Speculation with Little Wasting: Saving Cost in Software Speculation through Transparent Learning

Yunlian Jiang Feng Mao Xipeng Shen

Department of Computer Science
The College of William and Mary, Williamsburg, VA, USA 23185
{*jiang,fmao,xshen*}*@cs.wm.edu*

***Abstract*—Software speculation has shown promise in parallelizing programs with coarse-grained dynamic parallelism. However, most speculation systems use offline profiling for the selection of speculative regions. The mismatch with the input-sensitivity of dynamic parallelism may result in large numbers of speculation failures in many applications. Although with certain protection, the failed speculations may not hurt the basic efficiency of the application, the wasted computing resource (e.g. CPU time and power consumption) may severely degrade system throughput and efficiency. The importance of this issue continuously increases with the advent of multicore and parallelization in portable devices and multiprogramming environments.**

In this work, we propose the use of *transparent* statistical learning to make speculation cross-input adaptive. Across *production* runs of an application, the technique recognizes the patterns of the profitability of the speculative regions in the application and the relation between the profitability and program inputs. On a new run, the profitability of the regions are predicted accordingly and the speculations are switched on and off adaptively. The technique differs from previous techniques in that it requires no explicit training, but is able to adapt to changes in program inputs. It is applicable to both loop-level and function-level parallelism by learning across iterations and executions, permitting arbitrary depth of speculations. Its implementation in a recent software speculation system, namely the Behavior-Oriented Parallelization system, shows substantial reduction of speculation cost with negligible decrease (sometimes, considerable increase) of parallel execution performance.

***Index Terms*—Adaptive Speculation, Behavior-Oriented Parallelization, Multicore, Transparent Learning**

I. Introduction

Recent years have seen a rapid shift of processor technology to favor chip multiprocessors. Many existing programs, however, cannot fully utilize all CPUs in a system yet, even though dynamic high-level parallelism exists in those programs. Examples include a compression tool processing data buffer by buffer, an English parser parsing sentence by sentence, and an interpreter interpreting expression by expression, and so on. These programs are complex and may make extensive use of bit-level operations, unrestricted pointers, exception handling, custom memory management, and third-party libraries. The unknown data access and control flow make such applications difficult if not impossible to parallelize in a fully automatic manner. On the other hand, manual parallelization is a daunting task for complex programs, especially for those pre-existing ones. Moreover, the complexity and the uncertain performance gain due to input-dependence make it difficult to justify the investment of time and the risk of errors of the manual efforts.

Software speculation has recently shown promising results in parallelizing such programs [4], [18]. The basic idea is to dynamically create multiple speculative processes (or threads), which each skips part of the program and speculatively executes the next part. As those processes run simultaneously with the main process, their successes shorten the execution time.

But speculative executions may fail because of dependence violations or being too slow to be profitable (elaborated in Section II.) In systems with no need for rollback upon speculation failures—such as the *behavior-oriented parallelization* (*BOP*) system [4], failed speculations result in the waste of computing resources (e.g., CPU and memory) and hence inferior computing efficiency. The waste is a serious concern especially for multi-programming or power-constrained environments (e.g., laptops, embedded systems.) For systems where rollback is necessary, an additional consequence is the degradation of program performance.

Therefore, the avoidance of speculation failures is important for the cost efficiency of modern machines. Previous studies—mostly in thread-level speculation—have tried to tackle this problem through profiling-based techniques (e.g., [6], [10], [19].) The main idea is to determine the regions in a program that are most beneficial for speculation by profiling some training runs.

The strategy, however, is often insufficient for coarse-grained software speculation, because of the *input-sensitive* and *dynamic* properties of the parallelism. In a typical application handled by software speculation, the profitability (i.e., likelihood to succeed) of a speculative region often differs among executions on different program inputs, or even among different phases of a single execution. The profiling-based region selection can help, but unfortunately, is not enough for software speculation to adapt to the changes in program inputs and phases.

In a recent work [9], we proposed adaptive speculation to address the limitations of the profiling-based strategy. The idea is to predict the profitability of every *instance* of a speculative region online, and adapts the speculation accordingly. It departs from previous profiling-based techniques, which, after selecting the speculative region (e.g., a loop), typically speculate *every* instance (e.g., an iteration of a loop) of the region with no runtime adaptation. The proposed adaptive speculation has successfully avoided unprofitable instances of speculative regions, and improved cost efficiency evidently [9].

However, two limitations of the technique severely impair its applicability and scalability. *First*, it can only handle loop-level (case *a* in Figure 1) but not function-level speculations (case *b* in Figure 1.) This limitation is inherent to the runtime adaptation algorithm in the technique: Its prediction is based on previous instances of a speculation region in the current execution, whereas, in function-level speculation, the region often have only few invocations in the entire execution, making the prediction difficult. Such regions, on the other hand, often compose a major portion of the applications that rely on function-level parallelism. Thus, failed speculations are especially costly for those applications.

