Reactive Techniques for Controlling Software Speculation

Craig Zilles Naveen Neelakantam

University of Illinois at Urbana-Champaign
Siebel Center for Computer Science, 201 N. Goodwin Ave. Urbana, IL 61801
[zilles, neelakan]@uiuc.edu

Abstract

Aggressive software speculation holds significant potential, because it enables program transformations to reduce the program's critical path. Like any form of speculation, however, the key to software speculation is employing it only where it is likely to succeed. While mechanisms for controlling hardware speculation (e.g., saturating counters updated after each instance) are well understood, these techniques do not translate directly to software techniques because changing a speculation requires changing the code. As it stands, the dominant software speculation control technique, non-reactive profile-guided optimization, lacks the robustness to support aggressive speculation.

The primary thesis of this paper is that software speculation can be made to be robust by adding a reactive controller that can dynamically adjust the speculation. We make two primary observations about such systems: 1) reactive control systems can select behaviors on which to speculate with performance that equals or exceeds self-training, and 2) such control systems are remarkably latency tolerant. Although reactivity is required, it can be done at a low frequency; latencies of hundreds of thousands, or even millions of cycles, can be tolerated for most actions. Together these two characteristics imply that robust aggressive software speculation is a realistic goal.

1. Introduction

Programs are seldom creative; they tend to exhibit the same behavior over and over. This repetition has been well documented across a variety of program behaviors (*e.g.*, branches [13], memory dependences [10], and values [8]) and has made viable a broad range of speculative optimizations (*i.e.*, optimizations that improve one case at the expense of another). The key to applying such speculative optimizations is in knowing which program behaviors will be frequently exhibited and which will not.

Speculation techniques can be categorized by whether the optimization is applied before or after an instruction has been fetched, categories we will refer to as software and hardware speculation, respectively.

1. Hardware speculation typically involves tracking a program behavior using one or more hardware tables and consulting these tables in the front-end of the pipeline to decide whether (and how) to apply the optimization. If an optimization is selected, it is applied to the in-flight instructions as they proceed down the pipeline. The advantages of hardware speculation are that it is responsive to changes in program behavior and can apply the optimization to selective instances, but it requires that the optimization can be applied as the instructions travel down the pipeline.

2. In software speculation, the speculation is encoded directly into the instruction stream when the code is generated; many such mechanisms have be proposed, including EPIC/VLIW's advanced loads [9], rePLay's assertions [4], and Master/Slave Speculative Parallelization's (MSSP) approximations [20]. The two main advantages of this approach are: 1) the program can be transformed assuming the speculations are correct, and 2) the optimizations need not be applied on each dynamic instance as it travels down the pipeline. However, software speculation is not as reactive as hardware speculation, as there is often a latency associated with applying (or removing) an optimization due to the need for recompilation.

We believe these forms of speculation are complementary in nature and have distinct strengths. Specifically, software speculation (the focus of this paper) appears to be well suited for targeting highly-biased program behaviors, where such speculation enables program transformations that reduce the critical path or improve parallelism. We demonstrate that there is substantial opportunity for such optimizations in Section 2. The key challenge to exploiting aggressive software speculation lies in effectively controlling the speculation.

By and large, the dominant technique for controlling software speculation has been profile-directed optimization, operated in an open loop (*i.e.*, no feedback control on misspeculation rate). Because of the repetitiveness of programs, a profile of the program (either from a previous run or a portion of a current run) can be used, with reasonable accuracy, to predict the program's future behavior. In most systems (both static compilers and dynamic translators), code is optimized after a single profiling phase, with no further monitoring of program behavior. Where run-time behavior differs from the profile, a misspeculation cost is incurred. We demonstrate, in Section 2.2, that these techniques lack robustness and miss opportunity for correct speculation.

In this paper, we demonstrate that both the effectiveness and robustness of software speculation can be improved by adding a reactive system to control the speculation. In Section 3, we describe a simple, yet effective, model for controlling speculation that can consistently achieve misspeculation rates of less than one-half of a percent. Such misspeculation rates are conducive to aggressive speculation because misspeculation penalties that are two orders of magnitude greater than the benefit of correct speculation can be tolerated. In fact, the results achieved by our reactive system are comparable or even exceed those achievable by self-training (*i.e.*, profiling and evaluating using the same data set) in a static, non-reactive system.

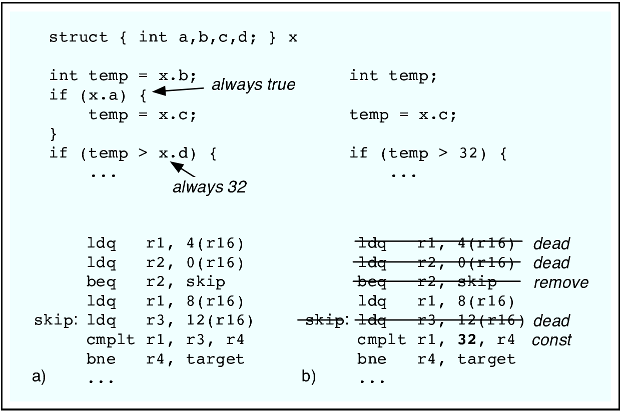
As the utility of a reactive control system *can* be demonstrated in architecture-independent manner, we have endeavored to do so. In this way, our results can be interpreted in other contexts where the benefit and costs of speculation are different. Nevertheless, it is important to demonstrate that the presence or absence of reactive control can have a first-order impact on performance. In the final section of this paper, we provide timing simulations of an MSSP machine that validate the observations from our abstract model and show that reactiveness in the speculation control policy can make the difference between speedups and slowdowns (Section 4).

In summary, we make three primary contributions:

1. In the context of branches, we perform a characterization of highly-biased program behaviors, including: i) estimating the optimization opportunity, ii) exploring the relationship between initial and overall behavior, and iii) providing examples and explanations of instances with time-varying behavior.
2. We present a simple reactive control policy that exploits most of the benefit of highly-biased branches.
3. We demonstrate that the model is insensitive to many of its parameters, most notably latency, making it conducive to implementation.

