqrt(v*v*T)*gauss_bm);
    if (S_cur - K > 0)
      payoff_sum += (S_cur - K);
```

A random number `gauss_bm` is drawn from a Gaussian distribution using the Box-Muller transform, which gets manipulated and determines whether the probabilistic `if`-statement is taken. The probabilistic value `S_cur` that controls the probabilistic `if`-statement is derived from the random number. Note that `S_cur` is later used in the control-dependent code past the probabilistic branch to update the `payoff_sum` variable.

3) Learning Algorithms: Algorithms in the field of reinforcement learning train agents such that their strategies to reach an optimization goal converge towards the optimum. Along the way of maximizing a certain reward function, a large space of available options is being explored, balancing between an action that is known to be good versus exploring other actions that may lead to an even higher reward. An example of a well-known strategy to balance the exploration/exploitation tradeoff is the epsilon-greedy policy:

```
def epsilonGreedyAction(state: State) = {
  if (Random.float() < epsilon)
    randomAction(state)
  else
    maxQAction(state)
```

Based on a uniformly distributed random variable, it decides with an ‘epsilon’ probability to explore other solutions or to continue exploring an existing direction.

4) Physical System Modeling: The study of physical and engineering systems heavily relies on simulation models that describe the system under study. For example, stochastic approaches to simulate the propagation of light through tissues and other materials has several applications in bio-medical imaging. A beam of photons is simulated as it propagates through the substance, and probabilistic models rule the interaction of each photon with the atoms in the substance. The snippet below from [16] illustrates this.

```
double s = -log(drand48()) / sigma_t;
double distToBoundary = 0;
if (muz > 0) distToBoundary = (d - z) / muz;
else if (muz < 0) distToBoundary = -z / muz;
if (s > distToBoundary)
  // calculate values
    if (muz > 0) Tt += w; else Rd += w;
    ...
// otherwise update location
```

5) Monte Carlo Simulation: Stochastic approaches are frequently used to compute the surface of an odd shape, or the area below a curve that is hard, if at all possible, to integrate through analytical methods. These Monte Carlo approaches rely on sampling random points and checking whether these points belong to the shape or appear underneath the curve.

```
for (i=0; i<NUM_ITER; i++) {
    dx = drand48();
    dy = drand48();
    if ((dx*dx + dy*dy) < 1)
       hits++;
```

Probabilistic models, for example Bayesian networks, are frequently used to describe statistical inference problems. These models can be expressed as probabilistic programs [17] for which the outcome is typically an answer to a statistical query. For most practical cases, enumerating the whole solution space to answer the query is practically impossible. Sampling through random selection is typically employed for finding the solution.

B. Existing Solutions

Probabilistic branches pose a clear challenge for branch predictors because of their inherent randomness. Unfortunately, existing techniques such as predication and control-flow decoupling (CFD) [18] fail as a general solution for probabilistic branches, as summarized in Table I. The key take-away is that predication and CFD cannot be applied to all probabilistic codes.

1) Predication: Predication removes hard-to-predict branches through if-conversion. Instead of computing a branch condition, a predicate is computed which then guards the computation, i.e., control dependences are converted to data dependences. Predication fits well the hard-to-predict probabilistic branches and can be easily applied to small probabilistic code sequences (e.g., small `if`-statements), however, the compiler may fail to if-convert more complex code sequences. For this reason, the GNU C compiler fails to if-convert the probabilistic branches for five of the eight benchmarks considered in this study.

TABLE I
SUMMARY OF WHETHER PREDICATION AND CFD CAN BE APPLIED.
Predication and CFD cannot be applied to all probabilistic codes.

Benchmark	*Predication*	*CFD*
DOP	✓	✓
Greeks	×	✓
Swaptions	×	×
Genetic	×	✓
Photon	×	×
MC-integ	✓	✓
PI	✓	✓
Bandit	×	×

2) Control-Flow Decoupling: Control-Flow Decoupling (CFD) [18] tackles so-called separable branches within loops. A separable branch is a branch whose control-dependent code is (almost) totally independent from the code leading to the branch. CFD splits a loop that contains a separable branch into two loops: the first loop executes the code leading up to the branch and stores predicates (and possibly data values) in a queue; the second loop in each iteration pops an entry from the queue to decide whether or not to execute the control-dependent part. By doing so, CFD eliminates the mispredictions for the separable branches. CFD is not a general solution for probabilistic branches, because not all probabilistic branches are separable. In particular, `Photon` contains a hard-to-split loop-carried dependence. The probabilistic branches in `Swaptions` and `Bandit` are reached through a function call from within a loop. The compiler is unable to inline the function, and hence CFD cannot split the loop. For the benchmarks where CFD is applicable, CFD incurs overhead compared to PBS because of increased loop overhead (two or more loops instead of one big loop), and additional push and pop operations to the queue to transfer the branch outcomes (and possibly data values) from the first to the second loop.

III. PROBABILISTIC BRANCH SUPPORT

Probabilistic Branch Support (PBS) is the solution proposed in this paper to eliminate mispredictions for probabilistic branches. In this section, we first categorize probabilistic branches, and then provide a high-level description of PBS.

A. Probabilistic Branch Categories

By analyzing several codes with probabilistic branches, we classify them into two broad categories based on the usage of the probabilistic values.

Category-1: The probabilistic value is not used after the branch. In this category, a random value is generated and is used only to determine whether the probabilistic branch is taken

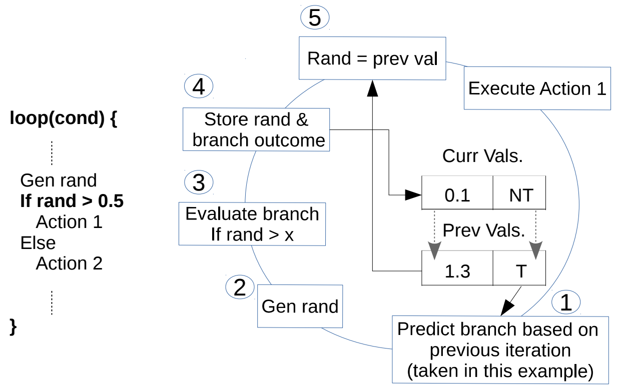

Fig. 2. Illustrating PBS operation: *The probabilistic value and branch outcome in the current iteration are stored for re-directing the next iteration.*

or not. The code following the branch is not data-dependent on the probabilistic value or any of its derivatives. Despite its simplicity, the majority of code cases we came across fall into this category. The code snippet from the genetic algorithm in Section II-A1 illustrates this.

Category-2: The probabilistic value is used after the branch. In this category, the probabilistic value is used to determine the branch outcome. The code that follows the branch is data-dependent on the probabilistic value or its derivatives. The code snippet from the financial application shown in Section II-A2 shows an example of this behavior.