In this work, we propose a *cross-run* transparent learning scheme to address the challenges facing the adaption of function-level speculations. The new scheme differs from the previous technique [9] substantially: Rather than adapt purely upon history instances, it uses incremental machine learning techniques (Classification Trees) to uncover the relation between program inputs and the profitability patterns of each speculation region. With the new scheme, the profitability of speculation regions can be predicted as soon as an execution starts on arbitrary inputs. Developing such a scheme has to overcome two difficulties. First, program inputs can be very complex with many options and attributes. The scheme has to extract the features that are important to the profitability of a region. Our approach employs an input characterization technique to address that problem. The second challenge is to predict with confidence. Because our approach builds the predictive model incrementally, it is important to determine whether the current model is accurate enough for use. Our design addresses the problem by integrating self-evaluation into the learning process.

The *second* limitation of the previously proposed adaptive speculation is that it is not scalable: The speculation depth (i.e., the number of speculative processes per region) can be at most one. The reason for the scalability comes from the complexity in the support of deeper speculations in the speculation system (*BOP*.) In this work, we extend the algorithm and implementation to allow arbitrary increment and decrement of speculation levels.

The elimination of the two limitations constitutes the two major contributions of this work. The first component, the support for adaptive function-level speculations, is substantially novel compared to the previous technique [9] as described earlier. The second component, the scalability extension, may seem relatively incremental; however, as described in Section III-B, this component has taken us substantial efforts as well, due to the implementation complexities in the speculation system, and the difficulties in determining the appropriate parameter values for the algorithm to achieve a good tradeoff between computing efficiency and cost savings.

```
...                      ...
while (1) {              BeginPPR(1);
 get_work();             work(x);
 ...                     EndPPR(1);
 BeginPPR(1);            ...
 work();                 BeginPPR(2);
 EndPPR(1);              work(y);
 ...                     EndPPR(2);
}                        ...
(a) loop-level           (b) function-level
```

Fig. 1. The speculation unit in *BOP* is a possibly parallel region (PPR), labeled by *BeginPPR(p)* and *EndPPR(p)*. The two examples illustrate the two kinds of PPRs *BOP* handles.

We implement both techniques in *BOP* [4], a recent software speculation system. Evaluations on a chip multiprocessor machine demonstrate that the proposed techniques are effective in preventing unprofitable speculations without sacrificing profitable ones. The techniques help *BOP* save a significant amount of cost, and meanwhile, cause little decrease but often increase to the program performance. The cost efficiency is enhanced significantly.

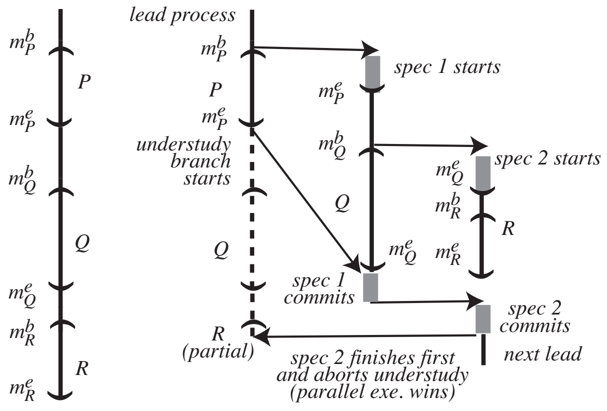
In the rest of this paper, Section II gives a brief review of the *BOP* system. Section III-A describes the transparent learning for function-level speculation. Section III-B presents the algorithm for scalable loop-level adaptive speculation. Section IV reports evaluation results, followed by a discussion on related work and a short summary.

II. Review on BOP

BOP creates one or more speculative processes to execute some *PPR* instances in parallel with the execution of a lead process. Figure 2 illustrates the run-time mechanism. The left part of the figure 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). The right part shows the parallel execution enabled by *BOP*. The execution starts with a single process, named *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 with the speculative execution by the spec processes. After creating the understudy process, the lead process falls into sleep. When spec 1 reaches m_P^e, the lead process is waked up and helps spec 1 to check for dependence violations. If there are no violations, the lead process commits its changes to spec 1 and aborts itself. Spec 1 then assumes the role of the lead process; the later speculation processes are handled in a similar manner. The kth spec is checked and combined after the first $k-1$ spec processes commit.

One special feature of *BOP* is that the understudy and the speculative processes execute the same *PPR* instances. The understudy's execution starts later but is guaranteed to be correct;



(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.