2. Motivation

This work was motivated by our work to develop a dynamic optimizer for the MSSP execution paradigm (a brief overview of MSSP is included in Section 4). MSSP provides an external verification mechanism that allows its master thread to execute unchecked speculative code. That is, speculative optimizations (based on recurring program behaviors) can be applied to the code without the need for checking or fixup code. As a result, MSSP code typically has a shorter critical path and is smaller (both statically and dynamically) than corresponding traditionally compiled code (with or without checked speculative optimizations (e.g., IA-64). An illustrative example is provided in Figure 1.



```
struct { int a,b,c,d; } x

int temp = x.b;    always true           int temp;
if (x.a) {  <-
    temp = x.c;                          temp = x.c;
}
if (temp > x.d) {                        if (temp > 32) {
    ...     always 32                        ...


      ldq   r1, 4(r16)                 ~~ldq   r1, 4(r16)~~   dead
      ldq   r2, 0(r16)                 ~~ldq   r2, 0(r16)~~   dead
      beq   r2, skip                   ~~beq   r2, skip~~     remove
      ldq   r1, 8(r16)                 ldq   r1, 8(r16)
skip: ldq   r3, 12(r16)         ~~skip~~: ~~ldq   r3, 12(r16)~~  dead
      cmplt r1, r3, r4                 cmplt r1, 32, r4     const
      bne   r4, target                 bne   r4, target
a)    ...                      b)      ...
```

Figure 1. An illustrative MSSP code approximation example: before (a) and after (b) approximation. *Profiles indicate the first* `if` *statement is highly biased to be true and the values of x.d are frequently 32. MSSP allows generation of speculative code (without checks) assuming these behaviors, leading to a significant simplification of the code.*

When its speculations prove correct, MSSP achieves the performance of the speculative code, but when a misspeculation occurs it often takes hundreds of cycles to be detected. Thus, for each speculation there is a modest benefit for being correct and a large penalty for being incorrect, necessitating a very low misspeculation rate (*e.g.*, less than 1 percent). This requirement holds for aggressive software speculation in other contexts (*e.g.*, rePLay [4] and thread-level speculation [18]).

This section serves to demonstrate that there is significant opportunity for aggressive software speculation and motivate the need for reactive speculation control systems (as described in Section 3) to harvest this opportunity. Due to space constraints, we have chosen to concentrate on one program behavior–conditional branches–throughout this paper. This choice both provides some context for the results we present (as many readers will be familiar with the branch behaviors of the SPEC2000 integer benchmarks) and is important because branches remain one of the most important constraints on the optimization of non-numeric programs. We have confirmed that these results are qualita-

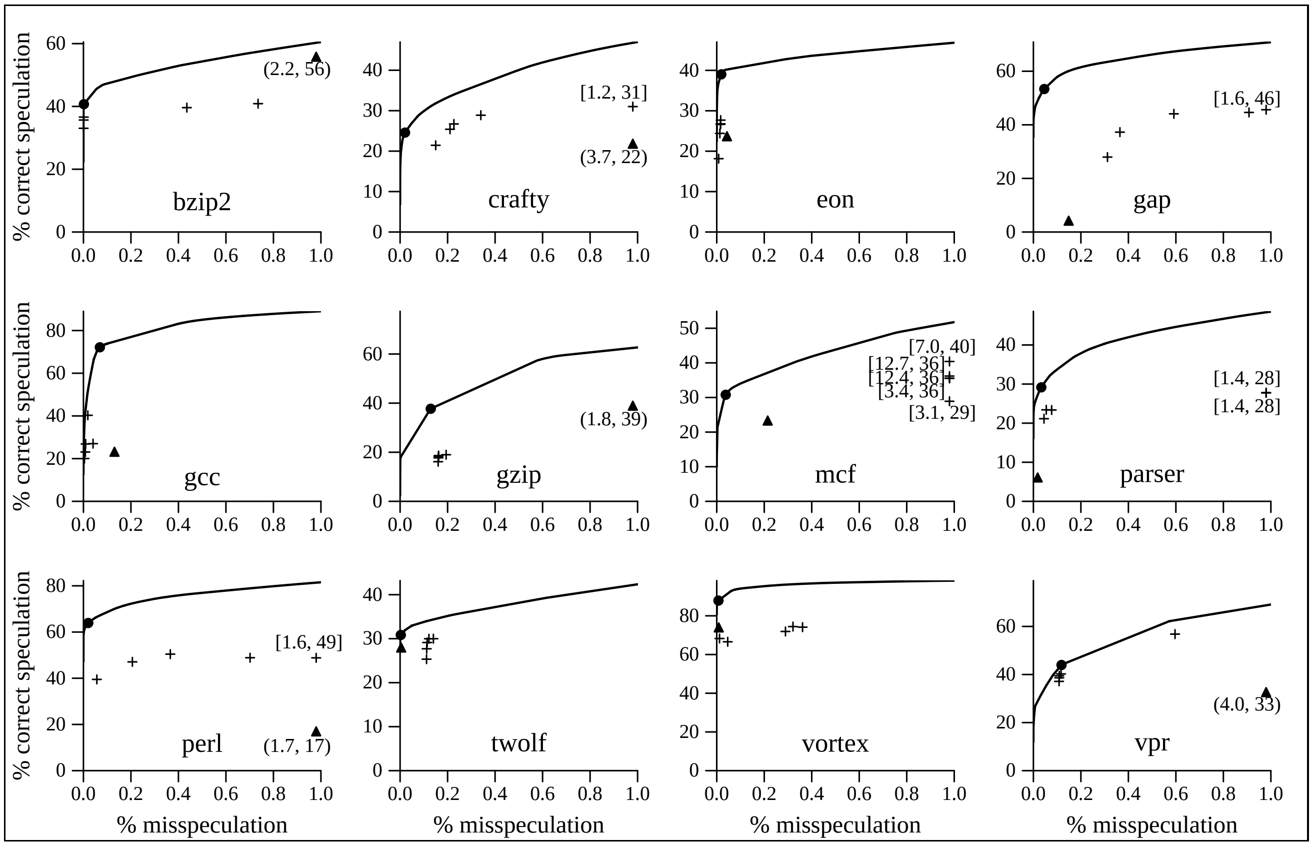


Figure 2. Correct/incorrect speculation trade-off demonstrates significant opportunity for software speculation. *The line represents the pareto optimal correct speculation rate that could be achieved for a given misspeculation rate with perfect knowledge of future branch outcomes (self-training). ●: a 99% threshold which is usually at the knee of the curve. As discussed in Section 2.2, △: results from using a training input (using a 99% threshold), +: results from using initial behavior to predict bias (using a 99% threshold and initial periods of 1k, 10k, 100k, 300k, and 1 million executions). Points that fall off the graph are labelled with their (x,y) location.*

tively consistent with other program behaviors (*e.g.*, loads that produce invariant values and memory dependences). This comes not as a surprise given the interaction between control flow and data flow in non-numeric programs.

In Section 2.1, we demonstrate that there is substantial opportunity for speculation that does not require the fine-grain control provided by hardware speculation. Then we explore (in Section 2.2) the challenges facing existing mechanisms to robustly exploit this opportunity. We conclude this section with a description of branches that make classification difficult, demonstrating that in many cases there is nothing to distinguish them from truly biased branches.

