Adaptive Fork-Heuristics for Software Thread-Level Speculation

Zhen Cao and Clark Verbrugge

School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
zhen.cao@mail.mcgill.ca, clump@cs.mcgill.ca

Abstract. Fork-heuristics play a key role in software Thread-Level Speculation (TLS). Current fork-heuristics either lack real parallel execution environment information to accurately evaluate fork points and/or focus on hardware-TLS implementation which cannot be directly applied to software TLS. This paper proposes adaptive fork-heuristics as well as a feedback-based selection technique to overcome the problems. Adaptive fork-heuristics insert and speculate on all potential fork/join points and purely rely on the runtime system to disable inappropriate ones. Feedback-based selection produces parallelized programs with ideal speedups using log files generated by adaptive heuristics. Experiments of three scientific computing benchmarks on a 64-core machine show that feedback-based selection and adaptive heuristics achieve more than 88% and 50% speedups of the manual-parallel version, respectively. For the Barnes-Hut benchmark, feedback-based selection is 49% faster than the manual-parallel version.

Keywords: Software thread-level speculation, fork heuristics, automatic parallelization, performance tuning

1 Introduction

Thread-level speculation (TLS) is a safety-guaranteed approach to automatic or implicit parallelization. Speculative threads are optimistically launched at *fork points*, executing a code sequence from *join points* well ahead of their parent thread. Safety is preserved in this speculative model by buffering reads and writes of the speculative thread. Once the parent thread reaches the join point the latter may be *joined,* committing speculative writes to main memory and merging its execution state into the parent thread, provided no read conflicts have occurred. In the presence of conflicts the speculative child execution is discarded or rolled back for re-execution by the parent.

The selection of fork/join points plays a key role in the performance of TLS, especially for software implementations as a result of higher overhead than its hardware counterpart. So far there are three sorts of fork-heuristics: static heuristics [5], static profiling heuristics [4, 10, 11] and dynamic profiling heuristics [9]. The first build mathematical cost-benefit models of speculative execution using compile-time program information, and use the models to predict profitable

fork/join points. This approach has the limitation that some model parameters, such as thread dependency probability and iteration count of nested loops, are unknown at compile-time, which in turn limits its effectiveness and application. The second heuristics compile and run the sequential program, collect profiling execution traces, and then use the traces to determine the best fork/join points. The drawback of this approach is lack of real parallel execution environment information, which limits accuracy of the fork point selection decision. The third approach is more promising for real estimation, but is currently based on hardware implementation, which is inappropriate and cannot be directly applied to software TLS.

This paper proposes adaptive fork-heuristics to solve the above problems. Adaptive heuristics are dynamic profiling heuristics for software TLS, which insert all potential fork/join points into the speculative program and rely entirely on the runtime system to determine profitable fork/join points and disable inappropriate ones. Since fork/join points are evaluated during real speculative parallel execution, all necessary information such as the thread conflict ratio and thread execution time is available, enabling accurate estimation of cost-benefit of each thread and thus each pair of fork/join points. On-the-fly fork/join point selection also eliminates the requirement of profiling runs and enables adaptation to different fork/join points for different input data. Our investigation demonstrates feasibility of this approach, as well as providing concrete data on actual performance in a realistic thread-level speculative system.

2 Related Work

The bulk of proposed fork-heuristics are static profiling heuristics. Java runtime parallelizing machine (Jrpm) [4], for instance, first profiles execution of a sequential program with a hardware profiler, and then dynamically speculates on the selected prospective loops after collecting enough profiling data to decide the best loops to parallelize. Du et al. [6] proposed a cost-model-driven compilation framework SPT to select candidate loops for speculative parallelization, which builds control-flow graphs and data-dependence graphs with profiling information of a sequential execution and uses the graphs to evaluate candidate loops based on the cost model. The STAMPede [11] TLS approach selects speculatively parallel loops based on several filter criteria: the loop execution coverage and iteration count are above a threshold and the loop body is neither too large nor too small. The Mitosis [10] compiler/architecture uses arbitrary pairs of basic blocks as fork/join points and models parallel execution based on profiling traces to estimate candidate pairs. The POSH [7] compiler simulates sequential execution on a train input set and models TLS parallel execution to select beneficial fork/join points.