Fig. 2. An illustration of the sequential and the speculative execution of 3 *PPR* instances

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 and correctly, the parallel run wins and the parallelism is successfully exploited; otherwise, the understudy's run wins and the speculative execution becomes a waste of computing resource. This race ensures that the execution supported by *BOP* won't be much slower than a sequential run.

The understudy enables another special feature of *BOP*: no rollback is required upon a speculation failure. Therefore, the main benefits of avoiding failed speculations for *BOP* is the enhancement of cost efficiency, rather than program performance. But for most other systems, performance improvement will be one of the major benefits of preventing speculation failures.

The bottom line is that the speculative processes may abort for two reasons: Their execution either violates certain data dependences or is just too slow to beat the sequential run by the understudy process. The occurrence of both reasons may depend on the inputs to the application and the phase changes in an application. Avoiding those failures can make *BOP* more cost-efficient.

III. Adaptive Speculation Algorithms

The goal of adaptive speculation is to speculatively execute only the *PPR* instances that are profitable. In *BOP*, a profitable *PPR* instance is the instance whose speculative execution will succeed. The success implies that the speculation contains no dependence violation, and meanwhile, runs fast enough to outperform the sequential execution.

The key to adaptive speculation is to accurately predict the profitability of every *PPR* instance. Our basic strategy is to recognize the profitability patterns of each *PPR* through runtime statistical learning or cross-run modeling.

For adaptive speculation to be effective, the profitability prediction must meet four requirements. First, it has to be *responsive*, able to adapt to the changes in program inputs, program phases, and running environments. Second, it must be *robust*, able to tolerate a certain degree of temporary fluctuation in program behaviors. The last-value-based predictor (using last time value for the prediction of the current time), for example, is responsive but not robust as it can be easily misled by temporary changes. Third, unlike many dynamic adaptation problems (e.g., branch prediction), the profitability prediction has to learn from *partial* information on previous *PPR* instances. If a previous *PPR* instance was predicted as unprofitable and was not speculatively executed, the correctness of the prediction will remain unknown. Finally, the prediction mechanism and the adaptive speculation scheme should support an arbitrary depth of speculation in order to be *scalable*.

In the following, we first present a cross-run learning algorithm for the handling of function-level *PPR*s, followed by a scalable way to treat loop-level *PPR*s. We implement both algorithms in *BOP* such that it automatically selects the right algorithm for a given *PPR* based on its type of parallelism.

A. Cross-Run Function-Level Adaptive Speculation

In function-level parallelism, a *PPR* tends to have few instances in a whole execution of the program. The small number of instances are often not enough for the previous adaptive scheme [9] to learn about the profitability patterns. Furthermore, the profitability of a *PPR* often depends on program inputs. Offline profiling-based techniques [6], [10], [19] are insufficient to adapt to the changes of program inputs.

1) Cross-Run Transparent Learning: Our solution is a transparent learner that incrementally builds a classification tree to recognize the relation between program inputs and the profitability of a *PPR*. The learning process is transparent, requiring no involvement from the program user or offline profiling runs.

The scheme works as follows. At the beginning of an execution, the runtime system (*BOP* in our experiment) converts the program input into a feature vector $\vec{v}$. During the execution, the runtime system records the success rate r of the speculations on each *PPR*. At the end of the execution, the runtime system converts each r into 0 if $r < 0.5$ or 1 if $r \geq 0.5$. After n real runs of the program, there will be a set of pairs for a *PPR*, $(\vec{v}_1, r_1), (\vec{v}_2, r_2), \cdots, (\vec{v}_n, r_n)$. A classification tree can then be built automatically from those data. On the next run of the program, the runtime system predicts the profitability of the *PPR*s in the new run by feeding the new input feature vector into the classification trees.

Implementing the scheme requires answers on how to extract the feature vector from a program input, how to learn from the date, and how to predict *confidently*.

2) Input Feature Extraction: Program inputs can be complex, including many options and hidden features. In this work, we employ a previously proposed technique, XICL-based input characterization [12], to resolve the complexity.

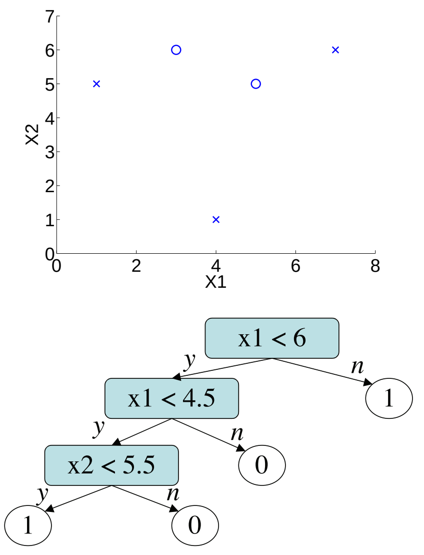


Fig. 3. A training dataset (o: class 1; x: class 2) and the classification tree.