2.1. Opportunity for Software Speculation

The potential for aggressive software speculation can be seen by looking at branch bias across whole program runs. In software speculation, a decision to speculate or not is made when the code is generated. In making this decision, the ratio of correct to incorrect speculations must be considered. For a branch, this ratio is the branch's bias; for example, removing the conditional branch in Figure 1 will benefit the execution whenever the branch is not taken and a misspeculation occurs whenever the branch is taken. Speculation will improve performance whenever the aggregate benefit exceeds the aggregate penalty:

$$(correct\_preds \times benefit) > (incorrect\_preds \times penalty)$$

Thus speculation should be applied to all branches whose bias (or more precisely the ratio of correct-to-incorrect speculation) exceeds the ratio of the misspeculation penalty to the correct speculation benefit:

$$\frac{correct\_preds}{incorrect\_preds} > \frac{penalty}{benefit}$$

We can get a sense for the amount of opportunity for software speculation by looking at the cumulative distribution of branch bias. In Figure 2, we have sorted branches by their bias and plotted the Pareto optimal trade-off between correct and incorrect speculation. That is, as we move away from the the origin, we are speculating on an increasing number of branches, yielding more correct speculations (y axis) as well as more misspeculations (x axis). On each curve, we have marked (with a circle) the point that represents speculating on all branches with biases exceeding 99%; for example, the point in `gcc` indicates that over 70% of branches could be eliminated with misspeculation rate of less than 0.1%. We find this 99% threshold sits at or near the knee of the curve in each benchmark, allowing correct speculation on between 25 and 90 percent (average 46 percent) of branches with an average of about one misspeculation every 20,000 instructions. Clearly with such misspeculation rates, very aggressive speculation (*i.e.*, where the misspeculation penalty is two orders of magnitude larger than the benefit of correct speculation) can be profitable.

While these results demonstrate significant opportunity, they are somewhat optimistic. In selecting the set of branches for speculation, the behavior of the whole program's run (representing future knowledge) has been used. In the next section, we explore to what degree one can predict which branches will be biased using a pair of conventional mechanisms.

2.2. Predicting the Set of Highly-biased Branches

While significant potential exists, we need a mechanism to decide on which branch instances to speculate. In this section, we present some data that demonstrates that this can be a non-trivial issue. Specifically, we consider two existing mechanisms—using profile data from a previous run and using profile data from the beginning of a run—and discuss their limitations.

Profiling from a previous run: Many aspects of program behavior are consistent from one data set to the next; so using the behavior of one input to predict the behavior of another works pretty well, in general [5, 16]. Nevertheless, some program behaviors are entirely input dependent; many programs are parameterizable (for example the optimization level of a compiler) and the input parameters become predicates for frequently executed branches. This presents a problem for aggressive software speculation: for one input a branch may be 100% biased in one direction and for another input the same branch may be 100% biased in the other direction. Furthermore, if the profile input and the evaluation input do not exercise the same regions of code, there will be branches that will not be considered for speculation. In general, because of these two effects, selecting speculation from a previous input may have both lower benefit and more misspeculations than self-training.

If the training input differs materially from the evaluation input, the difference in program behavior can be substantial. In Figure 2, the benefit and misspeculation rates achieved from selecting biased branches (using a 99% threshold) from a differing input are plotted as triangles; the set of inputs used is described in Table 1. For these inputs, the benefit is reduced by a factor of 3 on average and the misspeculation rate increases by a factor of 10. Using a higher threshold does not significantly reduce the misspeculation rate for some of the worst offenders (`crafty`, `parser`, `perl` and `vpr`) and achieves only approximately 3/4ths of the benefit. The misspeculation rate can be reduced by averaging together a number of profiles; while this does reduce misspeculation rate it also reduces opportunity as input-dependent branches will not be speculated on (data not shown). Overall, this form of speculation control does not do a good job of approximating self-training, an observation also made in [16].

Bmark	Profile Input	Evaluation Input	Len
bzip2	input.compressed	input.source 10	19B
crafty	ponder=on ver 0	ponder=off ver 5 sd=12	45B
eon	rushmeier input	kajiya input	9B
gap	(test input)	(train input)	10B
gcc	-O0 cp-decl.i ⋆	-O3 integrate.i	13B
gzip	input.compressed 4	input.source 10	14B
mcf	(test input)	(train input)	9B
parser	(test input)	(train input)	13B
perl	scrabbl.pl	diffmail.pl	35B
twolf	(train input) fast 3	(ref input) fast 1	36B
vortex	(train input)	(reduced ref input)	32B
vpr	-bend_cost 2.0	-bend_cost 1.0	21B

Table 1. Simulation data sets and run length *As our intention was to demonstrate the fragility of offline profiling, we attempted to find reasonable inputs whose behavior differed from the evaluation set. In some cases, we diverged from the standard SPEC training sets for profiling, which in most cases are unrealistically similar to the ref inputs. All benchmarks were compiled for the Alpha architecture using peak compiler optimization. ⋆ Since the optimization level of gcc is hard coded, we had to modify its execution to give the appearance of -O0.*

Profiling from initial behavior: Another approach is to use a branch's initial behavior during the run to predict its overall behavior. A recent study [17] shows that, in many programs, initial behavior is a more effective predictor of branch bias than having a profile from a training data set, but, in some cases, a significant number of executions need to be recorded in order to reliably predict a branch's behavior. We have found this to be true for characterizing highly-biased branches as well.

Since most of the highly-biased branches exhibit that behavior for their whole lifetimes, the bias of an initial segment of execution is an effective predictor of which branches will be highly biased. In fact, 80% of the benefit of self-training can be captured by choosing to speculate only on branches whose bias exceeds 99% for their first 1,000 executions. The remaining 20% of benefit is derived from branches that are not initially biased, but whose overall behavior is biased.

The difficulty with this approach is the same one observed in [17]: some branches change their behavior—sometimes drastically so—during their execution. Specifically, in our experiments 7% of the static branches selected as biased from their initial 1,000 executions had an average bias for the whole run that was below 99%; more than a third of these branches had average biases less than 90%. The inclusion of these false positives results in a misspeculation rate of 2.6%; without them, the misspeculation rate is only 0.13%.

It is tempting to think that by observing a longer initial sequence before making a decision, misspeculations can be eliminated; however this is not particularly effective. The crosses in Figure 2 show the benefit/misspeculation tradeoffs for 5 different training period lengths: 1k, 10k, 100k, 300k, and 1 million executions. While increasing the initial sequence length does reduce misspeculation rate (points farthest from the y-axis correspond to the shorter training period), in some cases (`bzip2, perl`) it takes more than 300k executions to reach a rate comparable to self-training. In one case, `mcf`, even 1 million is insufficient, leaving it with a 3% misspeculation rate. Furthermore, the cost of a longer training period is a reduction of the achievable benefit.

The problem with both of these mechanisms is that they lack robustness. While each works well in certain circumstances, we have observed misspeculation rates as high as one per 100 instructions executed. Clearly such misspeculation rates are unacceptable for aggressive software speculation, where misspeculation detection and recovery could take hundreds of cycles.

