CS229 Project: TLS, using Learning to Speculate

Jiwon Seo
Dept. of Electrical Engineering
jiwon@stanford.edu

Sang Kyun Kim
Dept. of Electrical Engineering
skkim38@stanford.edu

ABSTRACT

We apply machine learning to thread level speculation, a future hardware framework for parallelizing sequential programs. By using machine learning to determine the parallel regions, the overall performance is nearly as good as the best heuristics for each application.

1. INTRODUCTION

Parallelizing applications has become widely accepted as an inevitable trend to boost the performance of computers. Major microprocessor vendors have already announced their roadmap as multi-cores. However, the conventional method of parallelizing applications with locks significantly decreases programmability, hindering us from effectively exploiting the increasing computational resources. Thus, it has become crucial to devise a new, but simple framework for parallelizing applications. Recent studies have proposed several ways to replace the locking method and ease the work of parallelizing applications, among which thread-level speculation is a well-known candidate [1].

Thread level speculation (TLS) is a hardware technique to parallelize sequential programs by speculation. Although this is a promising approach to further utilize many-core architectures, choosing the speculation points is still a challenging task. Previous researches have attempted to solve this problem using heuristics [2, 3]. Although some heuristics have shown near-optimal performance for a limited set of applications, their applicability to other applications is questionable.

We propose using machine learning to find the optimal speculative regions of method level speculation. More specifically, our approach attempts to classify each method invocation site and speculatively fork threads at the correct call sites. By training the algorithms with various applications, this approach provides a general framework which efficiently parallelizes a wide range of applications.

Our main contributions are:

- This study is the first to explore the potential of using machine learning algorithms to automatically detect the parallel regions for thread level speculation.

- Our approach does not rely on runtime information. Using runtime information either involves additional runtime overhead or time consuming profiling. By combining machine learning with statically extracted features, we eliminate this burden.

- Machine learning offers a general framework that shows reasonable performance for a wide range of applications.

2. METHOD LEVEL SPECULATION
2.1 Overview

Over the last decade, thread level speculation has been widely recognized as a promising way to facilitate the shift from sequential to parallel programming. Unlike other parallelizing techniques, such as transactional memory [4], TLS exploits the parallelism in sequential applications by speculatively executing segments of code in parallel. Potentially parallel segments of a sequential code are usually identified and annotated by the runtime system or TLS compilers. Using the annotated data, TLS hardware then optimistically forks threads and maintains the flow of execution. Since parallelizing the code must not alter the application behavior, the TLS hardware also keeps track of memory loads and stores to detect true dependency violations. If such violation occurs among threads, the more speculative threads will roll back and re-execute their piece of code.

Locating the regions to speculatively parallelize has proven a difficult task. Randomly selecting any segment of code may actually degrade the performance due to excessive violations. Previous research has mainly focused on finding parallel regions at the loop level [5]. Parallelization in this case is achieved by mapping iterations to different threads. It is also possible to parallelize applications at the method level [2, 3], where speculative threads are forked at method invocation sites. Due to time constraints, we focus here only on method level speculation in this project.

2.2 Challenges

Although previous researchers have demonstrated the promise of TLS, the proposed heuristics are far from perfect. For example, these heuristics tend to favor different applications, thus limiting their applicability. Whaley [2] shows that a heuristic optimal for one application may

Method-Specific Features	Description
Store count	The number of memory stores within the method. The probability of a violation increases with the number of stores.
Code length	The length of the code. This number attempts to capture the correlation (if any) between the code size and the method execution time.
Call count	The number of method invocations in this method. The probability of longer execution time increases with the number of call count.
Loop count	The number of loops in this method. The probability of longer execution time increases with the number of loops (especially nested loops).

Call Site-Aware Features	Description
Distance to conflict	The minimum distance to a potential RAW conflict when the call site continues to run speculatively. The probability of a violation increases as the minimum distance to potential conflict decreases.
Number of conflicting accesses	The number of potential RAW conflicts when the call site continues to run speculatively. The higher this number, the greater the probability of a violation.
Remaining code length	The remaining code length of the method containing this call site. The probability of wasting idle cycles decreases with the remaining code length.

Table 1. The features used for learning

perform poorly on other applications. Therefore, given a specific application, it is difficult to decide which heuristic to use.

Another challenge in thread level speculation is the adverse side effects when using runtime information. Many existing heuristics require profiling data or runtime statistics. Although profiling provides useful hints about the actual execution of interest, profiling itself is impractical in many cases since the user must run the applications beforehand. Collecting runtime statistics and applying heuristics on the fly may also be undesirable as these methods impose significant additional runtime overhead.