B. PBS: Key Idea

The key idea of PBS is to store and retrieve probabilistic values and branch outcomes across subsequent executions of a probabilistic branch. More specifically, when fetching a probabilistic branch, we do not predict its outcome. Instead, the decision whether to take the branch or not is made based on the previous execution. As the branch instruction propagates through the pipeline, the instructions leading to the branch generate new probabilistic values. When the branch executes on a functional unit, we store its outcome (taken/not-taken) as well as the probabilistic values that may be needed after the branch. We then replace the probabilistic values with the old values from the previous execution of the probabilistic branch, which correspond with the direction followed by the branch at the fetch stage.

Figure 2 illustrates the approach taken by PBS for the loop example shown on the left. In step ❶, when the fetch unit encounters a probabilistic branch, it makes a decision based on the previous execution of the branch (‘taken’ in this example). In step ❷, the processor executes the instructions that precede the probabilistic branch. These instructions generate the probabilistic value ($rand = 0.1$) to be used by the probabilistic branch. In step ❸, the branch is executed and the branch decision is known (‘not-taken’); this branch outcome along with the probabilistic value are stored for future reference in step ❹. In step ❺, the current probabilistic value is replaced with the value that was generated in the previous iteration ($rand = 1.3$).

PBS requires an initialization phase to bootstrap the above mechanism on a processor pipeline where multiple instances of the same probabilistic branch can be outstanding at the same time. PBS therefore treats the first few executions of a probabilistic branch as a normal branch to record their outcomes and the corresponding probabilistic values. Once the first instance of a probabilistic branch has passed the execution stage, PBS then follows the recorded outcome and probabilistic values. From this point onwards, there are no more mispredictions for the probabilistic branch. Note that the first execution, or first few executions, of the probabilistic branch will follow the same direction. This pragmatic initialization mechanism does not lead to a program crash — probabilistic algorithms are designed to be robust to variations in the generated random numbers — however, it may lead to (minor) deviations in accuracy compared to the original algorithm, which we experimentally quantify in Section VII.

It is worth noting that PBS follows a deterministic approach. This is important for debugging purposes, i.e., one can fix the random seed and then deterministically replay the algorithm. In other words, PBS replays the same stream of data values when given the same initial random seed.

IV. PBS Correctness

PBS can be widely deployed. In fact, for all the benchmarks considered in this study, we were able to implement PBS as just described. We identify one necessary and sufficient condition for semantically-correct operation. **PBS can be used in algorithms where the probabilistic value gets compared to a value that does not change within a single probabilistic branch context. A probabilistic branch context is defined as the loop body that contains probabilistic branch instructions or reaches them through a function call.** Intuitively, if the comparison condition does not change across iterations, a probabilistic value that evaluates to *taken* in one iteration, evaluates to *taken* in any iteration. A context is considered the entire execution of a loop. In particular, when a loop starts with a probabilistic branch that gets compared to some value x, PBS correctness requires x to remain constant during the entire execution of the loop. Once the loop terminates, the context is considered done. Any later execution of the same loop is considered a new context. This new context is correct according to the PBS correctness rule even if the probabilistic value is now compared to a new value y ($y \neq x$) as long as y does not change during its context execution. According to this definition, a case of two disjoint loops are considered two separate contexts. Similarly, two nested loops are considered two distinct contexts. PBS can be used in both cases as long as each context in either respective case maintains the correctness rule.

The correctness condition ensures that the semantics of the algorithm are always respected, i.e., a probabilistic value that evaluates to taken leads to the execution of the taken code path, and a probabilistic value that evaluates to not-taken leads to the execution of the not-taken code path. Despite this correctness condition, we recommend programmers to be cautious as they apply PBS. In particular, the following example scenarios execute properly with PBS, but may slightly deviate from an execution without PBS support. Offline analysis is advised in such cases to assess whether this deviation is acceptable. Moreover, compilers can employ a safety mechanism to determine when it is recommended not to invoke PBS, as we will further discuss in Section V-B.

PBS starts its operation with an initialization phase during which it records few probabilistic values along with their corresponding branch decisions, as previously described in Section III. Reusing these values right after the initialization phase has a negligible impact on the code output and the probabilistic characteristics of the algorithm if the loop containing the branch executes a sufficiently large number of iterations (e.g., at least a couple thousand iterations). Our experimental results indeed demonstrate that this is the case for the probabilistic workloads considered in this study. However, if the loop consists of a relatively small number of iterations, as the following code listing shows, caution must be taken when using PBS.

```
for (int i=0; i<40; i++) {
    x = RAND;
    if (x < threshold)
        do_action_1;
    else
        do_action_2;
}
```

This particular loop consists of 40 iterations. Assuming PBS requires an initialization of 4 probabilistic values, 4 values out of 40 get used twice. We recommend performing offline analysis to determine whether using PBS significantly affects the probabilistic distribution seen by the algorithm or the outcome of the execution.

Another scenario where caution is advised includes probabilistic codes for which the value to which the probabilistic value is compared against is not constant within a context. Although this scenario goes against the correctness condition, PBS may still be applied, with care, if the value varies slowly. This is for example the case in simulated annealing, an evolutionary optimization algorithm in which a decreasing 'temperature' determines the likelihood of evaluating a radically different solution; the 'temperature' is a slowly varying parameter to which a randomly generated variable is compared. In such cases, PBS can still be applied at the cost of a (minor) deviation from the expected program behavior. Similarly, an offline analysis can help determine whether this deviation is acceptable.

It is important to mention that alternative proposals such as control-flow decoupling (CFD), discussed in Section II-B, do not face correctness issues, *if* they can be applied. CFD is a general solution for separable branches, that are not necessarily probabilistic; PBS on the other hand, specifically targets probabilistic branches. CFD splits the loop into two loops and depends on extra push and pop instructions to propagate branch decisions and any necessary data values from one split loop to the other. PBS does not incur extra instruction overhead, and can be applied to code cases that are not covered by CFD as explained in Section II-B. However, due to the initialization phase required by PBS, the order by which probabilistic values get consumed by the algorithm slightly differs. CFD does not cause such a change, leaving the semantics of the code unchanged. In other words, compared to CFD, PBS provides a new trade-off that is specifically optimized for probabilistic branches (i.e., simpler code transformations avoiding extra instruction overhead), but requiring more careful use. For both techniques, CFD and PBS, programmers and compiler designers have to identify candidate branches for optimization. To gain the benefits of PBS, programmers and compiler writers have to be cautious for cases as the ones described earlier.

V. PBS Implementation Details

We now describe the implementation details for PBS, which requires support from the ISA, software and hardware.

A. ISA Support

Software needs to mark probabilistic instructions so they get executed by our proposed PBS hardware unit rather than relying on the traditional branch predictor. To further support Category-2 probabilistic branches, software needs to provide the hardware with register names holding the probabilistic values that need to be replaced with values that suit the fetched code path.

A branch is traditionally implemented using two instructions, a compare instruction followed by a jump. To support both categories of probabilistic code sequences, both the compare and jump operations have to be extended with a probabilistic mode of operation. For that purpose, we propose two different alternatives for extending the ISA. The first is to explicitly add two new instructions to the ISA; the second is to modify the encoding of existing instructions, such that an unused bit represents the probabilistic nature of the branch. In either case, the ISA extensions enable replacing traditional branch instructions with probabilistic branches.