There is also research dedicated to static profiling heuristics. Whaley and Kozyrakis [13] proposed three classes of heuristics for method-level speculation, and found that single-pass heuristics lead to best speedups while simple/complex multi-pass heuristics tend to over/under speculation. Wang et al. [12] constructed

a loop-graph and used it for global loop selection to maximize program performance. Liu et al. [8] proposed an online-profiling approach to speculatively parallelize candidate loops. Online static profiling approaches [4, 8] have the advantage over offline-profiling that they do not require additional profiling input and can dynamically profile on the real data. However, these still lack parallel execution environment information for accurate estimation of fork/join points. Pure static heuristics are less common, since they lack runtime parameters. Dou and Cintra [5] proposed a thread-tuple cost-model to estimate speedups of candidate loops. As well, some heuristics combine profiling and static approaches, such as SPT [6].

Dynamic profiling heuristics have recently been studied. Luo et al. [9] proposed a dynamic performance tuning technique for selection of candidate loops. It used hardware performance monitors to profile runtime statistics such as instruction fetch penalty and cache miss, and estimated the efficiency of each thread and loop with the statistics. Unfortunately, this approach cannot be directly applied to software TLS without low-level and machine-specific access to hardware performance monitors.

All the above are hardware-TLS heuristics, which tend to focus on finer-granularity parallelism due to hardware resource constraints. In software-TLS, heuristics should focus on coarser-granularity parallelism as software TLS has higher overhead than hardware implementation. So far as we know, the adaptive heuristics we propose are the first heuristics specifically proposed for and validated in software TLS.

HEUSPEC [14] is a software speculation parallel model that dynamically adapts to different value predictors and granularity tasks. While adaptive fork-heuristics target the problem of fork point selection.

3 Adaptive Fork-Heuristics

Adaptive fork-heuristics add potential pairs of fork/join points to the speculative program, evaluate the cost-benefit of each pair during parallel execution and disable unprofitable ones. The design involves three aspects: (1) what the potential fork/join points are, (2) how to estimate the cost-benefit of each pair of fork/join points, and (3) how to disable fork points.

3.1 Potential Fork/Join Points

The potential fork/join points of the design are loop iterations and function (method) calls, since loops usually take the majority of program's execution time and function calls usually represent independent computation tasks. They are also the choice of most other TLS works, known as loop-level speculation and method-level speculation, respectively.

We also apply two optimizations for each loop: "blockize" and "end-barrier." Suppose there are n processors. Blockize splits the loop iterations into n blocks, which in turn avoids creating too many small threads. The exception is loops

with a small constant number of iterations, which do not need this optimization. The end-barrier optimization adds a barrierpoint just after the end of the loop. This optimization is beneficial because loops usually have dependency with their continuation, particularly for loop nests, in which case an inner loop thread may cause cascading rollbacks of the outer loop threads.

In this implementation, we add directives of fork/join/barrier points and perform the two optimizations manually. This is a limitation of our prototype—both the adding of fork/join/barrier points annotation and the optimizations can be automated by compiler transformation, enabling full automatic parallelization.

3.2 Cost-Benefit Estimation

The design uses a cost-benefit model to evaluate the profitability of each thread and each pair of fork/join points. The model assumes a constant time $T_{overhead}$ of overhead (thread creation, cache miss, buffering, etc) for each thread. Although this is an inaccurate approximation since threads with different memory access frequencies have different buffering overhead, we find it works well for our estimation, partly because we are only concerned with whether a thread is profitable, and not how profitable it is.

The runtime $T_{t,run}$ of a speculative thread t comprises two parts: work time $T_{t,work}$ and synchronization/validation/commit/rollback time, which are available through timing. If thread t commits, its cost-benefit is estimated as $\eta_t = T_{t,work}/(T_{t,run} + T_{overhead})$. If it rolls back, its cost-benefit is 0. Given a minimum cost-benefit threshold $\eta_{threshold}$, if $\eta_t < \eta_{threshold}$, then thread t is considered not profitable and should not have been speculated.

If the assumption holds that threads speculated at the same fork point always show similar behaviour (they always commit/rollback and have similar work time / runtime ratio), then we can directly use η_t to estimate the cost-benefit of the fork point. However, the assumption generally does not hold, even for fork/join points speculating independent loop iterations. The reason is that at the beginning of program execution many threads with dependencies are speculated, causing nondeterministic rollbacks.