3. MACHINE LEARNING APPROACH
3.1 Overview
Machine learning is a broad research field that studies and develops algorithms by which computers "learn" to extract useful information from a set of data. More specifically, machine learning algorithms capture the inherent statistical characteristics of a given data set and effectively predict the characteristics of new data. Therefore, if some information extracted from a piece of code can be mapped to a statistical model, we can use machine learning algorithms to obtain interesting characteristics of the code. In this paper, we propose using machine learning algorithms to find the statistically optimal parallel regions of a sequential code. In particular, we focus on method level speculation; hence, speculative regions are chosen by applying the learning algorithms at each method invocation site.

In contrast to the previous research discussed above, we do not perform any runtime analysis. We believe that the static context in a code is likely to contain sufficient information

of potentially parallelizable regions. Intuitively, the functions with similar runtime behavior, e.g. execution time, are expected to share some characteristics in their static context. Thus, although this approach does not guarantee high performance, machine learning algorithms should maximize the likelihood of performance optimality.

Since the workload size is determined at runtime, this feature cannot be extracted from static analysis. For simplicity, we assume that the workload size is always adequately large with the justification that most applications of interest require a large amount of runtime. Small-sized workloads may also show different parallel behavior even with the same code; therefore it is infeasible to specify parallel regions that uniformly apply to every dataset.

Unlike previous research, the goal of this project is not to exceed the performance of existing heuristic approaches. Amdahl's law implies that the speedup from parallelism is limited by the coverage of the speculative regions. For example, even with 70% coverage, the speedup is limited to 3.3 with an infinite number of cores [8]. Since previous studies using heuristics have shown results somewhat close to this limit [2, 6], only targeting performance would not be a significant contribution. Instead, we demonstrate that the machine learning approach shows comparable performance to the best heuristics for a wide range of applications. In addition, our approach overcomes the difficulty of choosing the optimal heuristic without the need for burdensome profiling.

Although this approach avoids profiling every application, it should be noted that this approach requires considerable amount of time on learning. Such learning process needs to be conducted on every machine as the hypothesis learned

on one machine may not apply to other machines. However, unlike profiling, learning is a one-time procedure, which can be done during OS installation or any other convenient time.

3.2 Features and Training Examples

The features used in a learning algorithm should capture the possibility of both speedup and violation. Table 1 describes the features used in this project. As can be seen, method-specific features provide more information on speedup, whereas call site-aware features extract violation probabilities. In selecting the method-specific features, we assume that the method execution time has a strong correlation with the potential speedup; that is, methods that tend to have a certain amount of execution time are likely to give good speedup. The exact correlation is up to the machine learning algorithm to decide. Table 1 also shows features which detect potential RAW (read-after-write) conflicts of memory access. To avoid complicated pointer analysis, we simplify the detection process by declaring a potential conflict when the types of objects loaded by a speculative thread overlap with those stored by an older thread. Although this naive analysis may produce false positives, machine learning algorithms are capable of overcoming such "noise."

Features are obtained using a two-pass Java bytecode analyzer. During the first pass, the analyzer gathers method information from the body of each method; this includes all of the method-specific features. With this information, the call site-aware features are computed in the second pass. Although we could have also explored sophisticated multiple-pass features, we streamlined this project due to time constraints.

In addition to extracting features, we need to label the training examples. Determining the label for given data, however, is not obvious, even to programmers, since optimal labeling may vary from system to system. One way to label the data is to exhaustively parallelize every combination of call sites and select the labeling with the best performance. However, the time required for this approach grows on the order of $O(2^{N: \text{# of call sites}})$, which is infeasible in most cases. Keeping in mind our time constraints, we adopt a greedy approach, which enables us to take advantage of previous research on heuristic approaches and use them as a baseline example. Since the previous results are close to the optimal case, we assume that their annotations of speculative regions are similar to those of the optimal case. From this baseline result, we alter one annotation at a time for each call site until we find the one that exhibits the best performance. We can repeat this step for a fixed number of times (r), or until the speedup is negligible. It should be noted that finding this near-optimal set of annotations only takes $O(N \cdot r)$ of execution time.

SpecJVM98	Description
_202_jess	Java expert shell system
_213_javac	Java compiler from the JDK 1..0.2
_227_mtrt	A variant of _205_raytrace
_228_jack	Java parser generator based on the PCCTS.

SPLASH-2	Description
barnes	Gravitational N-body simulation
water	System of water molecules simulation

Table 2. Java benchmarks

4. SIMULATION METHODOLOGY

4.1 Tools and benchmarks

To simplify the implementation, we reuse the trace-driven simulation infrastructure used in [2], including the instrumentation tool, trace analyzer, and trace-based TLS simulator. This trace-driven TLS simulation proceeds as follows. Traces are obtained by executing instrumented codes on a native AMD Opteron™ machine. Using the traces, the trace analyzer produces annotations of speculative regions according to a manually specified heuristic. The TLS simulator then steps through each trace and speculatively forks threads at annotated locations.