1) New ISA Instructions: We propose the following two instructions to implement PBS:

Probabilistic compare

`PROB_CMP optype, Prob_Reg1, Reg2`

The mnemonic `PROB_CMP` indicates a probabilistic compare instruction: `optype` is the comparison operation (e.g., less than, greater than, etc.); `Prob_Reg1` is the register containing the probabilistic value that is compared against the value in register `Reg2`. Once the comparison is done, the value in `Prob_Reg1` is stored in a specialized hardware unit for future reference (details provided in Section V-C). The execution unit replaces the value in this register with the probabilistic value that matches the direction that was already chosen at the fetch stage.

Probabilistic jump

`PROB_JMP Prob_Reg2, Immediate`

The mnemonic `PROB_JMP` indicates a probabilistic jump instruction; `Prob_Reg2` is the register containing an optional probabilistic value. When the instruction gets executed on a functional unit, the direction of the branch along with the probabilistic value in `Prob_Reg2` are stored by the PBS

```
Source Code     Traditional ISA    PBS
x = rand (0,1)  Rx = rand (0,1)    Rx = rand (0,1)
y = 5 + x;      add Ry,Rx,5        add Ry,Rx,5
if (x < 0.5)    mov R1,0.5         mov R1,0.5
  x = x*2;      cmp Rx,R1          PROB_CMP ge,Rx,R1
r = x * y;      jge dest           PROB_JMP Ry,dest
                mul Rx,Rx,2        mul Rx,Rx,2
                dest: mul Rr,Rx,Ry dest: mul Rr,Rx,Ry
```

Fig. 3. Code example illustrating PBS instructions. *The compare and jump instructions in the traditional code get replaced by probabilistic counterparts, and the registers holding probabilistic values are communicated to hardware.*

hardware. Instead of redirecting the fetch stage, the hardware unit replaces the value in `Prob_Reg2` with a value previously stored in the specialized hardware unit from the previous execution of the probabilistic branch; this value suits the branch direction and execution path taken by the fetch stage. Finally, `Immediate` is provided for target address calculation. If no probabilistic value needs replacement — this is the case for Category-1 branches — the `Prob_Reg2` register field is set to zero. If more than two values need replacement, additional `PROB_JMP` instructions are inserted, with `Immediate` set to zero for all but the last `PROB_JMP` instructions.

2) Leveraging Existing ISA Instructions: In contrast to introducing new ISA instructions, leveraging unused bits in the ISA allows flexible design decisions without losing backward compatibility. Legacy software will not have a problem running on new machines, and similarly, machines that lack PBS support can still execute software that contains probabilistic branches by treating probabilistic branches as normal branches.

To provide a general format for encoding instructions with similar characteristics, the encoding reserves specific fields in the instruction for clearly defined purposes that apply to all instructions in the particular category. However, this generalization results in fields that are used for specific instructions only, and are not being used for others. For example, a combination of comparison (e.g., `SLT`) and a branch in the MIPS ISA [19] provides the desired requirements for our ISA support for probabilistic branches. The `SLT` instruction falls into the R-class of instructions which has the `shamt` field unused except for the shift instructions. Similarly, most of the branch categories fall into the I-class of instructions which has the second register field unused (used for arithmetic/logic operations), and could therefore be used to mark our probabilistic instructions.

3) Code Example: Figure 3 illustrates a PBS code example with two probabilistic values, `x` and `y`. If the predictor decides to jump while the value in `Rx` is less than 0.5, the traditional code would require flushing the pipeline. PBS on the other hand takes the direction as previously recorded and replaces the values in `Rx` and `Ry` with values that match the taken code path (obtained from the previous execution of the probabilistic branch), while storing the newly generated random values for future use.

Note that because of out-of-order execution, instructions that follow the `PROB_JMP` instructions in program order may be executed before it. PBS guarantees that any instruction that reads a probabilistic value after `PROB_JMP` in the instruction stream gets the proper value after replacement. Both `PROB_CMP` and `PROB_JMP` specify probabilistic registers as destination registers to preserve the read-after-write dependency.

B. Software Support

The minimum software support for probabilistic branches is to simply mark them. There are several options to do so. The first option relies on the compiler to automatically decide when it is appropriate to use probabilistic instructions. The idea is to let the compiler track the location(s) in the code where random numbers are generated. By tracing the instructions that depend on the random value, the compiler checks whether any of the probabilistic derivatives control a branch instruction, and, if appropriate, encode the instruction accordingly as a probabilistic branch.

Other approaches require the programmer to pass the knowledge to the compiler. One way is to mark control flow statements as candidate cases for probabilistic branches. For example, in C/C++ this could be done using `#pragma` preprocessor directives. Alternatively, the programmer or library developer can create optimized custom libraries and invoke them in places of probabilistic control flow statements. In this paper, we manually convert traditional branches to probabilistic branches whenever appropriate.

The compiler could potentially implement further optimizations in support of PBS. The compiler could for example help to enforce correctness. After identifying candidate probabilistic branches, a compiler can provide a first safety net against inappropriate contexts for PBS. In particular, the compiler could determine through static analysis whether any of the identified probabilistic branches indeed compares against a constant value within a single context as described in Section IV.

In addition, the compiler may decide to not use PBS in case the number of probabilistic values that need replacement for Category-2 cases is too high, or when it cannot insert all the instructions that are necessary to perform the replacement. Heuristics can be used to determine whether the overhead of replacing all the values dwarfs the benefit of PBS.

C. Hardware Support

The hardware to support probabilistic branches requires extensions to the fetch and execute stages in an out-of-order pipeline, as shown in Figure 4. To support Category-1 branches, only the white areas are required. The shaded areas indicate the extensions needed for Category-2 branches.

PBS includes a new structure that resembles the branch target buffer (BTB), called the `Prob-BTB`, shown on the left in Figure 4. At the fetch stage, the `Prob-BTB` is indexed by the program counter (PC) of the `PROB_JMP` instruction. A tag match indicates that a probabilistic branch has been fetched, and the `target` address field points to the location to fetch from in the next cycle if the branch is taken. Unlike regular branches, the branch predictor is not probed to decide whether to take the branch; the `T/NT` field makes this decision for probabilistic branches.

