Adaptive Speculation in 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

Behavior-oriented parallelization is a technique for parallelizing complex sequential programs that have dynamic parallelism. Although the technique shows promising results, the software speculation mechanism it uses is not cost-efficient. Failed speculations may waste computing resource and severely degrade system efficiency. In this work, we propose adaptive speculation to predict the profitability of a speculation and dynamically enable or disable the speculation of a region. Experimental results demonstrate the effectiveness of the scheme in improving the efficiency of software speculation. In addition, the adaptive speculation can also enhance the usability of behavior-oriented parallelization by allowing users to label potential parallel regions more flexibly.

1 Introduction

It is often challenging to parallelize a sequential program with dynamic high-level parallelism, due to the code complexity, input-dependent behavior, and the uncertainty in parallelism. We recently proposed *behavior-oriented parallelization* (*BOP*) [3] to address 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 high overhead and the uncertain dependences, however, may make a speculation useless, in which case, the speculation forms a waste of computing resource. This waste can degrade system efficiency severely when the input to the program causes dependences for many instances of the speculated regions. The current *BOP* system blindly speculates every *possibly parallel region* (*PPR*), thus is cost-inefficient.

To reduce the waste caused by failed speculations, this work proposes adaptive speculation. The goal is to make *BOP* able to conduct only profitable speculations and automatically disable unprofitable ones. Besides improving computing efficiency, adaptive speculation provides *BOP* users greater flexibility in labeling *PPR*s. A user will be able to label more regions as *PPR*s as the adaptive scheme can automatically select profitable ones for speculation.

It is challenging to predict speculation profitability through program code analysis because the profitability depends on program inputs and runtime behavior. This work treats the problem as a statistical learning task. We develop an adaptive algorithm that recognizes the profitability patterns of a *PPR* by online learning from the previous instances of the *PPR*. A complexity in the learning is that the profitability of the earlier instances are not always unveiled: If a *PPR* instance is not executed speculatively, *BOP* cannot determine its profitability. The algorithm manages to learn from the partial information and adapt to the dynamic changes in profitability patterns. It yields over 86% prediction accuracy for five of six *PPR* patterns and saves up to 162% computation cost compared to the original *BOP* system.

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.