To implement and evaluate our machine learning approach, we modify the instrumentation tool to include call site locations in the trace. Since no runtime information is needed, application traces are not analyzed. As mentioned in section 3.2, we instead implement a Java bytecode analyzer to extract features from Java applications. These features, with labels, comprise the training and test data. Speculative regions are then selected by the learning algorithm for each method invocation site. In addition, we use the Weka machine learning software suite to easily compare different algorithms.

The TLS simulator only detects violations between memory accesses to heap data; that is, dependencies between stack variables are neglected. Without violation detection of stack variables, we cannot speculatively parallelize non-void methods since return values are stored in the stack. Due to time constraints, we did not resolve this issue; instead, we only considered parallelizing void methods. To simplify the simulation process even further, we only allow forking speculative threads at the beginning of a method invocation. Delaying thread forks may reduce CPU idle time to further utilize computation resources. However, without runtime information, it is unclear how to obtain the optimal forking point.

We use the Java benchmarks from [2] to create training and test data. Java applications, like other object-oriented
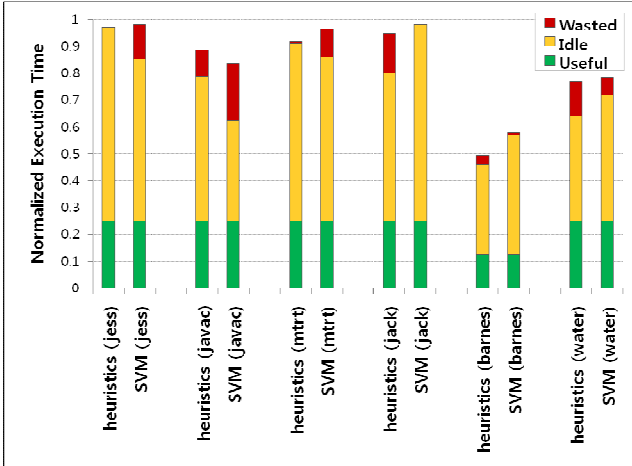
Figure 1. Speedup and execution time breakdown

programs, exhibit a large number of method calls, which is desirable for evaluating our method level speculation approach. Since simulating a benchmark is time consuming, we use only a subset of applications used in [2] for the experiment. Table 2 summarizes the Java benchmarks used in this project.

5. RESULTS AND DISCUSSION

For each application, we randomly chose half of the call sites for training examples and used the other half for test data. Using this data, we trained various machine learning algorithms, including logistic regression, Naïve Bayes, and SVM, among which the SVM algorithm shows the best average test error. Since the separating hyperplane is likely to be nonlinear, we use kernels to work in high dimensional feature space. After several experiments on the parameters, SVM with normalized polynomial kernel of degree 15 and a complexity constant of 200 yielded the best test errors on average.

5.1 Speedup Comparison

To evaluate the performance of our approach, we compare the speed up of TLS driven by heuristics and machine learning. Due to time constraints, we chose two heuristics from [2], which are SI-RT (runtime heuristic) and SI-SC (store heuristic). According to [2], these heuristics have shown the best results for barnes[1], jess, mtrt, and water. Although neither SI-RT nor SI-SC are the optimum heuristics for jack and javac, at least one performed somewhat comparably to the best speedup.

Figure 1 shows the speedups of the heuristics and machine learning approaches. As can be seen, the SVM speedups are close to those of the heuristics. In particular, SVM actually performs better than heuristics for javac. The results of some benchmarks differ from those in [2]; i.e. jess, mtrt,

---

[1] For barnes, we use eight processors.

Benchmark	Test Error	Benchmark	Test Error
jess	34.82%	javac	37.78%
mtrt	35.48%	jack	28.39%
barnes	36.75%	water	32.39%

Training Error = 24.37%

**Table 3. Training and test error rates**

and jack exhibit only 2% to 10% speedup, in contrast to 30% to 60% speedup in [2]. Since we only consider void methods for speculative candidates, it is very probable that such low speedup is due to the lack of inherent parallelism in void methods of those benchmarks. Nevertheless, SVM performs similarly to the best heuristics, suggesting that machine learning offers a promising approach for TLS.

5.2 Learning Results

Table 3 lists the training and test errors of each application. As can be seen, the error rates are relatively high compared to those of other typical machine learning uses. At first glance, this may seem to conflict with the results in section 5.1 since the performance level should drop at each misprediction. However, it should be noted that not all call sites may have the same impact on performance. In software engineering, it is usually assumed that 90% of the execution time of an application is spent in 10% of the code [9]. Therefore, we can assume that only a few speculative regions strongly influence the overall performance, while all other speculative candidates have negligible effect. Thus, as long as these critical regions are correctly predicted, mispredictions at other call sites do not significantly degrade the performance. For javac, barnes, and water, we believe that the learning algorithm successfully classified the critical regions, resulting in good speedup despite the high error rates. However, for the other applications, it is difficult to suggest such claim since the best heuristics for these applications performed poorly to start with.