When fetching a probabilistic jump instruction, PBS simply follows the direction stored in the `T/NT` field, i.e., the fetch

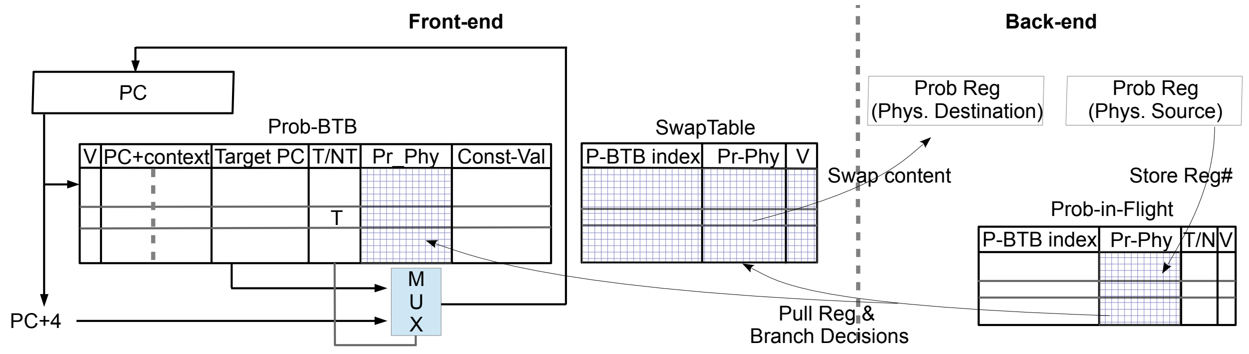

Fig. 4. Hardware support for PBS. *The* `Prob-BTB` *directs the execution of the next probabilistic branch, and along with the* `SwapTable`*, keeps track of the physical registers holding the corresponding probabilistic values. The* `Prob-in-Flight` *table keeps track of in-flight probabilistic branches.*

unit fetches either the following instruction or the one at the `target` address in the next cycle. This also triggers to pull in a recorded branch direction from the `Prob-in-Flight` table into the `Prob-BTB`. The `Prob-in-Flight` table holds the branch outcomes of in-flight instances of the probabilistic branch — we will discuss support for multiple in-flight probabilistic branches later.

The `Pr-Phy` field in the `Prob-BTB` contains a pointer to the (physical) register that holds a probabilistic value that corresponds to the direction taken by the `PROB_JMP` instruction, or the last `PROB_JMP` instruction in case there are multiple. Because the `PROB_CMP` and intermediate `PROB_JMP` instructions encode registers with probabilistic values, we use an extension to the `Prob-BTB`, called the `SwapTable`, to hold pointers to those (physical) registers. Both the `PROB_CMP` and `PROB_JMP` instructions replace the currently computed probabilistic values with these old values (which correspond with the branch direction) at the execution unit.

PBS relies on replacing newly generated values with old ones to provide dependent instructions fetched after the branch with proper probabilistic values. Therefore, the architectural registers encoded in both the `PROB_CMP` and `PROB_JMP` instructions serve a dual purpose. First, the architectural register is used as a source operand that holds the new probabilistic value. This value is compared against the branch condition during the execution of the `PROB_CMP` instruction and then saved for future use. Second, the same architectural register is a destination operand that holds the proper probabilistic value once they get swapped in during the execution of the `PROB_CMP` and `PROB_JMP` instructions.

PBS leverages register renaming to transparently replace values. Register renaming is applied to all the register operands of the `PROB_CMP` and `PROB_JMP` instructions, as would happen for regular instructions. Because the architectural register holding a probabilistic value is a source operand, it reads the physical register holding the latest copy of the architectural register. Moreover, the same architectural register is allocated a free physical register during renaming because it is a destination to be written to at execution time. The dependent instructions following the branch mark this physical register as a source operand. Both physical registers are passed to the execution stage. In addition, the execution unit receives the value of the physical registers provided by the `Prob-BTB` or the `SwapTable` — these physical registers hold the probabilistic values that led to the current branch outcome.

When the probabilistic instruction reaches the execution stage, we record the branch outcome and the source physical register holding the current probabilistic value in the `Prob-in-Flight` table. At the same time, the old value passed to the execution unit gets stored in the destination physical register, and the physical register holding it is freed. Dependent instructions wait for their source operands to be ready before they get issued to their execution units. Therefore, these instructions are guaranteed to read the proper probabilistic values after the swap operation is completed.

We set a limit to the maximum number of probabilistic branches that are in-flight between the fetch and execution stages. This number is fixed at design time. We propose the `Prob-In-Flight` table to hold up to that number of entries, as shown on the right-hand side of Figure 4. This table holds the physical registers of the newly generated probabilistic values along with the corresponding outcome of the probabilistic branch. Upon the execution of a probabilistic branch, a free entry is located in the `Prob-in-Flight` table and the physical registers holding the probabilistic values as well as the branch outcome are stored. An entry is removed from the `Prob-in-Flight` table when a new instance of the probabilistic branch gets fetched, i.e., upon fetching a probabilistic jump instruction, the physical register names and branch outcome are pulled from the `Prob-in-Flight` table into the `Prob-BTB` after which the entry in the `Prob-in-Flight` table is cleared. To facilitate the data movement to the `Prob-BTB` and `SwapTable`, the index of the corresponding `Prob-BTB` entry is stored in the `Prob-in-Flight` table.

When a new probabilistic branch gets fetched, it does not have any entries in the `Prob-BTB`. Hence, the fetch unit does not recognize the instruction as a probabilistic branch. The instruction will be executed as if it were a regular branch. As the probabilistic compare and jump finish their execution on the functional unit, an entry is allocated in the `Prob-in-Flight` table. Once the entry is complete (the branch outcome is known

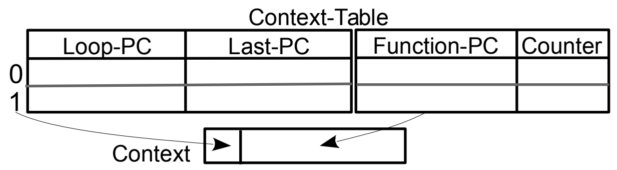

Fig. 5. Calling context support. *A context is constructed by the function call PC of the inner most loop.*

and the physical registers are recorded), it is pushed into the `Prob-BTB` and `SwapTable`. The push only succeeds if there is no entry allocated in the `Prob-BTB` for that probabilistic branch. Subsequent executions of the probabilistic branch will then detect the entry in the `Prob-BTB` for special treatment.

1) Calling Context Support: PBS ensures correctness as defined in Section IV. When executing a probabilistic branch for the first time, the value used in the comparison is registered in the `Const-Val` field in `Prob-BTB`. When executing the same branch at a following iteration, the execution unit compares `Const-Val` with the condition value of the compare instruction being executed. Non-matching values indicate a change in the condition, thus PBS may be risky to use. The entries corresponding to the branch in the probabilistic tables are flushed and the branch is treated as a regular branch.

Identifying probabilistic branches and tracking them in PBS hardware using their program counter (PC_{prob}) information alone is sufficient for most code scenarios. However, PBS tracks context information for correct execution in code cases where context matters. For example, issues may arise when the same probabilistic branch is reachable from multiple distinct paths in the code, e.g., through different function calls that are not inlined by the compiler. If the probabilistic values are treated differently depending on the calling context, PBS without context support considers (both in hardware and software) this a risky operation and treats it as a regular branch. Moreover, PBS needs to respect context start and finish boundaries to avoid conflicts resulting from two different executions of the same context (loop).

