Adaptive Software Speculation for Enhancing the Cost-Efficiency of Behavior-Oriented Parallelization

Yunlian Jiang Xipeng Shen
Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{*jiang,xshen*}*@cs.wm.edu*

Abstract

Recently, software speculation has shown promising results in parallelizing complex sequential programs by exploiting dynamic high-level parallelism. The speculation however is cost-inefficient. Failed speculations may cause unnecessary shared resource contention, power consumption, and interference to co-running applications. In this work, we propose adaptive speculation and design two algorithms to predict the profitability of a speculation and dynamically disable and enable the speculation of a region. Experimental results demonstrate significant improvement of computation efficiency without performance degradation. The adaptive speculation can also enhance the usability of behavior-oriented parallelization by allowing more flexibility in labeling possibly parallel regions.

1 Introduction

Many sequential programs—such as a parser parsing sentence by sentence and a compression tool compressing buffer by buffer—have high level parallelism, but are often difficult to parallelize, because of code complexity, input-dependent behavior, and uncertainties in parallelism.

Recently, *behavior-oriented parallelization*(*BOP*) [3] has been proposed to solve that problem. The goal of *BOP* is to improve the (part of) executions that contain coarse-grain, possibly input-dependent parallelism. Unlike traditional code-based approaches that exploit the invariance holding in all cases, *BOP* utilizes partial information to incrementally parallelize a program for common cases. As a software speculation technique, *BOP* guarantees the execution correctness by protecting the entire address space. The protection overhead and uncertain dependences in the code, however, may cause a speculation to abort without contributing any benefits. The useless speculations waste the computing resource and power of a machine, causing unnecessary interferences to the useful processes and other applications that run in the same system simultaneously. This issue is particularly important for a multi-programming environment and a power-constrained environment like a portable computing device. The current *BOP* system is oblivious to that issue, blindly speculating every possibly parallel region (**PPR**).

This work proposes adaptive speculation. The goal is to make *BOP* avoid unprofitable speculations but meanwhile keep profitable speculations unaffected, hence improving the cost-efficiency without sacrificing the parallelized program performance. As a side benefit, adaptive speculation can also make *BOP* easier to use by allowing users to label *PPR*s more flexibly: The unprofitable *PPR*s will be turned off automatically.

It is however difficult to predict speculation profitability through program code analysis, because the profitability depends on program inputs and runtime behavior. By treating the problem as a statistical learning task, we develop two adaptive algorithms that are able to learn the profitability patterns of a *PPR* during runtime. A complexity in the learning is that the profitability of the earlier instances is not always unveiled: If a *PPR* instance is not executed speculatively, *BOP* cannot determine its profitability. The two algorithms manage to learn from the partial information and adapt themselves to the dynamic changes in profitability patterns.

The first algorithm is an extension to last-value predictors and uses a dynamically adjustable threshold for adaptation. The second algorithm exploits long-term history and offers more flexibility in control by separating different factors apart. Both algorithms are reconfigurable, providing some "knobs" for users to adjust the tradeoff between parallelism exploitation and cost savings.

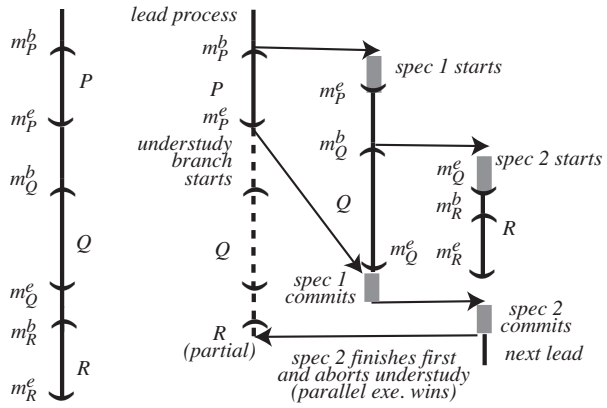
On six different *PPR* patterns, the two algorithms show 82% and 86% average prediction accuracy. On three real programs, the adaptive schemes help *BOP* system save more than 50% computation cost, without performance degradation (in many cases, the performance is even improved).

```
...
while (1) {
 get_work();
 ...
 BeginPPR(1);
 work();
 EndPPR(1);
 ...
}
```

Figure 1. possible loop parallelism

```
...
BeginPPR(1);
work(x);
EndPPR(1);
...
BeginPPR(2);
work(y);
EndPPR(2);
...
```

Figure 2. possible function parallelism



(a) Sequential execution of PPR instances *P, Q,* and *R* and their start and end markers.