We believe that this lack of robustness derives from the fact that once a decision to speculate is made it is never reconsidered. In Section 3, we demonstrate that, by adding a small amount of reactivity, the system can be made quite robust. We first, however, take a closer look at those branches that change behavior over their lifetimes.

2.3. Characterization of Changing Branches

When classifying branches from their initial bias, there are two challenging behaviors: 1) branches that start biased, but change to unbiased, and 2) branches that are initially unbiased that later become biased. The first category is the most serious because it represents potential misspeculations; the second category merely represents lost opportunity, and, as we show in Section 3, the loss is modest.

We looked closer at the first class of branches, hoping to find some characteristic that would distinguish them from branches that remain biased; we did not find a fool-proof mechanism, but we did not do an exhaustive analysis of program structure. Figure 3 provides some insight into the difficulty of this problem; five static branches from the benchmark `gap` are shown that are characterized as biased for at least the initial 20,000 executions (most of which are initially 100% biased) then change their behavior, in some cases completely reversing their bias. From solely looking at the sequence of initial branch outcomes, these branches are indistinguishable from completely biased branches.

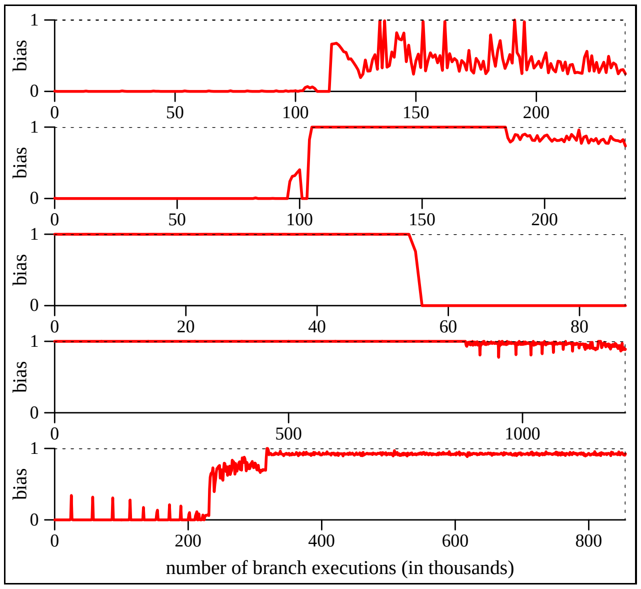


Figure 3. Five static branches with initially invariant behavior. *Branch bias averaged over blocks of 1000 dynamic instances. In all of these cases, the branch can be considered highly biased for at least the first 20,000 branch instances.*

Manual inspection of the source code provided little additional insight. In some cases, we found that the branch's behavior was correlated to a path or calling context and the branch's initial behavior was influenced by the control flow preceding the early executions. In other cases, no such correlation was identified, leaving us with the unsatisfactory explanation that the branch's behavior was "data dependent." In one case, the branch outcome was purely a function of a loop induction variable so that it was false the first 32,768 executions, then true the rest. We found no features that would enable these branches to be distinguished from truly biased branches.

3. Required Characteristics for Robust Software Speculation

In this section, we describe a simple model for controlling speculation that addresses the shortcomings of the aforementioned techniques. Despite its simplicity, this model is effective enough that its performance is comparable to, or exceeds, static self training (*i.e.*, using the same input for profiling as evaluation). We present

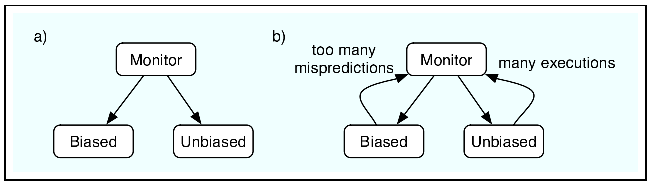


Figure 4. A finite-state machine model for branch behavior characterization.

Monitor period	10,000 executions
Selection threshold	99.5 percent
Misspeculation threshold	10,000 (+50 on misp., -1 otherwise)
Wait period	1,000,000 executions
Oscillation threshold	will not optimize a sixth time
Optimization latency	1,000,000 instructions

Table 2. Model Parameters.

this model separate from its implementation (discussed in Section 4) in order to demonstrate the fundamental requirements of such a classifier. We first describe the model and demonstrate its effectiveness; then, in Section 3.3, we demonstrate that the model is rather insensitive to most of its parameter settings so long as its basic form is intact.

3.1. A Simple Effective Model

The fundamental limitation of the models discussed in Section 2.2 is that they decide **once** whether or not to speculate on a branch. Although they have different mechanisms for making decisions, they both can be represented by the diagram in Figure 4(a). This limitation translates into a lack of robustness because there is no recourse when a branch has been incorrectly characterized.

The key to robustness is allowing branches to be reclassified when their behavior changes. Figure 4(b) shows a model with two additional transitions, both back to the monitor state. From the biased state, the transition should be taken when the branch is resulting in an undesirable rate of misspeculations. From the unbiased state, it is merely necessary to periodically revisit the monitor state. As we will show in our sensitivity analysis, the existence of these transitions is fundamental; most every other attribute of this model is of secondary importance.

Nevertheless, to evaluate the model, we need to assign values for the various model parameters. We attempted to select parameters for the model that would facilitate its implementation in a real system; Table 2 relates what we believe to be reasonable parameters, as described below.

To a large degree, the model parameters were chosen to reduce the effort required by the optimization system. Every transition into or out of the *biased* state requires the code to be re-optimized. The main drawback of using a model like Figure 4(b) is the potential for oscillating in and out of the biased state. We use a number of techniques to mitigate such oscillation:

1. First, we use a moderately long monitoring period (10,000 executions) as a simple filter for reducing the number of false positives.

2. Second, we introduce some hysteresis by using a stricter threshold for entry into the biased state than for eviction. For example, to target branches with average bias of greater than 99%, we require the bias to be greater than 99.5% to begin speculation, and we only evict branches when their bias falls below 98% for a non-trivial time period. This is implemented in the model using a saturating counter that counts up 50 on a misspeculation and down by one on a correct speculation; the branch is evicted if the counter reaches 10,000 (requiring at least 200 misspeculations). This hysteresis is necessary to tolerate short bursts of misspeculations by otherwise biased branches.

3. Third, we use a relatively long waiting period (1 million executions) in the unbiased state. In addition to reducing the frequency at which a branch's classification needs to be reconsidered, increasing this period reduces the likelihood that a branch which is only temporarily biased will be selected for speculation.