Tracking all function calls and loop contexts cannot be done in limited hardware tables. We observe that the most frequent and performance impacting branches are those executed in inner loops. Therefore, we limit our context information tracking in hardware to the two innermost loops only, and the function calls made at a depth of one inside these loops. We concatenate the PC of the function call plus a 1-bit index to the active loop with PC_{prob} and use them to index `Prob-BTB`. Thus, different paths leading to the same branch have separate entries in the probabilistic tables and do not conflict with each other. Once a loop terminates, all the entries associated with it in all the tables are cleared. This erases any possibility for incorrect behavior due to conflicts in the contexts leading to a probabilistic branch. Rebuilding history has negligible impact on PBS performance benefits, especially for frequently executed branches.

Figure 5 shows the additional information PBS needs to provide context support. PBS uses a table, called the `Context-Table`, to hold loop and function call information. The table has two entries, each tracking one loop information, for a total of two levels of nesting (i.e., two innermost). The first two fields per entry hold loop-related information, while the third and fourth fields hold function call information. A two-entry `Context-Table` suffices to achieve most of the performance benefit of PBS. Tracking a higher level of function and loop nesting is possible. However, we expect the impact of a branch on performance to be lower when executed relatively infrequently at outer loop nesting levels. Moreover, compilers that optimize for performance (vs. compactness) inline function calls whenever possible. Thus, supporting deeper function calls becomes unnecessary. The impact of a deep branch on overall performance is expected to fade as the number of function call increases. The overhead of the function calls shadows the negative impact of the branch mispredictions, leaving a lower chance for performance improvement.

We use the address of the first instruction of a loop as a representative of that loop. PBS dynamically detects a loop by tracking backward branch instructions to the beginning of the loop. When PBS encounters a backward branch whose loop address is identified for the first time, it allocates an entry for that loop in the `Context-Table`. The `Loop-PC` field stores the PC of the first instruction of the loop, while the `Last-PC` field stores the address of the backward branch instruction causing the loop. Upon encountering a backward branch to a loop that was previously detected, the `Last-PC` field of that loop is updated if the current branch instruction has a higher address. Maintaining `Loop-PC` and `Last-PC` allows proper estimation of loop boundaries, which is used to detect nested loops and the termination of a loop. A not-taken backward branch whose address is greater than or equal to the `Last-PC` indicates the termination of the loop. Our technique is inspired by a previously proposed general strategy for detecting loops with multiple levels of nesting [20].

As the processor encounters a new loop, it allocates an entry for it in the `Context-Table`. If both entries are occupied, the oldest one is removed. When a probabilistic branch is encountered during execution, it is associated with the latest loop pushed into the table. Therefore, the context information of this branch includes an index to the active loop within which the branch is reached. Because we decide to track only the two innermost levels of nesting, and thus use a table with only two entries, a single bit is sufficient to provide the loop context. Whenever a loop terminates, its corresponding entry is removed from the array, and all the branches associated with it are cleared from the probabilistic tables. The clearing process searches all the entries in the table for a matching context number, regardless of PC_{prob}, and negates their valid bit. Clearing table entries also leads to reclaiming all physical registers that hold probabilistic values, that are pointed to by the relevant table entries. If the older loop terminates before the newer one, both loops are erased.

To eliminate conflicts on a probabilistic branch reachable through multiple distinct function calls, we simply track the function call PC in the `Function-PC` field in the `Context-Table`. This field is initialized to a value of zero. When a function call is made within the loop body, the address of the instruction calling the function is recorded in this field. When a function returns, its respective field is zeroed. Upon reaching a probabilistic branch in the code, PBS uses the bit indexing the active (most recent) loop and the address of the function call made within that loop to provide the full context information required to segregate the probabilistic branch.

PBS only supports probabilistic branches reachable within a single function call from the active loop. If another function is called inside the first function, no probabilistic branches are supported (i.e., PBS treats all branches as regular ones) until all the inner functions return or another inner loop is encountered within one of the functions. `Context-Table` keeps a three-bit field, called `Counter`, to track the depth of function calls. A call to a function increments the counter of the active loop, while a return decrements that counter. PBS tracks the branches when this counter is set to zero (directly in the loop) or one (inside a function called from the loop body). Any time a loop is encountered, regardless of the function call depth, a new entry is allocated in the `Context-Table`. This newly allocated entry is similar to any randomly encountered loop and is dealt with as described earlier.

2) Hardware Cost and Scalability: The hardware cost associated with PBS is limited. To support one probabilistic branch, the hardware needs to provision for one entry in the `Prob-BTB` and a few entries in the `SwapTable` and `Prob-in-Flight` table. More specifically, to support one probabilistic branch with two probabilistic values and four in-flight copies of the branch, we need 51 bytes in the `Prob-BTB`, `SwapTable`, and `Prob-in-Flight`. These include extra context bits as explained earlier.

The context information per entry in the `Prob-BTB` is comprised of one bit for the loop information and a 48-bit PC for the function call. Additionally, there are 48 bits for the branch PC, 48 bits for the target PC, eight bits as an index to the physical register, a valid and `T/NT` bits, and a 64 bit for value comparison. An entry in the `SwapTable` consists of a 48-bit PC, three bits indexing the related entry in the `Prob-BTB`, eight bits to index a physical register, and one valid bit.

Assuming four probabilistic branches, this amounts to about 140 bytes. Each entry in the `Prob-in-Flight` table requires 2 bytes, so supporting four outstanding branches (with entries for both the compare and jump) leads to an additional hardware cost of 16 bytes. We observe one to three probabilistic branches in the applications that we studied. It is reasonable to assume that the number of probabilistic branches in an algorithm remains limited to a few branches. Assuming four probabilistic branches and accounting for the two entries in the context table each holding three 48-bit addresses and two three-bit counters, the total hardware cost amounts to 193 bytes.

Distinguishing probabilistic branches on a loop basis can significantly improve PBS' scalability. PBS relies on three hardware structures whose size is fixed at design time. In general, we expect a limited number of probabilistic branches per application. Therefore, we can provision PBS with sufficient table entries for most applications. As the processor traverses the application loops, entries occupying the probabilistic tables are promptly cleared leaving a chance to support new probabilistic branches that are scattered across the application.

TABLE II
BENCHMARKS AND THEIR CHARACTERISTICS.

Benchmark	*No. prob. branch*	*Category*	*Simulated Insns*
DOP	2/47	1	2.6 Billion
Greeks	3/50	2	2.9 Billion
Swaptions	3/309	2	17 Billion
Genetic	2/182	1	2.3 Billion
Photon	2/104	2	6.2 Billion
MC-integ	1/39	1	3.2 Billion
PI	1/45	1	1.3 Billion
Bandit	1/864	1	2.8 Billion

In cases where the number of probabilistic branches in the loop exceeds the provisioned structures, the hardware has to select which probabilistic branches to support and which to treat as a normal branch. This could be done using heuristics. For example, it may clear branches from outer loop levels first, and favor branches with a higher number of misses per instruction, or hinder branches with more than two probabilistic values to replace. We do not further evaluate these scenarios because all the applications we came across do not require replacing more than two values and/or do not have a large number of probabilistic branches to warrant such a study.

Upon a context switch, we recommend storing the 193 bytes of state information maintained by PBS and retrieving it when the context resumes. By doing so, PBS resumes its execution without incurring an additional initialization phase. In general, we expect that the number of probabilistic branches executed between two context switches to be significant enough such that even an additional initialization phase would not lead to a change in the probabilistic characteristics of the algorithm.