XICL is an extensible input characterization language. It offers a systematic way for a programmer to specify the input format and potentially important features for a program. With the specification, XICL translator automatically converts an arbitrary input into a feature vector. The whole process is transparent to the user of the program (the XICL specification is provided by the programmer as part of the program.) Furthermore, the programmers may indicate more than necessary features because the learner can select the important features automatically. That gives programmers flexibility.

Automatic input characterization can also be used in this work. For example, Ding and Zhong use runtime sampled long reuse distance to characterize inputs [5]. We choose the XICL-based technique for its generality.

3) Classification Trees: We select classification trees as the learner to recognize the relation between the profitability of a *PPR* and input feature vectors. A classification tree is a hierarchical data structure implementing the divide-and-conquer strategy [8]. It divides the input space into local regions, each of which has a class label. Figure 3 shows such an example. Each non-leaf node asks a question on the inputs and each leaf node has a class label. The class of a new input equals to the leaf node it falls into.

The key in constructing a classification tree is in selecting the best questions in the non-leaf nodes. The goodness of a question is quantified by an impurity measure. A split is pure if after the split, the data in each subspace has the same class label. Many techniques have been developed to automatically select the questions based on the entropy theory [8].

Classification trees are easy to build, handle both discrete and numeric inputs, and have good interpretability. More importantly, because the questions in each node are selected automatically, the learning process is also a feature extraction process; the important features of program inputs are selected automatically. These properties make it a suitable choice for profitability prediction.

4) Discriminative Prediction: Given that the classification trees are built incrementally, we need a scheme to decide whether the learner is mature enough to start prediction. Our solution is a self-evaluation scheme. By measuring the confidence of the learner, it enables discriminative prediction—that is, only predicting when the learner is confident. This scheme is important for reducing the risk of poor predictions.

The self-evaluation scheme works in this way. Each classification tree has a confidence value. The runtime system uses the prediction of the learner only if the confidence value of the corresponding classification tree is higher than a threshold (70% in our experiment.) Initially, every confidence value is 0; the runtime system ignores the learner and conducts speculation for all *PPR*s. At each speculation of a *PPR*, the learner conducts a prediction and checks with the real profitability of the speculation. If the prediction is correct, the learner sets r to 1; otherwise, $r = 0$. Then it updates the confidence value of the corresponding classification tree in a decayed average formula:

$$confidence = \gamma * r + (1 - \gamma) * confidence,$$

where, γ is the decay factor with a value between 0 and 1 (0.7 in our experiments.)

B. Scalable Loop-Level Adaptive Speculation

In loop-level parallelism, a *PPR* often has many instances. The speculative execution on earlier instances offers the opportunities for an adaptive scheme to learn the profitability pattern of the *PPR* and thus, to predict the profitability of future instances. Previously, we have implemented an adaptive scheme to handle loop-level *PPR*s [9]. However, it can only support one level of speculation. In this section, we describe the design and implementation of our extended algorithm, which removes the scalability limitation.

The high-level design of the adaptive algorithm is simple. The initial speculation depth, denoted as d, of a *PPR* is set to its upperbound, which is usually the number of computing units in the system minus 1 (as there is a lead process.) During the parallel execution of the *PPR*, the adaptive scheme observes the gain brought by the d-level speculation. If the gain is too low, the scheme decreases the speculation depth unless d is already 0; if the gain is high enough, the scheme increases the depth unless d already reaches the upperbound.

But the concrete design is more complex. It has to address the following problems. First, the measurement of gains determines the degree of responsiveness and robustness of the prediction mechanism, both of which must be met in the measurement of the gains. Second, when the speculation depth decreases to 0, *BOP* stops speculation on the *PPR*, so the gain won't increase anymore. But the future instances of the *PPR* are still possible to be profitable, especially after a phase change. So the adaptive scheme ought to include an appropriate level of exploration for speculation even if the current gain is low.

```
/*
 $i$ : index of a speculative region;
 $d$ : the current speculation depth;
 $maxdep$ : the maximum speculation depth;
 $gain[i][d]$ : gains of the $d$-level speculation;
 $quota[i][d]$ : quota for triggering $d$-level speculation;
 $\beta$ : aggressive factor; $\gamma$ : decay factor;
 $G\_TH$ : gain threshold;
*/

/* Initial values */
d = $maxdep$;
$gain[i][maxdep]$=1; $quota[i][maxdep]$=0;

/* Before starting a speculation */
// decrease speculation depth if needed
if ($d > 0$ & $gain[i][d]$ + $quota[i][d] * \beta < G\_TH$){
  $d$ - - ;
  $quota[i][d]$ = 0;//reset quota
  $quota[i][d+1]$ = 0;
  $gain[i][d]$ = 1; //reset gain
}
$quota[i][d+1]++$;//quota increase for the next level

/* After one execution of the speculative region */
// update gains
$g$ = 0;
if (speculation succeeds) $g$=1;
for ($k$=0;$k < d$;$k$++) {
  $gain[i][d]$ = $\gamma * g + (1-\gamma) * gain[i][d]$;
}
// increase speculation depth if needed
if ($d < maxdep$ &
 $gain[i][d+1] + quota[i][d+1] * \beta \geq G\_TH$){
  $d$++;
  $quota[i][d+1] = 0$;
}
```

Fig. 4. Algorithm for loop-level adaptive speculation of arbitrary depth.