4. Fourth, we limit the number of times each branch can oscillate. This is a necessity for the small number ($\sim$50 of over 7000) of branches that otherwise oscillate hundreds or thousands of times, even for our relatively short program runs. After a threshold number of oscillations, we conservatively choose not to speculate on these branches. We have found this limit to have little impact on the system's results with a two-thirds (on average) reduction in the number of requested re-optimizations.

For transitions into or out of the biased state, which are accompanied by re-optimization requests, we model the latency to make modifications to the code. Given the likely abundance of thread-parallel resources in future processors, we assume that re-optimization is performed in parallel with execution and hence has latency, but no overhead. We use a latency of 1 million instructions (the functional simulations described below has no notion of time). Thus, after a branch has been selected for speculation, we wait 1 million instructions before counting correct and incorrect speculations. Likewise, when a branch is evicted from the biased state, correct and incorrect speculations continue to be counted for the following 1 million instructions, until the repaired code fragment can be deployed. While the value of this latency is somewhat arbitrary, it represents an estimate of the latency for the first stage of our dynamic opti-

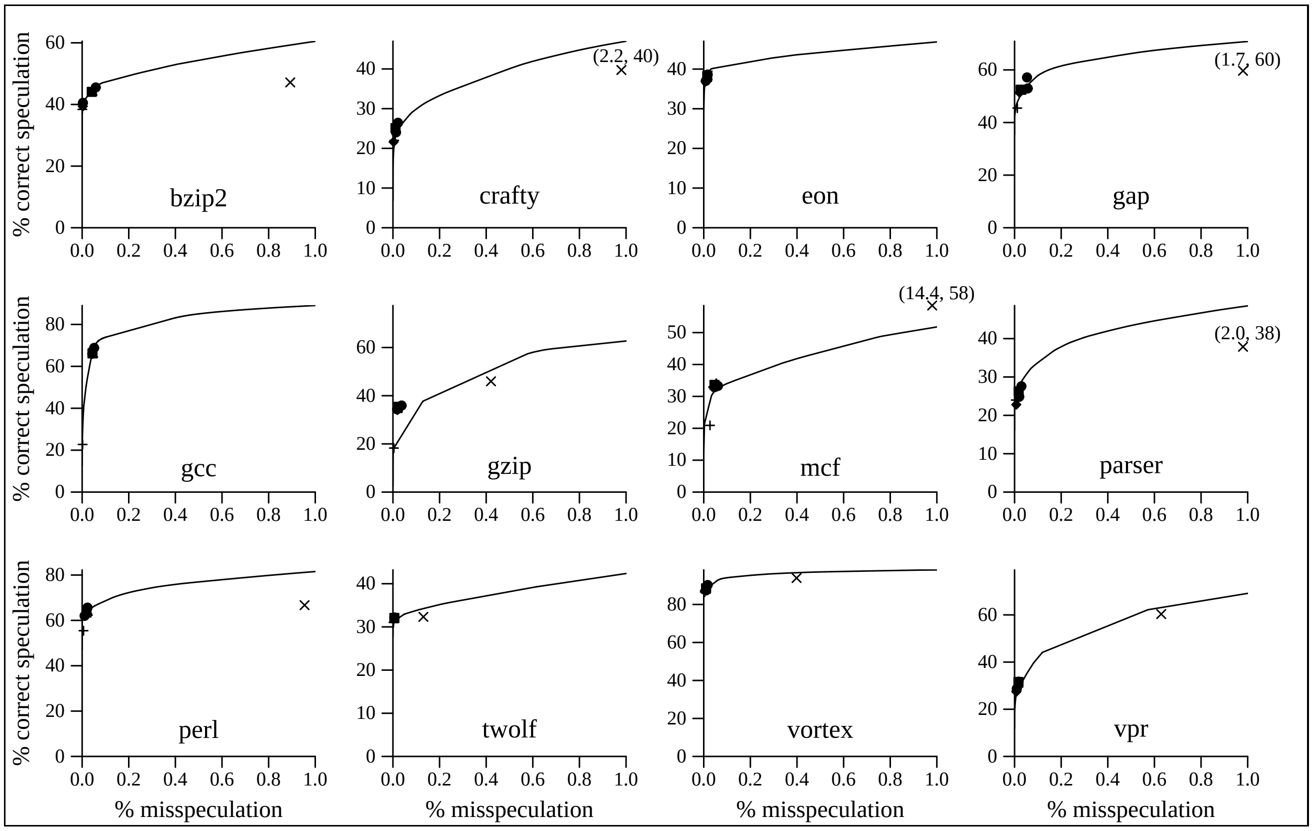


Figure 5. Reactive control performs comparably with self-training. *The line still represents the correct/incorrect speculation trade-off achievable through self-training. The other marks are results from the reactive control model.* ***square:*** *baseline,* ***x:*** *no eviction (without biased→monitor transition),* ***+:*** *no revisit (without unbiased→monitor transition),* ***circle:*** *eviction by bias sampling,* ***ellipse:*** *shorter revisit period,* ***diamond:*** *lower (1,000) eviction threshold,* ***triangle:*** *sampling. As all of the points except the* ***x*** *and* ***+*** *are collocated, the behavior of the model is primarily only sensitive to the presence of all of the transitions.*

mizer for the median-sized optimization region (∼100 instructions).

3.2. Reactive Model Performance

In this subsection, we demonstrate two characteristics of the model: 1) its ability to select a set of branches on which to speculate is comparable to what is achievable by self training, and 2) the model is rather forgiving with regard to its implementation, except that all of the transitions must be present.

As some of the changes of program behavior are only observed in long runs of the programs, we did the experiments in the context of a functional simulator to enable us to simulate the benchmarks to completion. These runs explore the behavior of the speculation control mechanism in an abstract context, independent of any particular speculation mechanism. The behavior of each branch is tracked independently, with the exception of modeling the optimization latency.

Figure 5 plots the results of these simulations in the same format as Figure 2. For reference, the self-training line is shown. The performance of the model with the parameters shown in Table 2 is shown by a square dot. In all benchmark runs, the performance is competitive with self training. In `gzip` and `mcf`, the model outperforms static self training, because it can adapt to the low frequency time-varying behavior of branches; for example, the average bias of the middle branch in Figure 3 is about 60% so it should not be selected for speculation by a static mechanism, but the reactive model can discern that its behavior consists of two highly-biased regions, each of which can be exploited.