(b) A successful parallel execution, with *lead* on the left, *spec* 1 and 2 on the right. Speculation starts by jumping from the start to the end marker. It commits when reaching another end marker.

Figure 3. An illustration of the sequential and the speculative execution of 3 *PPR* instances

In the rest of this paper, Section 2 gives a brief review of *BOP* system. Section 3 presents the algorithms for adaptive speculation. Section 4 reports evaluation results, followed by a discussion on related work and a short summary.

2 Review of BOP

In *BOP*, multiple *PPR* instances are executed at the same time. A *PPR* is labeled by matching markers: *BeginPPR(p)* and *EndPPR(p)*. Figures 1 and 2 show the marking of possible loop parallelism and possible function parallelism respectively.

Figure 3 illustrates the run-time setup. Part (a) shows the sequential execution of three *PPR* instances, P, Q, and R (which can be the instances of either a single *PPR* or different *PPR*s). Part (b) shows the speculative execution. The execution starts as the *lead* process. When the lead process reaches the start marker of P, m_P^b, it forks the first speculative process, *spec 1*, and then continues to execute the first *PPR* instance. Spec 1 jumps to the end marker of P and executes from there. When spec 1 reaches the start of Q, m_Q^b, it forks the second speculative process, *spec 2*, which starts executing from the end of Q.

At the end of P, the lead process starts an *understudy* process, which executes the code after m_P^e **non-speculatively**, in parallel to the speculative execution by the spec processes. Meanwhile, the lead process falls into sleep. The arrival of spec 1 to m_P^e wakes up the lead process, which helps spec 1 to check for conflicts. If there are no conflicts, the lead process commits its changes to spec 1 and aborts itself. Spec 1 then assumes the role of the lead process so that later speculation processes are handled recursively in a similar manner. The kth spec is checked and combined after the first $k-1$ spec processes commit.

Note that the understudy and the speculative processes execute the same *PPR* instances. The understudy's execution starts later but is guaranteed to be correct; on the other hand, the speculative execution starts earlier but contains more overhead and is subject to possible dependence violations. They form a sequential-parallel race. If the speculation completes earlier correctly, the parallel run wins and the parallelism is successfully exploited; otherwise, the understudy's run wins and the speculative execution loses and forms a waste of computing resource. (Using the understudy process is to ensure that the execution supported by *BOP* won't be much slower than a sequential run.)

3 Adaptive Speculation

Automatically avoiding unprofitable speculations has three benefits. First, it saves computing cost and thus improves system cost-efficiency. This saving is especially important when multiple programs compete for CPUs, cache, memory, or other computing resource. It is also critical when the running environment has very limited power supply—for example, in a portable device. Second, it boosts the usability of *BOP*. The adaptive scheme offers greater flexibility for *BOP* users to mark *PPR*s. Users can mark more *PPR*s in a program as the unprofitable ones will be disabled automatically during runtime. Finally, adaptive speculation may enhance program performance. When a *PPR* instance is predicted as unprofitable, *BOP* not only avoids the creation of spec processes, but also disables the creation of the understudy process and the bookkeepings (for conflict detection) by the lead process. Furthermore, it reduces the interferences from the spec processes to the other processes in terms of the contention for shared resources.

The key for adaptive speculation exists in accurate prediction of the profitability of a *PPR* instance. Unlike in other runtime prediction problems such as dynamic branch prediction, the correctness of a past prediction in this problem is not always uncovered. If a *PPR* instance is predicted as unprofitable and is not speculatively executed, the correctness of the prediction will remain unknown. The adaptive speculation, therefore, has to learn from the partial information. In addition, some runtime predictions like branch prediction are usually implemented on hardware with strict constraints on both space and time, whereas, adaptive speculation is a software scheme, permitting more sophistication in the prediction algorithms.

This section presents two dynamic predictors that learn from the prior instances of a *PPR* and predict whether the next several instances are profitable or not. They both focus on loop parallelism, learning from prior iterations of a loop *PPR*. For the purpose of clarity, the following description assumes that the speculative depth is 1—that is, there is at most one speculation process at any time.

3.1 Algorithm 1: Extended Last-Value-Based Prediction

We start with a simple design. Last-value based prediction is a typical technique used in runtime behavior prediction. It uses the value of the last few instances to predict the next instance. Our first attempt is to extend last-value based predictors to predict the profitability of speculations.