We propose 3 independent mechanisms to address this issue: global hint, local hint and interval hint. Global hint uses at least N_{warmup} threads instead of one thread to determine the cost-benefit of a pair of fork/join points. When a thread completes execution, its runtime plus overhead is accumulated to the runtime T_{run} of the pair $T_{run} = T_{run} + T_{t,run} + T_{overhead}$. The exception is when it is cascadingly rolled back, as a cascading rollback does not represent the cost-benefit of a thread. If it commits, its work time is accumulated to the work time T_{work} of the pair $T_{work} = T_{work} + T_{t,work}$. After $N >= N_{warmup}$ threads completes, the cost-benefit of the pair is then estimated as $\eta = T_{work}/T_{run}$. The hint disables the fork point if the cost-benefit is below a threshold. For local hint, if a thread decides not to speculate on a fork point then none of its child threads, grand-child threads, etc can speculate on the fork point. In other words, a local hint affects the sub-tree of a thread, hence the name. Interval hints directly use

the cost-benefit of a thread to decide profitability of its fork point; if a fork-point is disabled, it will try to speculate again after certain amount of time has passed. We find the global hint is the most effective for our benchmarks. It seems to work well on independent loops while the other two might suit more irregular applications. We plan to compare these hints in future work.

3.3 Disabling Fork Points

Each fork point has a globally unique id. The TLS runtime system maintains the attributes of the fork point, which can be accessed given the id. When a thread reaches a fork point, it queries the runtime system with the id whether it can speculate on the fork point. The runtime system then checks a flag variable of fork point attributes and returns the result. When a thread commits/rollbacks, if the adaptive fork-heuristics decide that one fork point is not profitable as discussed in section 3.2, the runtime system then set the flag variable to false to disable the fork point.

If a loop nest has independent outer loops, such as enumerating elements on a matrix, then we can select to speculate on any or all of these loops. Speculating on outer loops enables coarser granularity parallelism but tends to consume more memory than inner ones, while speculating on all loops maximizes parallelism. These decisions have important influence on performance. However, the adaptive fork heuristics will select all speedup loops, even though disabling inner ones may yield further speedups as a result of less thread overhead. Here, we add an option, nest-loop-disabling, to disable an inner loop if its parent nest loop stably commits (its N and η are above the thresholds).

We also propose an optimizing technique called *feedback-based selection* to achieve ideal speedups from the second compilation for our benchmarks. After the program completes execution, it records the cost-benefit of each fork point to a feedback-based selection log file. The next time the TLS compiler compiles the program, it reads the log file and does not insert inappropriate fork/join points as potential candidates. For points that behave differently depending on the input, the programmer can annotate them so that the compiler still insert them even though they are in the log file. The optimization prevents unprofitable fork/join points from hurting performance repeatedly for each compilation.

4 Implementation Framework

We implement the adaptive fork-heuristics into the MUTLS [2, 3] software-TLS compiler framework. MUTLS is a language and architecture independent software TLS framework purely based on the LLVM [1] intermediate representation (IR). It can exploit substantial parallelism from both loop- and method-level speculation. It supports compiler directives to annotate fork/join/barrier points. Each annotation also specifies an id. Threads speculated at a fork point will start execution from the join point with the same id, and will be joined when the non-speculative thread reaches that join point. A thread will also stop execution when it reaches a barrier point with the same id.

A sample program as well as its semiautomatically parallelized program annotated with adaptive fork-heuristics and feedback-based selection log file generated by the MUTLS compiler is illustrated in Figure 1. Given the log file, there are various criteria to decide inappropriate fork/join points, such as whether a fork point is disabled, whether the cost-benefit is below a threshold, and whether the ratio of committed/total threads is below a threshold. Combination of these criteria is also possible. In the current implementation, we simply decide not to add a pair of fork/join points as potential ones if the fork point is disabled.