Table 3 presents results regarding how often branches transition into and out of the bias state. Of the static (conditional) branches touched during the executions, 34 percent enter the biased state some time during the execution. Of these, about 7 percent–about 2% (37%×7%) of touched branches–are later evicted from the biased state. Some of these evicted branches get evicted more than once; the average evicted branch gets evicted 1.6 times (total evictions/static branches evicted). Almost half of conditional

	Static branches			total	%	misspec
Bmark	touch	bias	evict	evicts	spec.	dist.
bzip2	282	109	6	15	44.1%	26,400
crafty	1124	396	138	276	25.1%	109,366
eon	403	95	3	3	38.3%	105,552
gap	3011	1045	167	201	52.5%	36,728
gcc	7943	2068	11	12	66.3%	20,802
gzip	314	66	7	12	35.4%	43,043
mcf	366	210	22	47	33.6%	12,896
parser	1552	284	53	124	26.3%	50,643
perl	1968	1075	58	64	63.4%	55,382
twolf	1542	440	19	22	32.1%	165,711
vortex	3484	1671	67	104	88.5%	92,163
vpr	758	340	16	38	31.6%	65,588
ave		34%	2%	76	44.8%	65,000

Table 3. Model Transition Data. *Only a small fraction of branches need to be evicted from the biased state and mispredictions can be very far apart.*

branches can be eliminated by the speculation, incurring only one misspeculation every 65,000 dynamic instructions, on average.

3.3. Sensitivity Analysis

These above results are surprisingly insensitive to exactly how the model is implemented. Having explored a number of configurations, we have found that most changes merely shift the model's performance up or down along the self-training curve. We have included some sensitivity results in Figure 5; in many cases the points in the figure overlap, emphasizing the insensitivity. We describe these experiments below:

1. **Lower Eviction Threshold:** Lowering this threshold from 10,000 to 1,000 makes the system less tolerant of branches with varying biases, leading to a more conservative (lower correct and incorrect speculations) system.
2. **Evicting By Sampling:** Rather than tracking each branch's misspeculation rate continuously, this experiment periodically re-samples the branch's bias to make the eviction decision. Computing the bias of 1,000 samples every 10,000 executions (a 10% duty cycle) ends up evicting more branches resulting in a slight reduction of both correct and incorrect speculations.
3. **Sampling in "monitor" State:** Using a 1-in-8 sampling rate adds a little additional uncertainty causing a few unbiased branches to be declared biased. Larger sampling rates can be tolerated as well by lengthening the monitor period to keep the number of samples constant.
4. **More Frequent Revisit:** By lowering the revisit wait time by an order of magnitude to 100,000 executions, we introduce two competing factors: 1) a reduction of time spent by biased branches in the unbiased state (good), and 2) branches that are only momentarily biased are more likely to be selected and later evicted (bad).
5. **Optimization Latency:** All of the results discussed include a latency for optimizing and deploying new code. If this latency is set to zero (data not shown), the amount of correct speculations increases by only 0.1% and the number of misspeculations is reduced by a factor of 1.1. This latency tolerance arises from two factors: 1) the branch in question may not be executed again for many instructions, and 2) although the branch may not be considered *highly* biased, it still may be biased in the same direction; as a result only a fraction of future executions will cause misspeculations. Figure 6 shows the misprediction rate (fraction of branches not in the original bias direction) in the vicinity (up to 64 branches) of a transition out of a highly biased state. Over 50 percent of the static branches have biases less than 30% in the transition period. It is really only the 20 percent of branches that become perfectly biased in the other direction that require quick action.

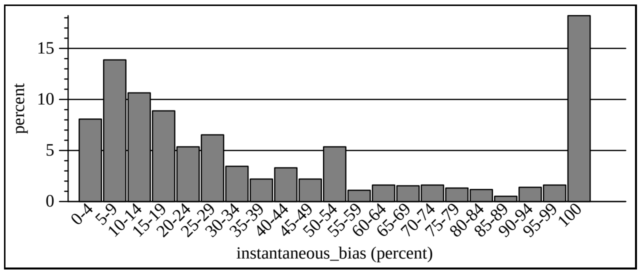


Figure 6. Instantaneous misprediction rate when a biased branch transitions from being biased. *Two behaviors are common when a branch leaves the* biased state*: 1) the branch bias softens (bias direction stays the same, but the percentage reduces), 2) the branch becomes perfectly biased in the other direction.*

If, however, the added transitions are removed, behavior changes significantly. If the revisit transition (unbiased→monitor) transition is eliminated, the model achieves only a little more than 80% of the correct speculations of the baseline. Removing the eviction transition (biased→monitor) increases misspeculation rate by almost two orders of magnitude. Table 4 relates the average benefit and misspeculation rates for each experiment.

The fact that the model is so insensitive to its implementation is a great boon to system implementors. It means that the model can be implemented in an efficient manner without a significant impact on its performance.

configuration	correct	incorrect
no revisit	35.8%	0.007%
lower eviction threshold	42.9%	0.015%
evictionby sampling	43.6%	0.021%
baseline	44.8%	0.023%
sampling in monitor	44.8%	0.025%
more frequent revisit (100k)	46.1%	0.033%
no eviction	53.9%	1.979%

Table 4. Model Sensitivity *Only the **no revisit** and **no eviction** configurations truly differ from the baseline.*

4. Model Implementation

While the previous section explored the proposed speculation control mechanism in the abstract, this section explores it in the context of a particular system: Master/Slave Speculative Parallelization (MSSP) [19, 20], to validate the results in the previous section with performance data. This section is organized to first provide a brief overview of the salient features of MSSP relevant to this work (Section 4.1). Section 4.2 describes our experimental methodology, and, in Section 4.3, we demonstrate the sensitivity of an MSSP execution to its control model.

4.1. MSSP Overview

Master/Slave Speculative Parallelization (MSSP) is an execution paradigm that provides a framework to support speculative program transformations. Unlike the speculative program transformations performed for EPIC/VLIW systems like IA-64 [3] and Transmeta's Crusoe [7], MSSP uses a speculative version of the program that has no code to check the speculations. Checking is not performed by the speculative program because eliminating the checks enables eliminating as much as two-thirds of the dynamic instructions executed by the speculative program.

To detect misspeculation, MSSP, like SlipStream [15], uses a second *trailing* execution to "check" the results of the speculative program's execution. While SlipStream checks the speculative program at every instruction boundary, MSSP performs checks only at selected instructions referred to as *task boundaries*. Because the programs need only correspond at the task boundaries, there is significant flexibility in the optimizations that can be applied to the speculative program. Specifically, MSSP is free to use optimizations that restructure the code (*e.g.*, in-lining, redundant load/expression elimination, register promotion, and constant folding). In contrast, SlipStream is constrained to use a strict subset of the original program.

The details of MSSP's operation are not essential for the purpose of this paper. For those unfamiliar with MSSP, only three characteristics merit attention: First, MSSP speculates at the granularity of a *task* (the sequence of instructions between two task boundaries); any (observed) misspeculation will prevent the commit of a whole task. Second, MSSP systems can be engineered such that the speculative program's execution is generally the performance bottleneck, so that benefit from any speculation optimizations performed on the speculative program translate directly into overall system performance. Third, the misspeculation detection and recovery latency can be hundreds of cycles^1, because misspeculations are detected by the trailing execution significantly after they occur and require restarting the speculative program from the trailing program's state. Thus, MSSP embodies an aggressive software speculation mechanism and requires an effective speculation control system to achieve robust performance.