Figure 4 depicts the flow chart of the extended algorithm. When a speculation of a *PPR* turns out to be profitable, the algorithm works the same way as a last-value-based predictor—predicting the next instance as profitable. On the other hand, when the previous speculation is unprofitable, the algorithm disables the speculation of the next *specBar[i]* instances of the *PPR* (i is the identity number of the *PPR*). The value of *specBar[i]* is dynamically adjusted: A failed speculation will make it increase by a factor of α ($\alpha \geq 1$); a successful speculation will reset the threshold to 1. (We name α the **bar-raising factor**.) This adjustment scheme results in an exponential increase of *specBar[i]* on consecutive failures of speculations. The increase makes the algorithm suitable for the cases when the speculations of most of the *PPR* instances are unprofitable, which could happen if the granularity of a *PPR* is too small or the region contains some dependences occurring frequently. The reset on a success, on the other hand, makes the algorithm adaptable to phase changes in program behavior.

The bar-raising factor, α, controls the increase speed of the penalty on consecutive speculation failures. The larger it is, the faster the penalty increases, thus, the less aggressively the *BOP* will speculatively execute the *PPR*. The value of α can be chosen statically or dynamically. The static

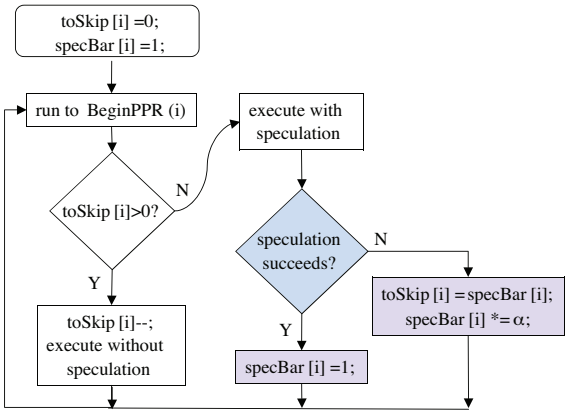


Figure 4. Extended last-value-based prediction

way is to let users select the value based on their desirable cost savings or experiments on some representative *PPR* patterns. Section 4 tests this strategy and shows that a single α value is able to make the algorithm predict the profitability of 6 different *PPR* patterns accurately. A dynamic strategy is to determine (and adjust) the value during an execution based on the actual program behavior.

In the integration of the algorithm into *BOP*, the lead process conducts all the operations of the algorithm except those in the non-white boxes in Figure 4. The answer to the question in the dark diamond box comes from the winner of the race between the understudy and the spec processes. The new lead process (i.e., the winner of the race) conducts the operations in the two dark rectangle boxes.

3.2 Algorithm 2: Decayed-History-Based Prediction

The first algorithm is not flexible in two ways. First, a single successful speculation will make *BOP* forget all the prior knowledge of a *PPR*, whereas, long-term history may be useful for the recognition of some profitability patterns. Second, although the use of a single parameter, α, is simple, it limits the flexibility in configurating the adaptive scheme; the factors affecting the adaptive scheme cannot be adjusted separately.

Our second algorithm tries to address those constraints. Figure 5 shows this more flexible algorithm. The responsibilities of different processes are marked in Figure 5 the same way as in Figure 4. This algorithm incorporates long-term history into a variable that contains the decayed gain produced by prior speculations. For a given *PPR* instance, the algorithm predicts that it is worthy of being speculatively executed only if either the current gain is large enough or the speculation of the *PPR* has been disabled for a long

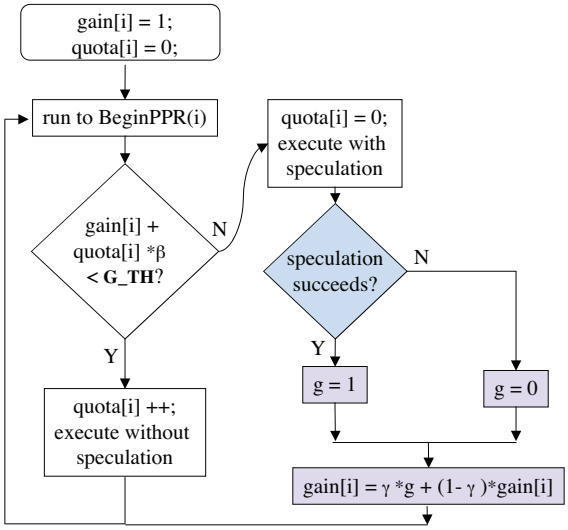


Figure 5. Decayed-history-based prediction

time. It uses three parameters to allow the separate control of different factors in the adaptive scheme.