Figure 4 outlines the adaptive speculation algorithm. The three code sections respectively correspond to the initialization, and the code to be executed before and after a d-level speculation (normal execution if d is 0). In the algorithm, *gain[i][d]* is used to store the (decayed average) gain of the d-level speculation on *PPR*$_i$. When a *PPR* is executed on d-level speculation, we consider each d-level speculation as a potential chance for starting $d+1$-level speculation. We use *quota[i][d+1]* to record the number of such chances since the latest time that the speculation level of *PPR*$_i$ becomes d. The use of *quota[i][d]* is the key for the adaptive scheme to learn with only partial information of the previous *PPR* instances.

The values of *gain[i][d]* and *quota[i][d]* together determine whether the speculation depth should increase or decrease. The formula is as follows:

$$\text{weighted gain} = \text{gain[i][d]} + \beta * \text{quota[i][d]},$$

where, β is a parameter named the **aggressiveness factor** with a value between 0 and 1. So, when *quota[i][d]* becomes high enough, the adaptive system will try $(d+1)$-level speculation on *PPR*$_i$, even if *gain[i][d]* is not very high. This strategy encourages a certain degree of the exploration to higher level speculation, which is vital for resuming high-level speculation after the depth decreases to 0. The threshold for the increase and decrease is *G_TH*, named **gain threshold**.

The formula for updating *gain[i][d]* incorporates both the profit of the just-finished speculation and the information of all the previous speculations of the *PPR*. It is in a form of decayed average:

$$gain[i][d] = \gamma * g + (1-\gamma) * gain[i][d],$$

where, γ is the **decay factor** with a value between 0 and 1, and g is 1 if the just-finished speculation succeeds and 0 otherwise. On a d-level speculation, the *gain[i][d]* is updated d times using this formula because d instances of the *PPR* are just executed.

Like the parameters in many runtime systems, the three parameters in the algorithm should be decided empirically. The decay factor γ determines how fast the influence of an earlier speculation decays; it decides the tradeoff between robustness and responsiveness of the system. The aggressiveness factor β determines the aggressiveness of the system in exploring higher level speculations. The gain threshold *G_TH* determines when to decrease and increase the speculation depth. Section IV shows that a single set of values for those parameters are enough for a variety of applications.

IV. Evaluation

We evaluate the adaptive speculation techniques on real applications as well as some constructed traces. Using traces allows us to measure the robustness of the techniques in a broader range of scenarios; the evaluation on real applications measures the effectiveness of the technique in practical uses.

A. Evaluation Metrics

We use *cost efficency ratio* as the metric in our evaluation. It is defined as follows:

$$cost\ efficiency\ ratio = \frac{speedup}{cost\ ratio};$$
$$(speedup = \frac{T_p}{T_s}; \quad cost\ ratio = \frac{\sum_{i=1}^{K} t_i}{T_s}),$$

where, T_p and T_s are the run times of the program with and without speculations (p for parallel; s for sequential), t_i is the time that process i actively runs for, and K is the total number of processes that are created in the parallel execution. *Cost* measures the total use of the computing units. *Cost ratio* is the cost normalized to what the sequential run takes; the lower the better, and the lower bound is 1.

Because *BOP* uses understudy processes, failed speculations cause little slowdown to the parallel execution. So, for *BOP*, the main goal of the adaptive speculation is to reduce the cost ratio to close to 1, meanwhile, achieving similar speedup as the default *BOP* does. It implies the maximization of the cost efficiency ratio.

We stress that using understudy processes is a special feature of *BOP*. Most other speculation systems have no similar mechanisms. For them, by avoiding failed speculations, the adaptive speculation will be able to reduce the number of rollbacks, and thus improve the performance of parallel executions, besides the enhancement of cost efficiency.