	Leading Core	**Trailing Cores**
Pipeline	4-wide, 12-stage pipe	2-wide, 8-stage
Window	128-entry inst. window	24-entry
ALUs	4 (1 complex) and 2 LD/ST	2, 1 LD/ST
Caches	64KB 2-way SA 64B blocks	8KB 8-way, 64B
	3 cycle (including AGEN)	same latency
Br. Pred.	8Kb gshare, 32-entry RAS,	same
	and 256-entry indirect	same
L2 cache	shared 1MB, 8-way SA w/64B blocks	
	10-cycle access minimum	
Coherence	10-cycle minimum hop between processors	
	(uncongested network)	
Memory	200-cycle lat. minimum (after L2)	

Table 5. Simulation Parameters

4.2. Experimental Methodology

The next section's performance results were collected with an execution-driven simulator that uses the SimpleScalar loader and syscall emulation functionality. Our simulations model an asymmetric chip multiprocessor (CMP) with one large core (used for the leading execution) and a collection of 8 smaller cores (used for the trailing execution). Our simulator models the activities of each core as well as the coherence protocol used to communicate between them. Parameters for the cores and the memory system are provided in Table 5.

Unlike prior work on MSSP [19, 20] that used self-training and an off-line generated speculative program, our simulator emulates a dynamic optimization system that generates speculative program fragments based on what is learned about the program as its execution progresses. Thus the system identifies hot program regions, characterizes them, and generates optimized versions which are then executed by the execution-driven simulator. While this system lets us predict the performance of the executed spec-

1 By comparing to a modified version of our simulated system that does not apply transformation when they are invalid, we computed the true cost of a misspeculation in our simulated system to be around 400 cycles.

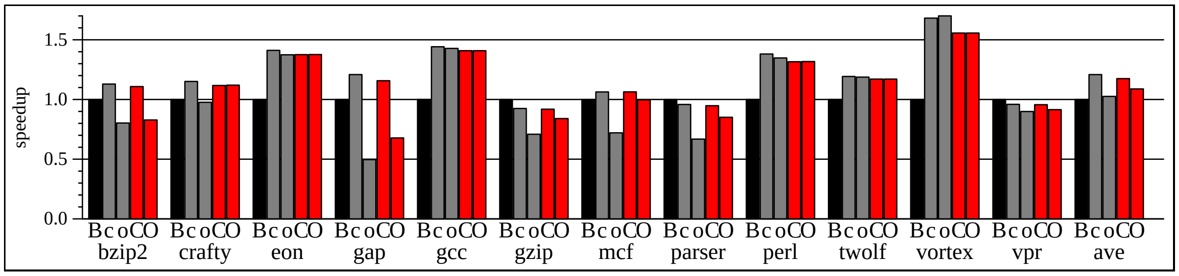


Figure 7. Lack of reactivity severely impacts performance. *Performance of an open-loop (**o**,* i.e.*, no reactivity) system significantly trails that of a closed-loop (**c**) system. Extending the initial monitoring period to 10,000 instances (data points **C** and **O**) from 1,000 only slightly mitigates the discrepancy, even for these short simulations. (data normalized to baseline (B) of normal superscalar execution.)*

ulative program, because the dynamic optimizer is implemented as part of the simulator we cannot determine how long it takes to generate the speculative program fragments. Currently, we model the optimization process as taking a fixed latency (a parameter to our simulator). It is important to note that, while the optimization process has a certain latency, that latency does not translate directly into overhead, as the optimization process is performed on the trailing processors when they are idle, in parallel with the execution of the program.

As simulating a chip-multiprocessor is somewhat computationally intensive, we use relatively short (200 million instruction) runs of the benchmarks. Each run begins from a checkpoint 5 billion instructions into the execution with cold caches and predictors. To approximate the behavior of the dynamic optimizer in steady state, we've parameterized its "hot region detector" to find and deploy regions artificially fast, to minimize the impact of "warming up" the dynamic optimizer.

Experiments with our functional simulations suggest that the shortness of these runs does perturb their sensitivity to model parameters, in the following manner. First, these short runs are more sensitive to the time in the monitor state; longer monitor periods significantly reduce the number of correct speculations. Second, the shorter runs are more sensitive to optimization latency, again because it reduces the number of correct speculations. Third, shorter runs are less sensitive to the choice of control policy, because over a shorter period of time a branch is less likely to change bias and the impact of that change is bounded. Nevertheless, these runs still demonstrate the key thesis of this paper, that the choice of control policy has a first-order effect on performance.

For the graphs in the next section, we are primarily concerned with the relative performance of the MSSP configurations. In spite of this, we have chosen to normalize our results to a non-MSSP execution on the large core of our CMP, in order to demonstrate that the choice of a poor control policy can reduce MSSP performance below that of a "vanilla" superscalar. The speedups of MSSP relative to the baseline should be interpreted as lower bounds as these short runs exhibit an unrepresentatively large warm-up overheads.

4.3. Results

Due to space constraints, we limit our performance simulation-based validation to the two most important results: 1) the necessity of the eviction ($biased \rightarrow monitor$) arc in the closed-loop model, and 2) the lack of sensitivity to optimization latency.

Figure 7 compares the performance of closed- and open-loop (*i.e.*, those with and without the eviction arc). Even in these short runs, which are desensitized to this effect (see Section 4.2), the absence of the eviction arc significantly impacts performance (18%). Using a longer monitoring period somewhat reduces the impact of using the open-loop policy, but an 11% descrepency remains, even for these short runs. A few benchmarks (*e.g.*, `eon`, `gcc`, `perl`, and `twolf`) show limited sensitivity because few branches need re-characterization at this program point. Vortex's large working set makes it sensitive to the monitor period in these short runs.

The above experiments were done with a optimization latency of zero. Figure 8 shows that optimization latency has little impact on performance. While optimization latency potentially reduces the number of correct speculations and increases the number of incorrect speculations, in practice, optimization latencies of 0, 100k and 1 million cycles have almost indistinguishable performance ($< 2\%$). As short runs have less time to amortize the optimization latency, we expect that in longer runs the sensitivity would be even lower.