In Figure 5, the array element $gain[i]$ records the exponentially decayed average of the speculation success rate of *PPR*$_i$. The **decay factor**, γ, is greater than 0 but less than 1; the larger it is, the faster the influence of a past speculation decays. The element $quota[i]$ records the number of the instances of *PPR*$_i$ that have been executed without speculation since the last failed speculation. The factor β is called the **aggressiveness factor**; the larger it is, the less quota is needed for resuming the speculative execution. The **gain threshold**, *G_TH* ($0 \leq$*G_TH*≤ 1), determines the tolerance of the adaptive scheme to speculation failures; the larger it is, the more likely occasional failures of speculative execution will disable the speculation of the next few *PPR* instances.

As showed in the flow chart, both a successful speculation and a non-speculative execution of a *PPR* instance contribute to the sum that triggers the next speculation (the white diamond box in Figure 5). The contribution from a successful speculation ranges from $\gamma * (1-$*G_TH*$)$ to γ, whereas, a skipped speculation contributes β. Intuitively, a non-speculative execution should not be more encouraging for resuming speculation than a successful speculation. Therefore, β should usually be smaller than $\gamma * (1-$*G_TH*$)$ and greater than 0.

This algorithm extends the first algorithm in three aspects. First, the number of consecutive disabled speculation is bounded by *G_TH*$/\beta$, whereas, Algorithm 1 has no such bound. As a result, the exponentially increased number of disabled speculations may cause Algorithm 1 to miss a whole profitable phase that follows a long unprofitable phase (i.e., pattern 2 followed by pattern 1 in Figure 6). Second, Algorithm 2 learns from not only recent failures but also the long-term speculation success rate. In Algorithm 1, if a speculation succeeds, the next instance of the *PPR* will definitely be speculated; if fails, it will be definitely skipped. But in this algorithm, a success may not be enough to trigger a speculation if it is a rare success after many failed trials; similarly, a rare failure may not necessarily lead to a skip of speculation. So, this algorithm may better tolerate occasional abnormal events. Meanwhile, the use of decayed average helps the algorithm respond quickly to phase changes. The third difference between the two algorithms is that, in Algorithm 2, the aggressiveness in trying a speculation is a tunable factor (β) separated from the tolerance to speculation failures (*G_TH*), whereas, in Algorithm 1, the aggressiveness is mingled with the adjustment of the speculation bar.

Adaptive Speculation for Function Parallelism A *PPR* of function parallelism may have only one instance in a whole execution of the program. The two algorithms described above are not applicable in this scenario, since both rely on previous instances of a *PPR* for adaptation. Cross-run learning is a promising solution for this case. It is to learn the relation between program inputs and the profitability of the *PPR* in the past runs. The detailed study is out of the scope of this paper.

4 Evaluation

This section first reports the accuracy of the adaptive algorithms on predicting the profitability of 6 typical *PPR* sequences, and then shows the savings of computation cost by the adaptive *BOP* system on some real programs.

4.1 Prediction Accuracy

To comprehensively evaluate the two adaptive schemes, we test the algorithms on a series of profitability patterns. Figure 6 contains three typical patterns. The first two patterns correspond to the scenarios in which either unprofitable or profitable speculations are rare; the third pattern is when each *PPR* instance has 50% chance to be profitable. The pure randomness of the third pattern determines that no algorithms can predict the profitability of its instances with an accuracy consistently higher than 50%. Therefore, we focus on the first and the second patterns. In addition, we use their combined patterns to test the capability of the adaptive algorithms in handling pattern changes. A combined pattern consists of a number of equal-length subsequences of the two basic patterns, which interleave with each other. For each (basic or combined) pattern, we create a random sequence composed of 200,000 elements. An element is either 0 or

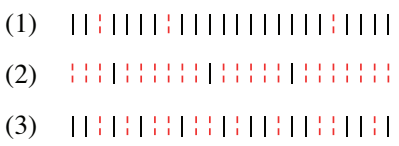


Figure 6. Basic patterns of speculation profitability. (|: profitable; ¦: non-profitable.)

```
for (γ=0.05; γ < 1; γ+=0.05){
 for (G_TH = 0.05; G_TH <= 0.9; G_TH += 0.05){
  for(k = 1; k < 100; k += 10){
   β = γ*(1-G_TH)/k;
   ...  ...}}}
```

Figure 7. Generation of the configurations included in Figure 8 (b).

1, respectively standing for a profitable and unprofitable *PPR* instance. In our test set, there are 2 basic patterns, respectively containing 90% and 10% profitable instances, and 4 combined patterns, consisting of many same-length subsequences of the two basic patterns; the length of a subsequence in the 4 combined patterns is 10^2, 10^3, 10^4, and 10^5 respectively.
The formula to compute prediction accuracy is as follows:

$$accuracy = \frac{|Spec \cap Prof| + |\overline{Spec} \cap \overline{Prof}|}{|Prof \cup \overline{Prof}|},$$

where, *Spec* is the set of *PPR* instances that are executed speculatively, and *Prof* is the set of *PPR* instances that are profitable to be speculatively executed. Thus, the denominator in the criterion is the total number of opportunities for (profitable or non-profitable) speculation; the numerator is the number of correct predictions for both profitable and non-profitable *PPR* instances.