B. Evaluation on Traces of Various Patterns

We first test the loop-level adaptive speculation on 6 synthetic traces. Each trace contains 2 million elements, whose values are either “1” or “0”, representing a profitable or non-profitable *PPR* instance. In trace-1, 5% elements are “0”; they randomly distribute through the trace. Trace-2 is the opposite, with 95% elements being “0”. The other 4 traces all have phase changes. There are two kinds of phases, containing either 5% “1” or 5% “0”. Each of the 4 combined traces is composed of a number of interleaving instances of the two kinds of phases. The 4 traces differ in the length of a phase instance, ranging from 10^2 to 10^5. These 6 traces represent different profitability patterns and phase changes.

We apply the loop-level adaptive algorithm to the traces to online learn the profitability patterns and produce predictions. In the evaluation, each PPR instance is assumed to take a single time unit, and a d-level successful speculation saves d time units.

Table I shows the results. The adaptive speculation has little effect on the speedup of applications. However, it reduces the cost substantially in both of the two speculation depths. Except on trace-1, it enhances the cost-efficiency of *BOP* by 14–72%. The enhancement is due to the caution and dynamic control in the adaptive speculation. As a tradeoff, on the almost completely profitable trace (trace-1), there is a 4.6% decrease of cost-efficiency; the aggressive speculation by *BOP* happens to fit the trace better. Overall, the adaptive speculation is effective in predicting, and adapting to, the profitability patterns in those traces. The results on the combined traces demonstrate its robustness to phase changes of different frequencies.

The values for the three parameters γ, G_TH, and β are chosen empirically ($\gamma = 0.4$, *G_TH*$=0.25$, $\beta = 0.0037$.) Our experiments have seen that a broad range of values for the three parameters generate results similar to what Table I shows. (Details omitted for lack of space.) That set of parameter values are used throughout all our experiments including the following ones.

C. Results on Applications

1) Methodology: Our experiments run on an Intel quad-core Xeon machine (3.4GHz) with Linux 2.6.23 installed. All applications are compiled using GCC 4.0.1 with a “-O3” flag.

Table II lists the benchmarks we use. They are chosen mainly for two reasons. First, as *BOP* is specifically designed for exploiting coarse-grained dynamic parallelism [4], such kind of parallelism must exist in the benchmarks that are meaningful for *BOP* studies. Many programs in existing benchmark suites are not qualified. As showed in previous work [4], [14], the programs in Table II have proved to meet that requirement. Second, the behaviors of these programs change substantially on different inputs, making them especially suitable for this work, given that the focus of this study is on addressing input-sensitivity in speculations.

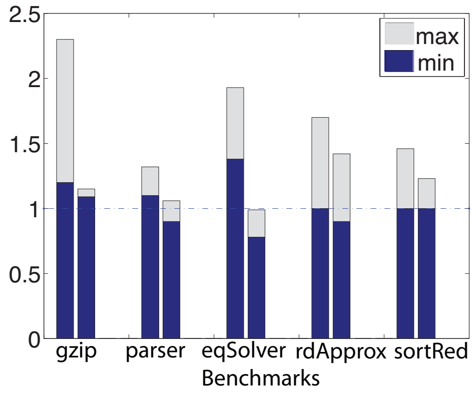


Fig. 5. The ranges of cost-efficiency ratios (left of a bar pair) and speedups (right of a bar pair) brought by adaptive speculation.

Among those five benchmarks, the first two contain loop-level parallelism, and the others have function-level parallelism. The small number of inputs to the programs with loop-level parallelism is enough to cover various behavior patterns of those programs, while the more inputs to the other programs are necessary for evaluation of the cross-run learning technique. These benchmarks come from various domains with a spectrum of inputs and features, forming the representative for a range of diverse applications.

2) Results: Figure 5 reports the cost-efficiency ratios and speedups brought by adaptive speculation on all benchmarks. The maximum speculation depth is 3.

The few inputs on each of the first two benchmarks induce a spectrum of profitability patterns. For a given file, the sequential code of *gzip* compresses one buffer per iteration and stores the results until the output buffer is full. The *PPR* markers are around the buffer loop. We use an 80MB file as input and set the buffer size to 192KB, 320KB, or 1.6MB. The different buffer sizes affect the granularity of the parallelism and thus the profitability of the *PPR* instances. When the buffer is small, the benefits of speculation is hard to offset the overhead. The adaptive speculation successfully avoids many unprofitable *PPR* instances, yielding 130% improvement of cost-efficiency. Meanwhile, the program runs 15% faster than in the default *BOP* system. The speedup is mainly because the smaller number of processes reduces memory bandwidth pressure. On large buffers, there are fewer speculation failures, hence the relatively modest improvement of cost-efficiency.