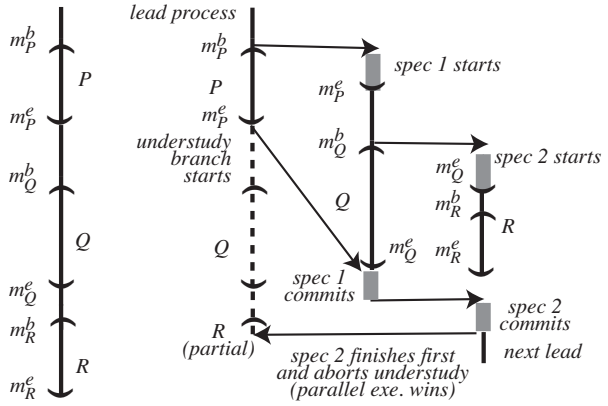
Figure 3 illustrates the run-time setup. Part (a) shows the sequential execution of three *PPR* instances, P, Q, and R. Part (b) shows the speculative execution. The execution starts as the *lead* process. When the lead process reaches the start marker of P, m_P^b, it forks the first speculative process, *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

```
...
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 three *PPR* instances

second speculative process, *spec 2*, which jumps ahead to execute from the end of Q.

At the end of P, the lead process starts an *understudy* process, which reexecutes the following code *non*-speculatively. The lead process itself then waits for spec 1 to finish, checks for conflicts; if no conflict is detected, it commits its changes to spec 1, which assumes the role of the lead process so 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. *BOP* employs a set of techniques to efficiently detect conflicts for execution correctness [3].

One of the important features of *BOP* is that the understudy and the speculative processes execute the same *PPR* instances. The understudy's execution is part of a sequential execution of the program and thus is absolutely 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 guarantees the basic efficiency of the program. In the latter case, the speculative processes abort, and their execution becomes a waste of computing resource.

3 Adaptive Speculation

To avoid the waste caused by failed speculative executions, we develop a statistical predictor that learns from prior instances of a *PPR* and predicts the profitability of its future instances. The output of the predictor is a binary value, indicating whether a *PPR* instance is profitable to be speculatively executed.

This prediction task resembles branch prediction, but differs in two aspects. First, in branch prediction, the correctness of a prediction can always be determined immediately after the branch is resolved, whereas, in adaptive speculation, the profitability of a *PPR* instance cannot be uncovered unless the instance is speculatively executed. So, if a *PPR* instance is predicted as not beneficial and is not speculatively executed, the correctness of the prediction will remain unknown. The adaptive speculation, therefore, has to learn from the partial information. The second difference is that branch prediction is usually implemented on hardware with strict constraints on both space and time, whereas, adaptive speculation is a software scheme, permitting more sophisticated algorithms.

The adaptive algorithm to be presented focuses on loop parallelism; it learns from prior iterations of a loop *PPR*. For the purpose of clarity, the following description assumes the speculative depth to be 1—that is, there is at most one speculation process at any time.

Figure 4 shows the adaptive algorithm. In this algorithm, the array element $gain[i]$ records the exponentially decayed average of the speculation success rate of PPR_i. The *decay factor* is γ ($0 \leq \gamma \leq 1$); the higher 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 previous failed speculation. The factor β is the *aggressiveness factor*; the higher it is, the less quota is needed for resuming the speculative execution. The *gain threshold*, *G_TH* ($0 \leq G\_TH \leq 1$), determines the tolerance of the adaptive scheme to speculation failures; the higher it is, the more likely occasional failures of speculative execution will disable the speculation of the next several *PPR* instances.

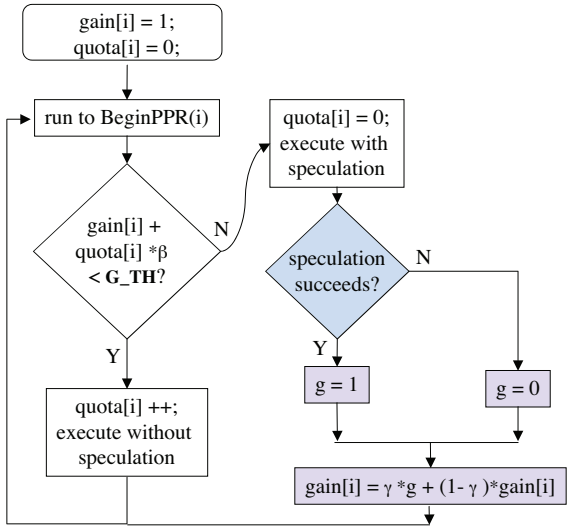


Figure 4. Adaptive algorithm