4.1.1 Algorithm 1

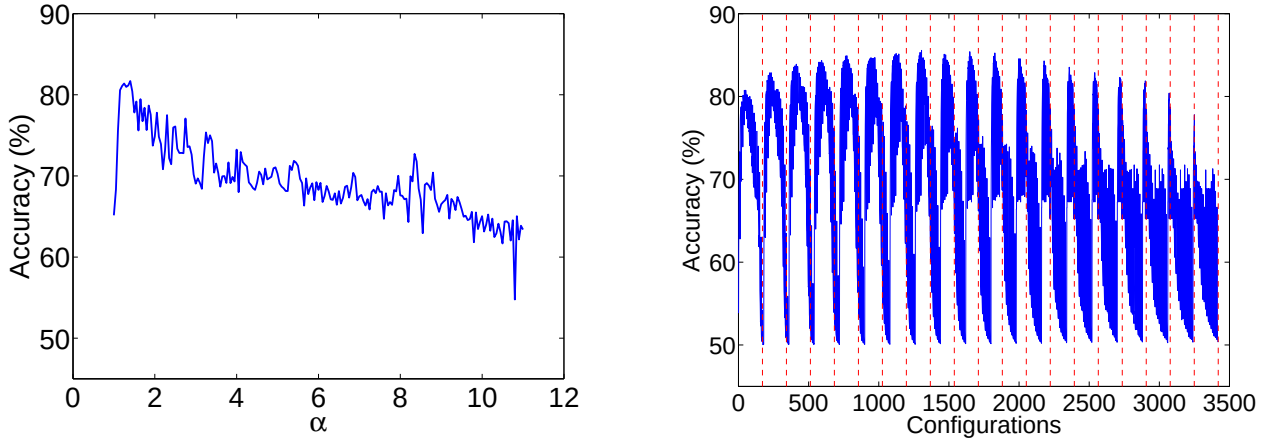
Figure 8 (a) depicts the prediction accuracies—averaged over all 6 patterns—of the first algorithm under different configurations. Recall that the algorithm contains only one parameter, the bar-raising factor, α. We experimented 200 values of α in the range between 1 and 11. The results show that the average accuracy increases from 65.1% to 81.7% when α increases from 1 to 1.4, and then decreases to 54.8% as α increases furthermore. The configuration, $\alpha = 1.4$, is the best among all, named as *globally-chosen configuration*, corresponding to the acc_g in Table 1.
The second column of Table 1 shows the accuracies of the algorithm on each pattern when the globally-chosen configuration is used. On pattern 1, the prediction error is mainly due to the rigid property of the algorithm that every failed speculation will disable at least one speculation, even if the failure is a rare event. The results on the four combined patterns show that as the frequency of pattern changes increases, the accuracy decreases from 86% to 71.7% due to the exponentially increased spec threshold in the algorithm.
The third column in Table 1 reports the distance between acc_g and acc_l, the accuracies of the prediction when the algorithm uses the configurations best for each individual pattern. The less than 1.4% distance indicates that a single configuration, $\alpha = 1.4$, can make the algorithm well adapt to different patterns. The fourth column of Table 1 shows the best value of α for each pattern.

4.1.2 Algorithm 2

Figure 8 (b) contains the average prediction accuracy of the second algorithm under different configurations. There are 3420 configurations in the graph, the order of which is determined by the nested loop showed in Figure 7.