The program, *parser*, is an English sentence parsing tool. Its *PPR* is the loop for parsing a group of sentences. Two factors determine the profitablity of the *PPR*s in the program: the group size, and the content of the input sentences. The former determines the *PPR* granularity; the latter may cause dependences among *PPR* instances when an input sentence is a command sentence (i.e., sentences starting with “!”.) The input file we use is derived from the *ref* input, containing 400 sentences with 2 command sentences. We set the group size to be 2, 5, 10, and 50. The cost efficiency improves by 38–93%. The adaptation causes certain decrease of program performance as

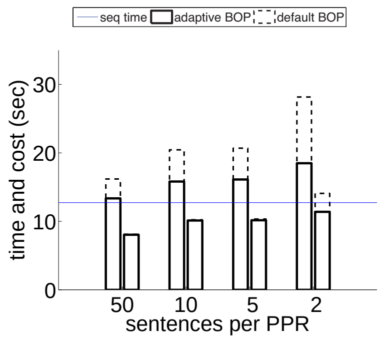
TABLE I
COMPARISON OF BOP AND ADAPTIVE BOP (ABOP) ON DIFFERENT PROFIT PATTERNS*

		max depth=1					max depth=3				
		speedup		cost-ratio		$\frac{CE_{abop}\%}{CE_{bop}}$	speedup		cost-ratio		$\frac{CE_{abop}\%}{CE_{bop}}$
trace	phase length	BOP	ABOP	BOP	ABOP		BOP	ABOP	BOP	ABOP	
1	2×10^6	1.90	1.90	1.03	1.03	100	2.79	2.59	1.11	1.08	95.4
2	2×10^6	1.03	1.00	1.47	1.02	139.9	1.00	1.00	1.75	1.02	171.6
3	10^2	1.33	1.27	1.25	1.04	114.8	1.47	1.27	1.43	1.04	118.8
4	10^3	1.33	1.30	1.25	1.02	119.8	1.47	1.41	1.43	1.05	130.6
5	10^4	1.33	1.31	1.25	1.02	120.7	1.48	1.45	1.43	1.05	133.4
6	10^5	1.33	1.31	1.25	1.02	120.7	1.47	1.45	1.43	1.05	134.3

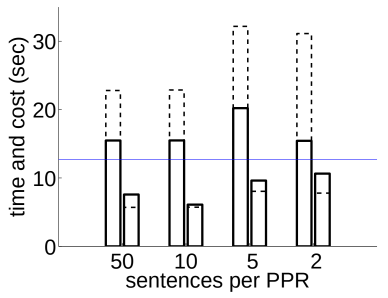
$*$ CE: cost-efficiency ratio; $\gamma = 0.4$, $G\_TH = 0.25$, $\beta = 0.0037$.

TABLE II
BENCHMARKS

benchmark	description	parallelism	input#	features of input
gzip	GNU compression tool v1.2.4 [1]	loop-level	3	-
parser	Sleator-Temperley Link Parser v2.1 [1]	loop-level	4	-
eqSolver	linear equation system solver [4]	function-level	34	num of equations per system
rdApprox	locality approximation from time [13], [14]	function-level	20	num of variables, length of trace, reuse distance histogram patterns
sortRed	sort and reduction of arrays [3]	function-level	24	array size, array type (sorted or not)



(a) maximum speculation depth is 1



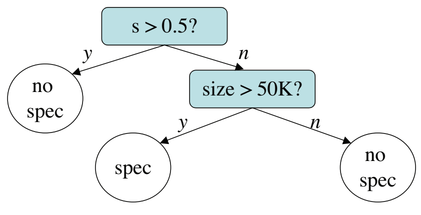
(b) maximum speculation depth is 3

Fig. 6. The sequential time, the cost (left of a bar pair), and the parallel execution time (right of a bar pair) of *Parser* on various PPR granularities.

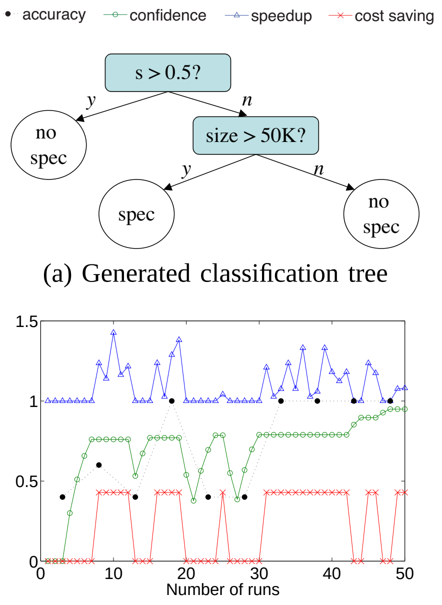
a tradeoff of the much more significant cost reduction. Figure 6 shows more detailed results. When the maximum speculation depth is 1, the adaptive speculation reduces computing cost by 12.5–35%, and meanwhile improving performance by up to 21%. When the maximum depth is 3, the executions become faster. Although the adaptive speculation causes certain loss of speedups due to prediction errors in the warm-up stage, the cost saving is much more substantial, 38–93%.