VI. Experimental Setup

Before evaluating PBS, we first describe our benchmarks and simulation infrastructure.

A. Benchmarks

We evaluate PBS using eight benchmarks selected from different application domains. Table II enumerates their characteristics, including the ratio of probabilistic to regular (static) branches, the category the probabilistic branches below to, and the total number of dynamically executed instructions.

Many financial applications rely on Monte Carlo simulation to carry out estimations that are challenging to obtain analytically. The digital option pricing (DOP) benchmark (taken from [21]) simulates call and put operations using probabilistic values drawn from a Gaussian distribution. This application has a single Category-1 probabilistic branch. Another interesting Monte Carlo simulation case calculates the Greeks which is a measure for the sensitivity of an option to the factors that affect it. Greeks has three Category-2 branches that depend upon each other. Swaptions from PARSEC [22] includes three Category-2 branches. We evaluate the full application run using the simsmall input set.

We use a Genetic algorithm to represent evolutionary optimization algorithms. Our benchmark is derived from the one available at [14] and includes two independent Category-1 branches. For this application, a variation in the seed of the random number generator results in a significant variation in the paths followed by the algorithm. To get reliable performance measurementss, we consider 8 different seeds over which we compute the average and 95% confidence interval.

We also consider Photon, a stochastic simulation for the transportation of light in a thin translucent slab. We use the application taken from [16], which features two independent Category-2 branches. We include two kernel benchmarks to represent stochastic computation. The first one is the famous estimation of the value PI [23]. The second one, MC-integ, integrates a curve under a predefined function similar to [24]. Both benchmarks feature a single Category-1 branch.

Finally, Bandit is a multi-armed bandit application to represent learning algorithms. The code relies on an epsilon-greedy policy to determine whether to exploit current knowledge or to explore other arms towards maximizing the sum of reward from pulling arms. The code is adopted from [25] and contains a single Category-1 branch.

We use the GNU C compiler v4.6.3 to compile the benchmarks with optimization flag `-O3`.

B. Simulation Infrastructure

We use Sniper 6.0 [26] to perform experiments in this paper. We assume an aggressive four-wide out-of-order core with a 168-entry reorder buffer configured after Intel's Sandy Bridge. The memory hierarchy consists of two levels: a split L1 cache with 32 KB for each of the I- and D-caches, and a unified 2 MB L2 cache. The branch misprediction penalty is set to 10 cycles to re-fill the front-end pipeline after the branch is executed. We consider two branch predictors: (i) a 1 KB tournament predictor modeled after the Pentium-M [12], consisting of a global branch predictor, a bimodal branch predictor and a loop branch predictor; and (ii) an 8 KB TAGE-SC-L predictor taken from the 2016 Branch Prediction Championship [13]. Experiments were conducted assuming PBS hardware support (188 bytes) for four distinct probabilistic branches, with four outstanding branches in flight, which is sufficient for our benchmarks.

VII. Evaluation

We now evaluate PBS. We quantify branch predictor performance as well as processor performance, and we evaluate how PBS reduces negative branch predictor interference. Finally, we evaluate the correctness of the output under PBS and evaluate the impact of PBS on the observed randomness of the probabilistic values.

A. Branch Predictor Performance

We use mispredictions per 1K instructions (MPKI) to quantify the improvement in branch predictor performance. Because PBS completely eliminates mispredictions for the probabilistic branches, we expect PBS to significantly reduce the number of mispredicted branches per instruction. Figure 6 reports the reduction in MPKI through PBS for the tournament and TAGE-SC-L predictors.

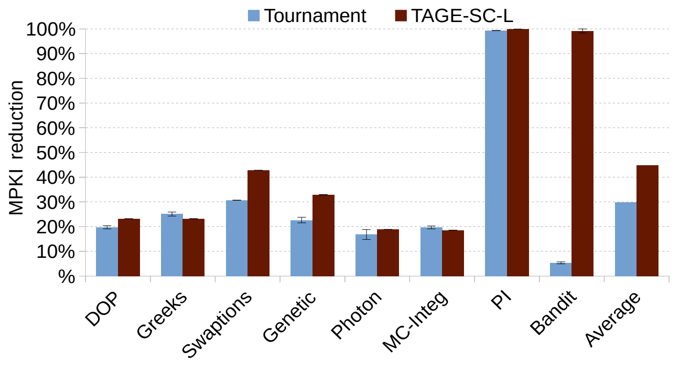

Fig. 6. MPKI reduction through PBS. *PBS significantly reduces the number of misses; higher reductions achieved for the more advanced TAGE-SC-L.*

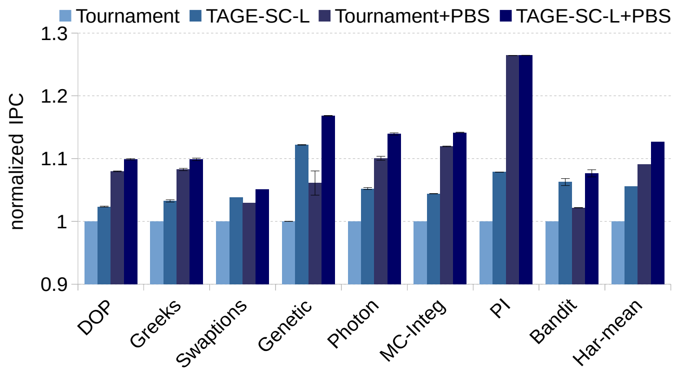

Fig. 7. Normalized IPC for a 4-wide superscalar processor. *Performance improves significantly with PBS support.*

PBS reduces MPKI by 29.9% average for the tournament predictor, and up to 99%. We achieve even higher reductions in MPKI for the TAGE-SC-L predictor, by 44.8% on average. (A reduction by 99% is achieved for the benchmarks for which the mispredictions are dominated by probabilistic branches, see Figure 1.) Because the higher accuracy for the TAGE-SC-L predictor comes from more accurately predicting regular branches, the fraction of mispredictions due to probabilistic branches increases, hence the higher reduction in MPKI through PBS for the TAGE-SC-L predictor.

B. Processor Performance

Figure 7 reports normalized performance (IPC) for the (i) baseline tournament predictor, (ii) TAGE-SC-L, (iii) tournament predictor with PBS, and (iv) TAGE-SC-L with PBS. PBS yields a significant performance improvement. In particular, performance improves by 9% on average (up to 26%) and by 6.7% on average (up to 17%) when augmenting the tournament and TAGE-SC-L predictors with PBS, respectively. It is interesting to note that the tournament branch predictor with PBS outperforms the TAGE-SC-L predictor. This is a significant return on investment given the small hardware cost for implementing PBS.