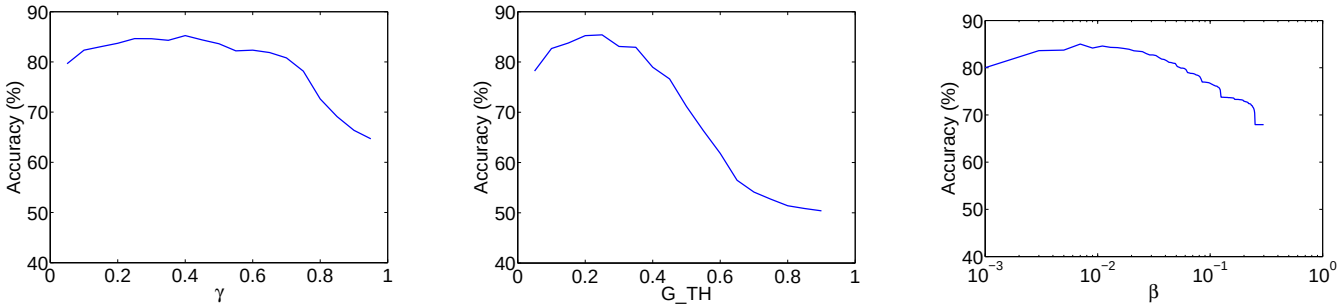
The configuration, ($\gamma = 0.4$, *G_TH*$= 0.25$, $\beta = 0.0073$), produces the most accurate prediction, 85.6% on average. The fifth column in Table 1 shows the accuracy on each of the 6 patterns when this configuration is used. The first combined pattern shows the lowest accuracy, 77.2%, due to the frequent pattern changes. All other patterns show accuracies higher than 86.2%. The third column reports the difference between acc_g and acc_l. The no greater than 3.2% difference shows that a single configuration is enough for the algorithm to achieve its best prediction for different patterns.

Figure 9 illustrates the influence of each of the three factors in Algorithm 2. We use ($\gamma = 0.4$, *G_TH*$= 0.25$, $\beta = 0.0073$) as the seminal configuration, and change one factor each time to obtain the three graphs. Figure 9 (a) shows that the increase of the decay factor γ improves prediction accuracy first and then decreases it, indicating the usefulness of both short-term and long-term history. The *G_TH* has more significant influence on the prediction accuracy as shown in Figure 9 (b). The accuracy drops from 82.9% to 56.5% sharply when *G_TH* is greater than 0.35. The reason is that it takes too long for a speculation to resume after a failed speculation when *G_TH* is large given that the β value is only 0.0073. The arch-shape curve in Figure 9 (c) reveals the tradeoff caused by the aggressiveness factor β. When it is too small, the algorithm cannot adapt quickly enough to pattern changes; when it is too large, it causes too many failed speculations.



(a) Algorithm 1 (b) Algorithm 2

Figure 8. Prediction accuracy of adaptive algorithms. On (b), the configurations in a panel (i.e., the area between two adjacent vertical lines) have the same value of the decay factor; Figure 7 shows the complete order of the configurations.



(a) Decay factor (b) Gain threshold (c) Aggressiveness factor

Figure 9. The effect of each individual parameter in Algorithm 2. The seminal configuration is as follows: $\gamma = 0.4$,*G_TH*$= 0.25, \beta = 0.0073$**.**

Table 1. Profitability prediction accuracy*

	Algorithm 1			Algorithm 2		
Patterns	acc_g(%)	$acc_l - acc_g$(%)	α_l	acc_g(%)	$acc_l - acc_g$(%)	$(\gamma$,*G_TH*$, \beta)_l$
1	82.4	0	1.0	89.6	0.6	(0.25, 0.30, 0.0159)
2	89.5	0.5	10.0	86.8	3.2	(0.05, 0.90, 0.0001)
(1,2)-10^2	71.7	0.1	1.7	77.2	1.5	(0.85, 0.10, 0.0084)
(1,2)-10^3	78.4	0.4	1.3	86.2	0.2	(0.35, 0.25, 0.0085)
(1,2)-10^4	82.0	1.4	2.8	88.0	1.0	(0.20, 0.25, 0.0021)
(1,2)-10^5	86.0	0.2	1.7	88.2	1.8	(0.05, 0.65, 0.0002)
Average	81.7	0.4	–	86.0	1.6	–

$*$ acc_g is the accuracy using the globally-chosen configuration ($\alpha = 1.4$ for Algorithm 1; $\gamma = 0.4$,*G_TH*$= 0.25, \beta = 0.0073$ for Algorithm 2); acc_l is for the configurations best for each pattern, showed in columns 4 and 7.

4.1.3 Comparisons

The comparison between the two algorithms' results (when both use their globally-chosen configurations) shows that Algorithm 2 is superior than Algorithm 1, reducing the prediction error of Algorithm 1 by 23.5%. It outperforms Algorithm 1 on almost all patterns. The 41% error reduction on the first pattern is mainly due to its better tolerance to occasional speculation failures, an advantage of the use of the decayed accumulative gain. Unlike in Algorithm 1, a failed speculation in this algorithm won't cause the skip of speculation on the next instance of *PPR* if it is a rare case in the recent history.

Algorithm 2 also shows better robustness to pattern changes than Algorithm 1, reflected by the higher accuracies and less fluctuations on the combined patterns. This advantage mainly comes from the bounded number of consecutive skips and the separation of the different factors, enabling more flexible configuration of the adaptive algorithm.