In the algorithm, both a successful speculation and a non-speculative execution (also called a skipped speculation) of a *PPR* instance contribute to the sum that triggers the next speculation (the white diamond box in Figure 4). The contribution from a successful speculation ranges from $\gamma * (1 - G\_TH)$ to γ, and a skipped speculation contributes β. Intuitively, a non-speculative execution should not be more encouraging than a successful speculation. Therefore, β should usually be smaller than $\gamma * (1 - G\_TH)$ (and greater than 0).

The adaptive algorithm has three important features. First, the number of consecutive skips is limited by a constant upperbound, G_TH/β. This property prevents *BOP* from skipping a whole parallelizable phase that follows a long unprofitable phase (i.e., pattern 2 followed by pattern 1 in Figure 5). Second, the algorithm learns from not only recent failures but also long-term speculation success rate. This property is useful for the algorithm to tolerate occasional abnormal events. Meanwhile, the use of decay helps the algorithm respond quickly to phase changes. The third property is that the aggressiveness in trying a speculation is a tunable factor (β) separated from the tolerance to speculation failures (G_TH), enabling more flexible control.

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

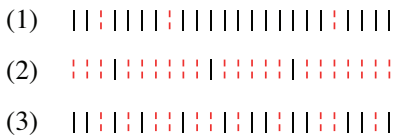


Figure 5. Basic patterns of speculation profitability. (|: profitable; ¦: non-profitable.)

4 Evaluation

This section first presents the accuracy of profitability prediction of the adaptive algorithm, and then reports the improvement of the computation efficiency of *BOP* system.

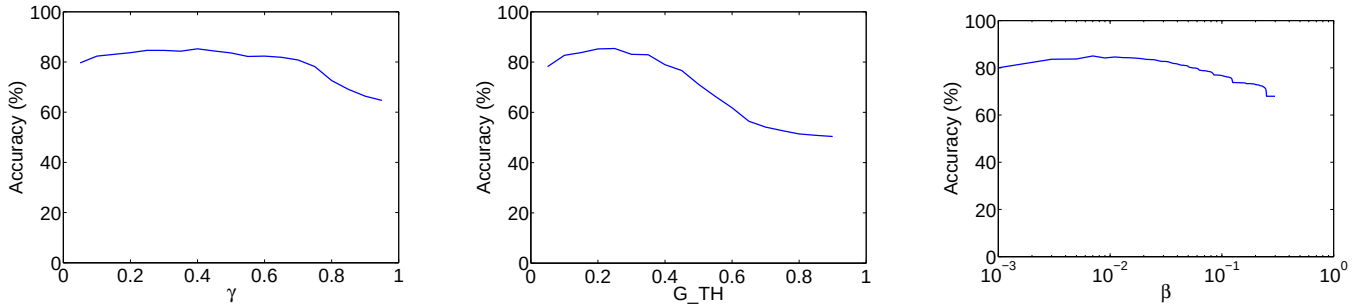
4.1 Prediction Accuracy

To comprehensively evaluate the adaptive schemes, we test the algorithm on a series of profitability patterns. The three basic patterns are showed in Figure 5. The first pattern is when unprofitable speculations are rare; the second is when the profitable speculations are rare; the third is when unprofitable and profitable speculations are evenly mixed. The pure randomness of the third pattern determines that no algorithms can predict the instances in the pattern 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 algorithm in handling pattern changes. A combined pattern consists of a number of equal-length subsequences of the two basic patterns that interleave with each other. For each (basic or combined) pattern, we create a biased random sequence composed of 200,000 elements. An element is either 0 or 1, respectively standing for *PPR* instances that are profitable or unprofitable to be executed speculatively. The two basic patterns respectively contain 10% and 90% dependences. We use 4 combined patterns; the length of a subsequence in them is respectively 10, 100, 1000, and 10000.

The prediction accuracy is defined 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 speculated. Thus, the denominator in the criterion is the total number of opportunities for (successful and non-successful) speculation, which is equal to half the total number of *PPR* instances; the numerator is the number of correct predictions for both profitable and non-profitable *PPR* instances.



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

Figure 8. The effect of each individual parameter in the adaptive algorithm. The seminal configuration is as follows: $\gamma = 0.4, G\_TH = 0.25, \beta = 0.0073$**.**

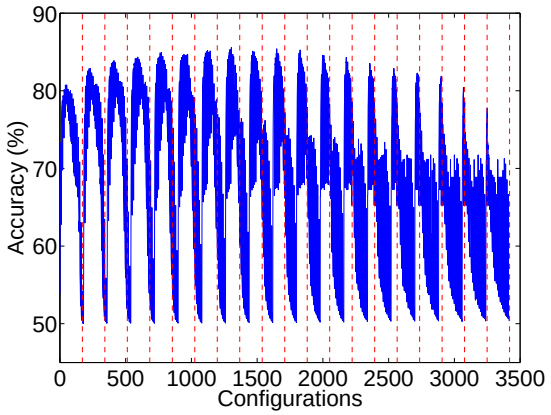


Figure 6. Prediction accuracies using different configurations. The configurations in a panel (i.e., the area between two adjacent vertical lines) have the same value of γ. The order of the configurations follows the loops in Figure 7.