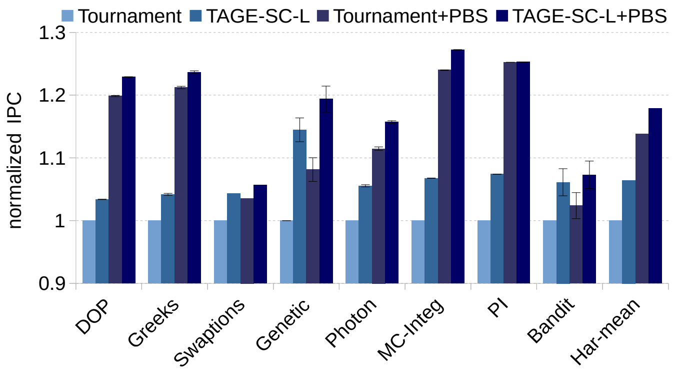

Fig. 8. Normalized IPC for an 8-wide superscalar processor. *Even higher performance improvements are achieved for a wider processor pipeline.*

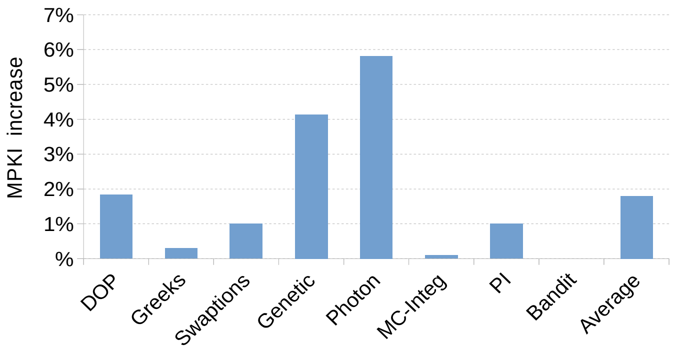

Fig. 9. Increase in MPKI for the tournament predictor due to probabilistic branches incurring negative interference with regular branches. *Eliminating probabilistic branches from accessing the branch predictor leads to less negative interference.*

Even higher improvements are obtained for a wider processor pipeline, see Figure 8 for an 8-wide superscalar processor with a 256-entry ROB. PBS improves performance by 13.8% (up to 25%) for the tournament predictor, and by 10.8% on average (up to 19%) for TAGE-SC-L predictor.

C. Branch Predictor Interference

The reduction in MPKI through PBS primarily comes from eliminating the mispredictions due to probabilistic branches. However, there is also a secondary effect that comes into play. Because we do not access the branch predictor for probabilistic branches, PBS may lead to less negative interference in the branch predictor. We set up the following experiment to quantify this effect: we first run a simulation in which we let all (regular *and* probabilistic) branches access and update the branch predictor; in our second simulation, we only access and update the branch predictor for the regular branches, i.e., we filter out the probabilistic branches. The delta in misprediction rate then is a measure for the negative interference due to probabilistic branches.

Figure 9 reports the increase in MPKI because of negative interference due to probabilistic branches for the tournament branch predictor. (We report the maximum increase in MPKI across 7 experiments with different random seeds per benchmark.) The impact is evident, reaching up to 5.8% and a

TABLE III
RANDOMNESS TEST RESULTS: 95% CONFIDENCE INTERVALS ACROSS 114 DIEHARDER TEST CASES IN THE ORIGINAL CODE VERSUS AFTER PBS. *PBS does not significantly affect the randomness of the random number sequence.*

	Original			*PBS*		
	PASS	WEAK	FAIL	PASS	WEAK	FAIL
Swaptions	48-40	11-6	64-58	50-40	12-7	63-57
Genetic	16-13	4-1	98-96	15-13	5-2	97-96
Photon	42-32	27-13	64-50	57-31	29-8	58-45
MC-integ	27-24	2-0	88-87	27-25	1-0	88-87
PI	27-24	3-0	88-87	26-24	2-1	88-87
Bandit	25-18	25-18	76-68	26-20	27-19	72-64

couple of percents on average. For the TAGE-SC-L predictor (not shown in the graph), we observe that negative interference is negligible, which makes intuitive sense because of its larger size and more sophisticated organization.

D. Correctness of the Output

As mentioned earlier, PBS may lead to (slight) inaccuracies compared to the original algorithms because of the initialization bootstrap phase when previously unseen probabilistic branches get executed. We now quantify these inaccuracies. Probabilistic algorithms are designed to deal with variations in the random numbers being generated. In other words, there is inherent non-determinism, and as a result, the output is approximate, not exact. Hence we need to resort to application-specific metrics to quantify the impact of PBS on accuracy, similar what is done in AxBench [27].

For DOP, Greeks, Swaptions, MC-integ and PI, we compute the relative error between the output of the original run versus the run with PBS; we use the same random seed for both runs. We observe that the error is zero for these applications under PBS.

For Genetic, we consider the success rate of the trials to find a particular the problem. The original code achieves an average success rate (successful trials divided by the total number of trials) of 0.2 and a 95% confidence interval of [0.18, 0.22]. The average success rate under PBS equals 0.206 with a 95% confidence interval of [0.18, 0.23]. Because the confidence intervals overlap, we can state that there is no statistical evidence that PBS differs from the original run.

For Photon, we compare the output images using the Average Root-Mean-Square error. The observed variation is small (3.9%) and falls within the acceptable range [28], [29]. For Bandit, we compare the reward and regret values the algorithm reaches at the end of the run. The observed error is zero under PBS.

E. Impact on Randomness

PBS alters the original sequence of random values as part of the bootstrap phase. One valid question is whether this affects the randomness and distribution of sequence of random numbers effectively perceived by the algorithm.

To verify this in more detail, we perform DieHarder randomness tests [30], using version 3.31.1. This can only be done for the benchmarks whose probabilistic branches are controlled by a value derived from a uniform distribution. (DOP and Greeks use values derived from a Gaussian variable.) For each benchmark, we run the DieHarder tests seven times, each time using a different seed to initialize the random number generator. For each seed, we create two files of random values, the first one contains the values as they are generated by the original code, and the second one contains the values in the order as they get processed under PBS. DieHarder includes 114 test cases for which the outcome could be PASS, WEAK or FAIL. For each application, we report the 95% confidence interval for the possible outcomes of the tests.

As shown in Table III, we find that the random numbers generated directly by the original code pass for several tests and fail for others. For most results, the 95% confidence intervals achieved by PBS and the original code overlap significantly. In general, the failed tests are consistently similar for most of the benchmarks. PBS even shows a slight potential for improvement in some cases, such as Swaptions, Bandit and Photon. Similarly, the WEAK and PASS results of PBS and the original code significantly overlap for almost all benchmarks. The classification for a few tests may become WEAK instead of PASS or vice versa. For example, in Genetic the intervals are almost identical, but there is a possibility that the number of tests classified as WEAK increase by one and the number of test classified as PASS decrease by one. The same increase in the number of WEAK tests may also be attributed to the decrease in FAIL tests. For all the other benchmarks, including PI, MC-integ, Bandit and Photon, the classification suggests a potential improvement by PBS. In general, the results of PBS and the original code significantly overlap, indicating that the two techniques are statistically identical. The bottom line is that PBS does not significantly affect the randomness and distribution of the random number sequence.