The only pattern that Algorithm 2 underperforms Algorithm 1 is the second pattern, which contains 90% unprofitable *PPR* instances. The 2.7% difference between them is mainly due to the relatively more aggressive speculation by Algorithm 2.

Overall, Algorithm 2 is a better choice than Algorithm 1. So, the next section concentrates on the evaluation of Algorithm 2 on real programs.

4.2 Computation Efficiency

We integrate Algorithm 2 into *BOP* and conduct experiments on 3 programs to measure the effects of the adaptive scheme on program performance and cost savings. We use a dual-core Intel Pentium-D machine (3.4GHz) equipped with GCC 4.0.1.

As a coarse-grained parallelization tool, BOP requires the *PPR*s to have large granularity so that the parallelization benefits can offset the overhead in process creation and correctness protection. Following the original BOP study [3], we label the major computation loops in the programs as *PPR*s: the compression of a buffer for *Gzip*, the parsing of a set of sentences for *Parser*, and the reduction of a set of numbers for *Reduction*. Each of the regions contains a number of function invocations and loops.

4.2.1 *Gzip* v1.2.4 by J. Gailly

The first program we use is the *gzip* in the SPEC CPU2000 benchmark suite. The program takes one or more files as input and compresses them one by one using the Lempel-Ziv coding algorithm (LZ77). In the previous work, the program has been parallelized to compress a single file in parallel under the support of *BOP* [3]. Each *PPR* instance is the compression of a data buffer, whose size determines the parallelism granularity and thus the profitability of the speculations by *BOP*. In this experiment, we let the *gzip* compress an 84MB file. We change the buffer size to 1.6MB, 320KB, or 192KB to vary the *PPR* granularity.

Table 2. Efficiency comparison on *gzip**

	buffer size	1.6MB	320KB	192KB
accuracy	org-bop	0.96	0.52	0.00
	adapt-bop	1.00	0.58	0.91
cost (s)	seq	12.44	11.64	11.68
	org-bop	15.53	25.74	31.97
	adapt-bop	14.15	16.03	15.96
time (s)	seq	12.44	11.64	11.68
	org-bop	7.76	13.10	15.98
	adapt-bop	7.07	12.87	13.90

**seq* stands for sequential runs; *org-bop* stands for the original *BOP*; *adapt-bop* stands for adaptive *BOP*.

Table 2 shows the results. When the buffer size is 1.6MB, most *PPR* instances are profitable to be speculatively executed. Both the original and the adaptive *BOP* achieve almost fully correct prediction. They accelerate the program by 60% and 76% respectively. We take the total times of all the processors that work on the program as the cost metric. On this granularity, the adaptive *BOP* costs 8.8% less than the original *BOP*. As the buffer size becomes smaller, more *PPR* instances become unprofitable; the parallel *gzip* becomes slower than the sequential *gzip*. In this case, the adaptive *BOP* automatically disables most of the speculations, saving execution cost by 37.8–50.0% and accelerating the original parallel *gzip* by 1.7–15.0%.

4.2.2 Sleator-Temperley Link *Parser* v2.1

Table 3 shows the result on another SPEC CPU2000 benchmark, *parser*. This program is an English sentence parsing tool; a *PPR* is the parsing of a group of sentences. As the group size decreases from 50 to 2, more *PPR* instances become unprofitable. On the smallest granularity, the original *BOP* runs 5.2s slower than the sequential run because of speculation overhead, whereas, the adaptive *BOP* in this case removes most of the overhead by automatically disabling most speculations. The cost saving is as much as 61.8%. On the other hand, for large granularities, the adaptive scheme enables most of the speculations and accelerates the sequential run by 79.2%, a speedup similar to what the original *BOP* produces.

Table 3. Efficiency comparison on *parser*

sentences pe *PPR*		50	10	2
accuracy	org-bop	0.86	0.94	0.50
	adapt-bop	0.86	0.94	0.89
cost (s)	seq	12.33	12.40	12.41
	org-bop	12.97	18.96	35.16
	adapt-bop	12.83	18.87	14.21
time (s)	seq	12.33	12.40	12.41
	org-bop	6.99	9.48	17.58
	adapt-bop	6.88	9.43	12.33

Table 4. Efficiency comparison on *reduction*

dependence		0	10%	50%	90%
accuracy	org-bop	1.00	0.90	0.42	0.12
	adapt-bop	0.99	0.91	0.60	0.82
cost (s)	seq	16.54	16.94	16.99	17.18
	org-bop	24.81	26.45	33.66	38.29
	adapt-bop	17.29	17.64	18.00	18.56
time (s)	seq	16.54	16.94	16.99	17.18
	org-bop	12.40	13.23	16.83	19.15
	adapt-bop	12.50	13.07	17.23	17.98