```
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 6.

Table 1. Profitability prediction accuracy*

Patterns	acc_g(%)	$acc_l - acc_g$(%)	$(\gamma, G\_TH, \beta)_l$
1	89.6	0.6	(0.25, 0.3, 0.0159)
2	86.8	3.2	(0.05, 0.9, 0.0001)
(1,2)-10^2	77.2	1.5	(0.85, 0.1, 0.0084)
(1,2)-10^3	86.2	0.2	(0.35, 0.25, 0.0085)
(1,2)-10^4	88.0	1.0	(0.2, 0.25, 0.0021)
(1,2)-10^5	88.2	1.8	(0.05, 0.65, 0.0002)

$*$ acc_g is the accuracy using the globally-chosen configuration; acc_l is for the locally-chosen configuration; $(\gamma, G\_TH, \beta)_l$ shows the corresponding configurations.

Figure 6 contains the prediction accuracy of the adaptive algorithm using different configurations; an accuracy is the average value over all 6 patterns presented earlier in this section. 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, an average accuracy of 85.6%. The second column in Table 1 shows the accuracy when using this configuration (denoted by acc_g, g for *g*lobal) on each of the 6 patterns. 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 of acc_g from the accuracies produced by the best configuration for each individual pattern, acc_l (l for *l*ocal). The small difference demonstrates that the adaptive algorithm is able to handle different profitability patterns by using a single configuration.

Figure 8 illustrates the influence of the three factors. 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. The increase of the decay factor improves pre-

Table 2. Efficiency comparison on *parser*

sentences pe *PPR*		2	5	10	50
acc	adapt-bop	0.89	0.88	0.94	0.86
	org-bop	0.50	0.88	0.94	0.86
cost (s)	seq	12.40	12.27	12.40	12.33
	adapt-bop	20.12	38.60	28.30	17.64
	org-bop	52.75	38.73	28.45	17.96
time (s)	seq	12.29	12.27	12.27	12.33
	adapt-bop	12.33	12.87	9.43	6.88
	org-bop	17.58	12.91	9.48	6.99

diction accuracy first and then decreases it, indicating the tradeoff between the usefulness of long-term and shor-term history information. The aggressiveness factor shows the similar influence trend. Compared to those two factors, the gain threshold has more significant influence on the prediction accuracy.

4.2 Computation Efficiency

Table 2 shows the comparison between the original and the adaptive *BOP*, represented by *org-bop* and *adapt-bop* respectively. We use a dual-core Intel Pentium-D machine (3.4GHz). We choose *parser* in SPEC CPU2000 as the benchmark because the workload size of a *PPR* in *parser* is easy to control. We test on 4 different *PPR* granularities; the larger a *PPR* is, the easier it is for speculation benefits to offset the overhead (no dependences exist between *PPR*s). For the smallest granularity, org-bop runs 5.2s slower than the sequential run, whereas, adapt-bop removes most of the overhead by automatically disabling most of the speculations. On the other hand, for large granularities, adapt-bop enables most of the speculations and accelerates the sequential run by up to 79.2%, a speedup similar to what org-bop produces. The "cost" data in the table are the total times of all the processors that work on the program. The adaptive scheme saves up to 162% of the org-bop's cost by improving profitability prediction accuracy to over 86%—the "acc" data in the table.

5 Related Work

Automatic loop-level software speculation is pioneered by the lazy privatizing *doall* (LPD) test [5]. 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 [8] and *ordered transactions* in X10 [7]. 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 [6].

BOP system is unique in that the speculation overhead is proportional to the size of program data rather than to the frequency of data access. *BOP* is the first to speculatively privatize the entire address space and apply speculative execution beyond the traditional loop-level parallelism.

The speculation in prior systems is 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 work makes software speculative parallelization adaptive by predicting the profitability of a speculative execution.

6 Conclusion

This paper proposes adaptive speculation for *BOP*. Based on execution history, the adaptive scheme dynamically predicts the profitability of a speculation and disables the speculations that are unlikely profitable. Experiments demonstrate that the algorithm can produce accurate prediction for different profitability patterns, which improves system efficiency significantly.

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] 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.

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

[7] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. March 2007.

[8] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for java. pages 439–453, 2005.