VIII. Related Work

We have seen a significant body of work on branch prediction over the past few decades, see [3], [7], [8], [9], [10], [2] to name a few. Other proposals include neural branch prediction [4], [5], [6], O-GEHL [31] and TAGE [11]. Recent innovations include augmenting the main predictor with side predictors that target specific branch patterns, as in TAGE-SC-L [32], [13]. In this paper, we propose PBS to eliminate mispredictions for the specific class of probabilistic branches.

Several techniques have been proposed to reduce the branch misprediction penalty by fetching and executing along multiple control flow paths [33], [34], [35]. PBS does not require to execute along multiple paths to eliminate the branch misprediction penalty.

Several papers have proposed techniques for branch pre-execution [36], [37], [38], [39], [40]. These techniques rely on executing the instructions leading to the branch on dedicated hardware resources in an attempt to resolve the branch earlier. PBS effectively pre-executes probabilistic branches without a dedicated hardware context.

Branch Vanguard [41] decomposes the branch into prediction and resolution operations to help the compiler tailor a schedule for in-order cores. It targets highly-predictable, unbiased branches on in-order cores. PBS targets highly unpredictable probabilistic branches on out-of-order processors.

To avoid flushing instructions upon a misprediction, several techniques have been proposed to identify control and data-independent instructions [42], [43], [44]. Such techniques require significant hardware overhead to identify reconvergence points, track data dependences and re-execute instructions.

Wish branches [45] strike a sweet spot between predication and branch prediction. A wish branch resorts to predicated execution when the branch is hard to predict, avoiding the misprediction overhead. The wish branch is predicted, if it is easy to predict, thereby avoiding predication overhead. PBS does not resort to predication nor branch prediction for probabilistic branches.

Branch-on-random [46] is a new branch instruction that specifies the probability at which the branch should be taken, unlike regular branches which specify the condition upon which the branch should be taken. Branch-on-random was conceived to reduce the overhead of program instrumentation compared to a software-only solution. PBS is more generally applicable to other probabilistic codes, including Category-2 branches and generalized probabilistic distributions.

PBS also bears some similarity with approximate computing [47], [48], [49] because of the inherent tolerance to variations in execution for probabilistic codes. However, while approximate computing focuses on leveraging approximation in data streams, PBS leverages approximation in control flow.

IX. Conclusions

In this paper we observe that several important categories of applications rely on probabilistic branch behavior which is inherently difficult to predict, even for state-of-the-art branch predictors. We present Probabilistic Branch Support (PBS), a hardware/software cooperative solution in which software identifies probabilistic branches to hardware through two novel probabilistic compare and jump instructions that can be transparently implemented in the ISA. PBS records and retrieves probabilistic values and branch outcomes across subsequent executions of a probabilistic branch, thereby completely eliminating the misprediction penalty. With minimal hardware overhead of 193 bytes, PBS improves MPKI by 45% on average (up to 99%) and IPC by 6.7% on average (up to 17%) for a 4-wide superscalar processor with a state-of-the-art TAGE-SC-L branch predictor, while incurring negligible algorithmic inaccuracy.

Acknowledgements

We thank the reviewers for their valuable suggestions. This work was supported by the European Research Council (ERC) Advanced Grant agreement No. 741097, and the FWO projects G.0434.16N and G.0144.17N. This work was further supported by U.S. National Science Foundation grant no. CCF-1438286. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture (ISCA)*, 1981.

[2] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 1991.

[3] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1998.

[4] D. A. Jiménez, "Piecewise linear branch prediction," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[5] D. A. Jiménez, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[6] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[7] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1997.

[8] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[9] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1997.

[10] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[11] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction-Level Parallelism*, vol. 8, 2006.

[12] V. Uzelac and A. Milenkovic, "Experiment flows and microbenchmarks for reverse engineering of branch predictor structures," in *Proceedings of International Symposium on Performance Analysis and System Software (ISPASS)*, 2009.

[13] A. Seznec, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[14] M. Buckland, "Genetic algorithm example," 2013. [Online]. Available: http://www.codemiles.com/c-examples/genetic-algorithm-example-t7548.html

[15] M. Halls-Moore, "Calculating the Greeks with Finite Difference and Monte Carlo Methods in C++," 2013. [Online]. Available: https://www.quantstart.com/articles/Calculating-the-Greeks-with-Finite-Difference-and-Monte-Carlo-Methods-in-C

[16] Scratchapixel, "Monte Carlo methods in practice," 2015. [Online]. Available: https://www.scratchapixel.com/code.php?id=31&origin=/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice

[17] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the International Conference on Software Engineering (ICSE): Future of Software Engineering*, 2014.

[18] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

[19] J. Heinrich *et al.*, *MIPS R4000 Microprocessor User's Manual*. MIPS technologies, 1994.

[20] J. Tubella and A. Gonzalez, "Control speculation in multithreaded processors through dynamic loop detection," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1998.

[21] M. Halls-Moore, "Digital option pricing with C++ via Monte Carlo methods," 2013. [Online]. Available: https://www.quantstart.com/articles/Digital-option-pricing-with-C-via-Monte-Carlo-methods

[22] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[23] J. Wynne III, "Serial to parallel: Monte Carlo operation," 2013. [Online]. Available: https://www.olcf.ornl.gov/tutorials/monte-carlo-pi/

[24] C. P. Robert, *Monte Carlo Methods*. Wiley Online Library, 2004.

[25] J. Komiyama, "BanditLib: A simple multi-armed bandit library," 2015. [Online]. Available: https://github.com/jkomiyama/banditlib

[26] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, 2014.

[27] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, 2017.

[28] J. Park, E. Amaro, D. Mahajan, B. Thwaites, and H. Esmaeilzadeh, "AxGames: Towards crowdsourcing quality target determination in approximate computing," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[29] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[30] R. G. Brown, "DieHarder: A random number test suite," 2011. [Online]. Available: http://www.phy.duke.edu/~rgb/General/dieharder.php

[31] A. Seznec, "Analysis of the o-geometric history length branch predictor," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[32] A. Seznec, "A new case for the TAGE branch predictor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[33] T. H. Heil and J. E. Smith, "Selective dual path execution," Technical report, University of Wisconsin-Madison, Tech. Rep., 1996.

[34] A. Klauser, A. Paithankar, and D. Grunwald, "Selective eager execution on the polypath architecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[35] E. M. Riseman and C. C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. 100, 1972.

[36] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[37] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[38] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1998.

[39] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[40] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2001.

[41] D. S. McFarlin and C. Zilles, "Branch vanguard: decomposing branch functionality into prediction and resolution instructions," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[42] C.-Y. Cher and T. Vijaykumar, "Skipper: a microarchitecture for exploiting control-flow independence," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2001.

[43] Y. Chou, J. Fung, and J. P. Shen, "Reducing branch misprediction penalties via dynamic control independence detection," in *Proceedings of the International Conference on Supercomputing (ICS)*, 1999.

[44] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[45] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2005.

[46] E. Lee and C. Zilles, "Branch-on-random," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[47] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[48] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

[49] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.