These simulations also sheds some light on the correlations between branches. Because MSSP performs speculation at the granularity of tasks, we found that in some cases the misspeculation rate is noticeably lower than is predicted by the abstract model. This occurs when multiple failed speculations occur within the bounds of one task, resulting in a single task misspeculation. In some cases, this arises

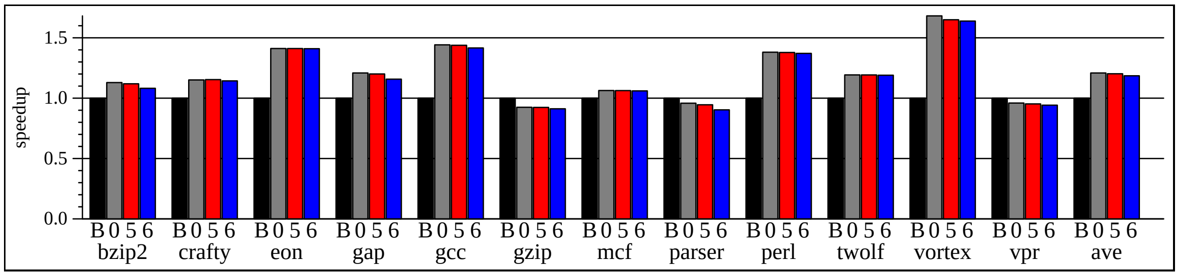


Figure 8. As predicted by the model, performance is rather insensitive to optimization latency. *Performance when (re)optimization latency is 10^5 (**5**) and 10^6 (**6**) cycles is only slightly less than when (re)optimization is performed instantaneously (**0**). (again, data normalized to baseline (B) of normal superscalar execution.)*

from multiple instances of the same static branch. We also found behavior changes in one static instruction are correlated to that of another. The benchmark `vortex` demonstrates this clearly. Figure 9 plots the behavior of the 139 static branches that have significant periods of both being biased ($> 99\%$) and unbiased ($< 99\%$). The period when each branch is highly biased is shown as a horizontal line, and it can be seen that some instructions change their behaviors in groups. The behavior is present (generally to a lesser extent) in about half of the SPEC2000 integer benchmarks.

The fact that branches are correlated means that fewer code re-optimizations are necessary than transitions of the model. In our current implementation, we find that about half of the time it is necessary to re-optimize a code region (a function or loop body in the distiller) there is more than one change to make.

5. Related Work

In a keynote talk [14], Smith demonstrated anecdotal examples of branches that drastically change their bias during program execution, but does not discuss them in the context of any mechanism or policy.

A number of researchers have recently been concerned with the "phase behavior" of programs, primarily in the context of adaptive processors [2, 6, 11, 12]. Generally, the phases they wish to distinguish are large (to amortize the cost of adaptation) and significantly different (typically regions are characterized by distinct regions of the static program). In contrast, the present work is concerned with tracking the change of behavior of individual branches.

While the Dynamo dynamic optimizer [1] does not monitor program behavior directly, their preemptive flushing of the fragment cache will force re-optimization of regions when phase changes (of the sort described in the previous paragraph) occur. As such phase behavior is somewhat orthogonal to the behavior changes of individual instructions, this policy will likely perform somewhere between closed-loop and open-loop policies. Because Dynamo used limited speculation, this policy was sufficient because of the limited downside.

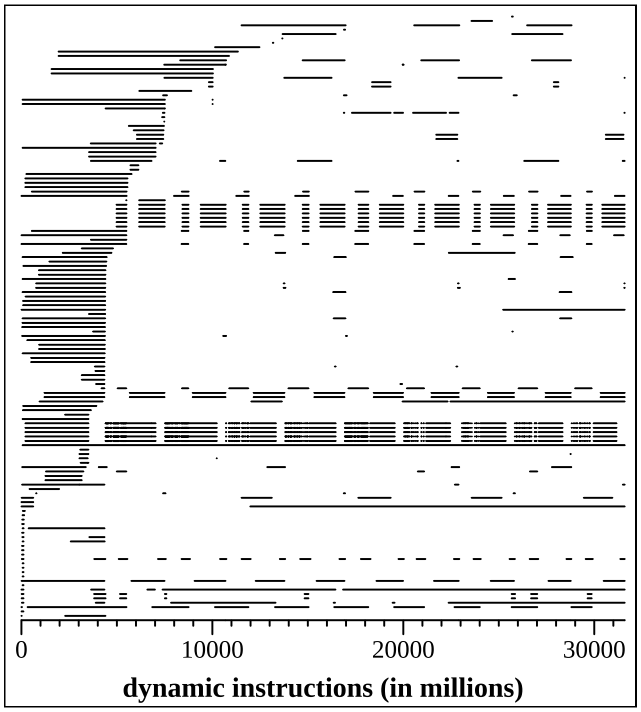


Figure 9. Behavioral changes for different static branches may be correlated. *This plot shows the 139 static branches (one per horizontal track) in the benchmark `vortex` that flip between being characterized biased and unbiased; each horizontal line shows the period of time when the branch is considered biased. It can be seen that there are groups of branches that change behavior together.*

It is our belief that Transmeta's Code Morphing Software (CMS) reactively controls its software speculation as is described in this article, but they refuse to publicly comment on the issue.

6. Conclusion

In this paper, we have demonstrated that software speculation can be effectively controlled and the key is introducing a reactive control system. By monitoring program behavior throughout the application's execution and re-optimizing when program behaviors change, software speculation can be made robust. In fact, such mechanisms achieve results comparable to self-training, a practice generally thought to be optimistic.

We described a simple yet effective model for controlling software speculation. We have shown that the model is surprisingly insensitive to its parameters, which should greatly facilitate its efficient implementation. Most notably, the model is latency tolerant, an important characteristic when the latency of code (re-)optimization is considered. By virtue of conservatively selecting highly-biased program behaviors on which to speculate, a reactive control system can maintain a low misspeculation rate while only periodically sampling program behavior.

7. Acknowledgements

We thank Lee Baugh, Pierre Salverda and the anonymous reviewers for their contributions to improving the content and presentation of this paper. This work was supported in part by National Science Foundation grants CCR-0311340, CCF-0347260 and EIA 02-24453, Intel and equipment donations by AMD, HP, and Intel.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[2] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[3] C. Dulong. The IA-64 Architecture at Work. *IEEE Computer*, 31(7):24–32, July 1998.

[4] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. Performance Characterization of a Hardware Framework for Dynamic Optimization. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[5] J. Fisher and S. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[6] M. Huang, J. Renau, and J. Torrellas. Positional Processor Adaptation: Application to Energy Reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[7] A. Klaiber. The Technology Behind Crusoe Processors. Transmeta Whitepaper, Jan. 2000.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[9] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4), Nov. 1993.

[10] A. Moshovos, S. E. Breach, T. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[12] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[13] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

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

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[16] D. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991.

[17] Y. Wu, M. Breternitz, J. Quek, O. Etzion, and J. Fang. The Accuracy of Initial Prediction in Two-Phase Dynamic Binary Translators. In *2^{nd} IEEE/ACM Symp. Code Generation and Optimization*, Palo Alto, CA, Mar 2004.

[18] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[19] C. Zilles. *Master/Slave Speculative Parallelization and Approximate Code*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, Aug. 2002.

[20] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.