The other three programs all contain function-level parallelism. *EqSolver* is a program previously used for comparing *BOP* and fine-grained thread-level parallelism [4]. The program solves 8 independent systems of equations using the *dgesv* routine in the Intel Math Kernel Library 9.0. The solving of each equation system is a *PPR*. The 34 inputs used in the experiment have different equation system sizes, ranging from 200 to 2500. The program, *rdApprox*, is a tool (publically available [13]) for approximating the data locality or reuse distances of a program's execution from its time distances [14]. The program contains 913 lines of code, 22 functions, many pointer uses and dynamic memory management. Its *PPR*s are the two most time-consuming steps among all 9 steps of computation. Its input set includes 20 memory reference traces, whose sizes range from 500 to 100,000. The program, *sortRed*, is derived from a reduction program used in previous work [3]. It sorts an array and then conducts a series of reduction operations to either the original or the sorted array. Unlike *eqSolver* or *rdApprox*, the profitability of its *PPR*s depends on not only the size of the problem, but also the value of an input option, "-s", which determines whether the original ("-s 0") or the sorted array ("-s 1") will be used in the reduction step. The sorting and reduction are marked as the two *PPR*s in the program.

In the experiments on function-level parallelism, each time, we randomly pick one input to run the program. The cross-run learner gradually builds up a classification tree. Figure 7 (a) shows the classification tree of *sortRed*. Figure 7 (b) reports the evolvement of the prediction accuracy and confidence as more runs are seen. The ascending trends reflect the improvement of the prediction model. The cost efficiency is improved by up to 242%; the speedup attains up to 45% enhancement. The large improvements occur when the confidence is high enough and the speculations happen to be unprofitable. The avoidance of failed speculations also significantly removes the memory protection overhead on the critical path, thus bringing the speedup. On *eqSolver* and *rdApprox*, the cost-efficiency



(a) Generated classification tree



(b) Detailed results

Fig. 7. Incremental learning results on *sortRed*; "s" for option "-s"; "size" for the array size. (Each black dot is the mean accuracy of 5 runs.)

improvements are respectively up to 70% and 46%, and the speedups are up to 42% and 23%.

V. Related Work

Automatic loop-level software speculation is pioneered by the lazy privatizing *doall* (LPD) test [11]. Later techniques speculatively privatize shared arrays (to allow for false dependences) and combine the marking and checking phases (to guarantee progress) [2], [7]. Two programmable systems are developed in recent years: *safe future* in Java [21] and *ordered transactions* in X10 [20]. 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 [17].

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 [6], [10], [19]. This work makes software speculative parallelization adaptive by predicting the profitability of a speculative execution. Last-value-based prediction has been commonly used in both hardware, such as dynamic branch prediction, and software, such as phase prediction (e.g., [15], [16]). The profitability prediction differs from prior problems in that the history information is only partially uncovered, and thus more sophisticated algorithms become necessary.

VI. Conclusion

In this work, we propose the use of *transparent* statistical learning to make speculation cross-input adaptive. The technique is unique in that it requires no explicit training, but is able to adapt to changes in program inputs. It is applicable to both loop-level and function-level parallelism, permitting arbitrary depth of speculations. Experiments in *BOP* show the promise for reducing the waste of computing resource in software speculations, a desirable feature especially for portable devices and multiprogramming environments.

Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. 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 or IBM.

References

[1] SPEC CPU benchmarks. http://www.spec.org/benchmarks.html.
[2] 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.
[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Behavior-oriented parallelization. Technical Report TR904, Computer Science Department, University of Rochester, 2006.
[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior-oriented parallelization. In *Proceedings of PLDI*, San Diego, USA, 2007.
[5] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of PLDI*, 2003.
[6] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of PLDI*, 2004.
[7] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. In *Proceedings of SC*, 1998.
[8] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.
[9] Y. Jiang and X. Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *Proceedings of ICPP*, 2008.
[10] T. A. Jonhson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of PPoPP*, 2007.
[11] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of PLDI*, June 1995.
[12] X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Proceedings of LCPC*, 2007.
[13] X. Shen and J. Shaw. Toolkit for approximating locality from time. http://www.cs.wm.edu/˜xshen/Software/Locality.
[14] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of POPL*, 2007.
[15] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of ASPLOS*, 2004.
[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of ASPLOS*, San Jose, CA, October 2002.
[17] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of ISCA*, 1995.
[18] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceeding of Micro*, 2008.
[19] T.N. Vijaykumar and G.S Sohi. Task selection for a multiscalar processor. In *Proceedings of Micro*, 1998.
[20] Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *PPoPP*, March 2007.
[21] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for java. In *OOPSLA*, pages 439–453, 2005.