Although high error rates may not necessarily lead to low performance, skewing toward randomness is unacceptable. For example, the average test error rate measured above is around 30%. If we assume that there is only one critical region per application, then statistically 30% of the applications should show low performance. To ensure reasonable performance for a wide range of applications, the error rates should be kept low as possible. In addition, since the call sites are not equally important, this implies that we need a new cost function with different weights for speculative regions, and a different measure to evaluate the accuracy of the learning algorithm.

We attribute the high error rates to several factors. One factor is that the current features do not completely capture all necessary information of parallelization. For example, the features explained in section 3.2 include no information
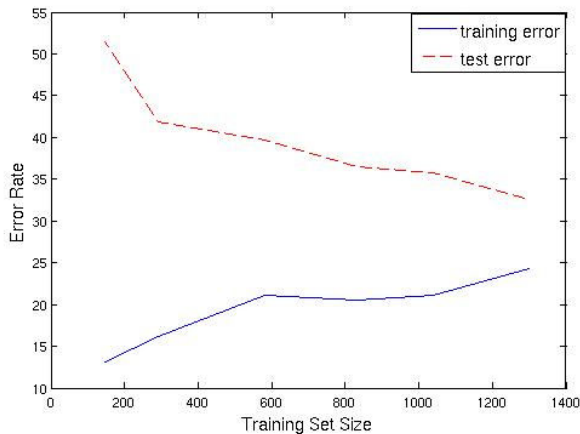
Figure 2. Error rates vs. training set size

about the number of processors. Failure to consider the number of processors may result in excessive thread forks, which was the case for barnes. As mentioned in section 5.1, the speedup from SVM approached the speedup of heuristics only after increasing the number of CPUs. Figure 2 also illustrates the high bias of our algorithm. As the training set size increases, the training error quickly converges to the test error, suggesting that more features are needed. Another factor is the imperfect labeling. A greedy approach was used to label the training and test data, thus increasing the noise in the data.

6. FUTURE RESEARCH

In order to show reasonable speedup for every application, parallelizing non-void methods is inevitable. This approach requires considerable changes to the bytecode analyzer as well as the TLS simulator. Since return values of methods are usually used immediately, we also need to implement value prediction for the return values [7]. These changes may affect machine learning algorithms and/or require additional features.

As mentioned above, we need to resolve the high bias of our model. Thus, experimenting with a larger number of features is one possible direction for future research. We can also reduce bias by improving the current features using sophisticated static analysis, e.g. alias analysis.

Since correctly classifying the critical speculative region is more important than the overall error rate, we can also devise a weighted cost function that favors the critical regions. However, applying such a cost function introduces new problems, such as defining and locating a critical region.

Another interesting direction is to integrate our approach with a runtime system. Although runtime information was excluded from our algorithm, the runtime system itself can be used to compensate for the machine learning approach.

For instance, the runtime system can count the number of violations for a method and correct the mispredictions if the count reaches a certain threshold. If dynamic binary translation is included, we can also target non-Java applications without their source codes [8].

7. CONCLUSION

This study is the first to apply machine learning algorithms to thread level speculation. In particular, we target method level speculation by learning the optimal speculative points from static features. Features are extracted using a Java bytecode analyzer. Simulation results show that SVM performs almost well as the best heuristics for each application. The high training and test error rates, however, need to be significantly reduced. Parallelizing non-void methods should also be considered.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] Sohi, G. S. et al. Multiscalar Processors. In *Proc. of the Intl. Symp. on Computer Architecture*, June 1995

[2] Whaley, J. and Kozyrakis, C. Heuristics for Profile-driven Method-level Speculative Parallelization. In *Proc. of the Intl. Conference on Parallel Processing*, 2005

[3] Chen, M. K. and Olukotun, K. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proc. of the Intl. Symp. On Computer Architecture*, Oct. 1998

[4] Herlihy, M. and Moss, J. E. B. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. Of the Intl. Symp. On Computer Architecture*, 1993

[5] Chen, M. K. and Olukotun, K. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proc. of the Intl. Symp. on Computer Architecture*, June 2003

[6] Warg, F. and Stenstrom, P. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proc. of the Intl. Parallel and Distributed Processing Symp.*, 2003

[7] Oplinger, J. T. et al. In Search of Speculative Thread-Level Parallelism. In *Proc. of the Intl. Conference on Supercomputing*, June 1999

[8] Hertzberg, B. and Olukotun, K. DBT86: Dynamic Adaptive Thread-Level Speculation. Submitted to *Proc. of the Intl. Symp. On Computer Architecture*, 2008

[9] http://en.wikipedia.org/wiki/Pareto_principle