4.2.3 *Reduction* with Uncertain Dependences

In both *gzip* and *parser*, most of the speculation failures arise because the parallelism granularity is too small to offset the speculation overhead. We use a reduction program to test the scenario where the main reason for speculation failures is the uncertain dependence violations. The reduction program initializes an array of 64 million integers, performs 3 *square-root* and 3 *fmod* operations on each integer, and then adds the results together. During the computation, the program may or may not reset a global flag depending on the resolution of a conditional branch. The parallel version adds the numbers in blocks; each block is a *PPR*. The computation on two blocks may have dependences on each other if the former one resets the flag. We control the frequency of the occurrence of the dependence to 0, 10%, 50% (pattern 1 followed by pattern 2), and 90%. Table 4 reports the result. The adaptive scheme keeps the 38% speedup by *BOP* for the cases with few dependences, and meanwhile, saves up to 51.5% cost when there are 90% dependences.

Summary Overall, the adaptive scheme helps *BOP* reduce the execution cost significantly, especially when an execution contains many unprofitable *PPR* instances. Meanwhile, the adaptive scheme produces speedup similar to that by the original *BOP*; in most cases, the performance even becomes slightly better.

5 Related Work

Automatic loop-level software speculation is pioneered by the lazy privatizing *doall* (LPD) test [8]. Later techniques speculatively privatize shared arrays (to allow for false dependences) and combine the marking and checking phases (to guarantee progress) [1, 2, 4]. Two programmable systems are developed in recent years: *safe future* in Java [15] and *ordered transactions* in X10 [14]. The first is designed for (type-safe) Java programs, whereas *PPR* supports unsafe languages with unconstrained control flows. Ordered transactions rely on hardware transactional memory support for efficiency and correctness. Hardware-based thread-level speculation relies on hardware extensions for bookkeeping and rollback, having limited speculation granularity [11].

There have been some explorations on selecting program regions for parallelization or speculation. Static methods use heuristics from program code [13] or special graph-partitioning algorithms [6]. Profiling-based techniques employ dynamic information observed in the training runs and empirically search for the best regions [3, 7]. The speculations in all those systems are definite, in the sense that once a region is selected (by users or profiling tools) as a candidate for speculative execution, it will always be speculatively executed. This current work makes software speculative parallelization adaptive: Some instances of a region will be speculatively executed, but some others will not, depending on their likelihood to be beneficial or not. The adaptivity is important as the profitability of a region is input-sensitive. Hardware-based techniques dynamically inspect instruction streams and other runtime patterns to select regions for speculation; they are hard to benefit from high-level knowledge on program structures [12].

Many learning techniques exist in the realm of Statistical Learning [5]. However, we are not aware of any incremental learning techniques that can directly deal with incomplete history information like the *PPR* profitability in this current work. Last-value-based prediction has been commonly used in both hardware, such as dynamic branch prediction, and software, such as phase prediction (e.g., [9, 10]). Our work (the first algorithm) extends last-value-based prediction and makes it amendable for learning from partial information. The second algorithm in our work is novel in effectively combining long-term history with short-term experience.

6 Conclusion

This paper describes two adaptive speculation algorithms. Based on execution history, the adaptive schemes dynamically predict the profitability of a speculation and

disable the unprofitable speculations. Experiments demonstrate that the two algorithms can produce accurate prediction for different profitability patterns. Especially, the decayed-history-based algorithm offers more configuration flexibility and produces more accurate prediction. Experiment results show the promise of the adaptive speculation in improving both the efficiency and performance of behavior-oriented parallelization.

Acknoledgement We thank Chen Ding, Rudolf Eigenmann, and the anonymous reviewers of ICPP08 for all the insights and comments that have helped the improvement of this work. This material is based upon work supported by the National Science Foundation under Grant No. 0720499. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. H. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[2] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. Technical report, CS Dept., Texas A&M University, College Station, TX, 2002.

[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior-oriented parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, San Diego, USA, 2007.

[4] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. In *Proceedings of SC'98*, 1998.

[5] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[6] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, 2004.

[7] T. A. Jonhson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, March 2007.

[8] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[9] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[12] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 1998.

[13] T. Vijaykumar and G. Sohi. Task selection for a multiscalar processor. In *Proceedings of the International Symposium on Microarchitecture*, December 1998.

[14] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, March 2007.

[15] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for java. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 439–453, 2005.