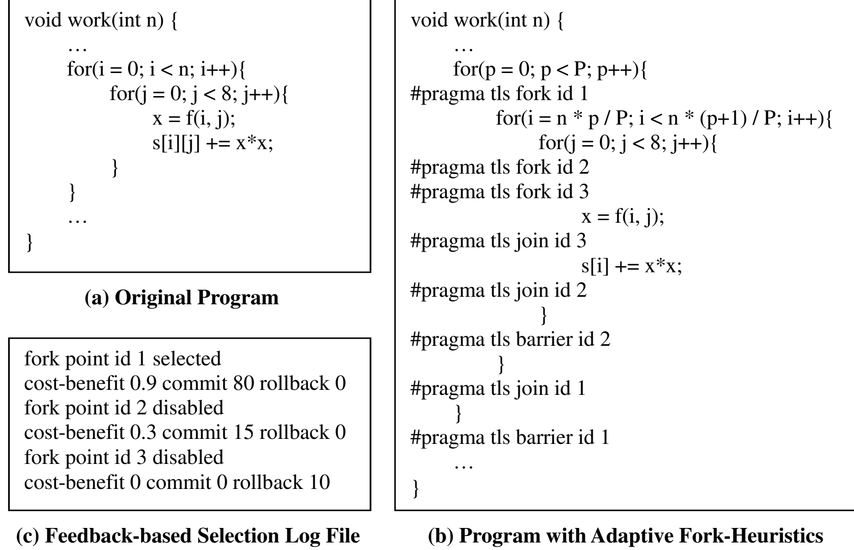

Fig. 1. Semiautomatic-Parallelization of a Program

5 Experimental Results

Experiments are performed on a AMD Opteron machine with 64 2.2GHz processor cores (4×16 core, 8×2MB L2 cache) and 64GB memory. We use three scientific computing benchmarks: Barnes-Hut (bh), molecular dynamics (md) and Mandelbrot (mb). The performance results are shown in Figures 2, 3 and 4 and compared in Figure 5.

For each benchmark, we present the speedups of the manually speculated program (manual version), the semiautomatically parallelized program using adaptive fork-heuristics (adaptive version), as well as the optimized program parallelized by feedback-based selection (feedback version). We also present the results of the program speculating at every fork point (no hint version) as a baseline. The manual version evenly distributes the computing tasks to N processor cores, which serves as the reference implementation. We ran the semiautomatic-parallel programs 10 times each and present the maximum, average (arithmetic mean) and minimum speedups. The feedback version is parallelized using the log file produced by the maximum speedup run.

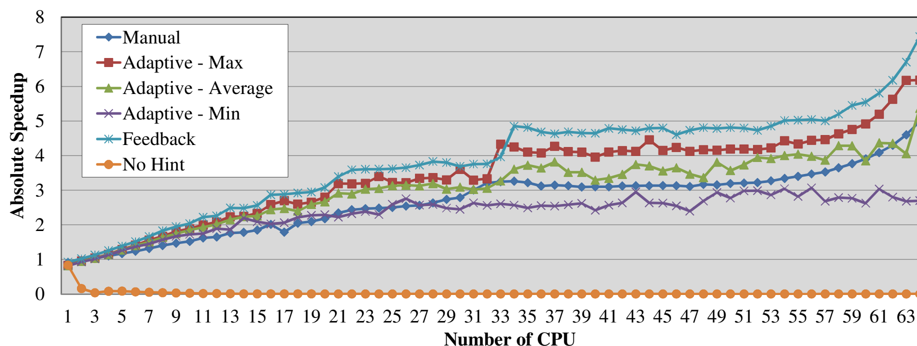

Fig. 2. Speedup Results - Barnes Hut

We can see that the adaptive fork-heuristics perform excellently on these benchmarks. The feedback and adaptive versions of all benchmarks achieve more than 88% and 50% of the manual parallel performance at 64 cores, respectively. The feedback version of Barnes-Hut is even 49% faster than the manual version. The adaptive version also beats the manual one on average. In addition to the top-level task distribution loops, the heuristics also select some fork points in the task computation functions. However, those fork points result in better speedups not because they contribute to more parallelism, but just because of cache issues. In fact, we find that even just adding pure rollback yields higher speedups than the manual version—even failed speculation acts as pre-fetching. This case demonstrates the power of adaptive fork heuristics that can be directly applied on real execution, which can always avoid inappropriate fork points and try to select as more beneficial ones as possible. In contrast, other fork heuristics use pre-defined fork points that do not guarantee benefits on real execution. Moreover, they cannot select such cache-beneficial fork points due to lack of real parallel execution environment information.

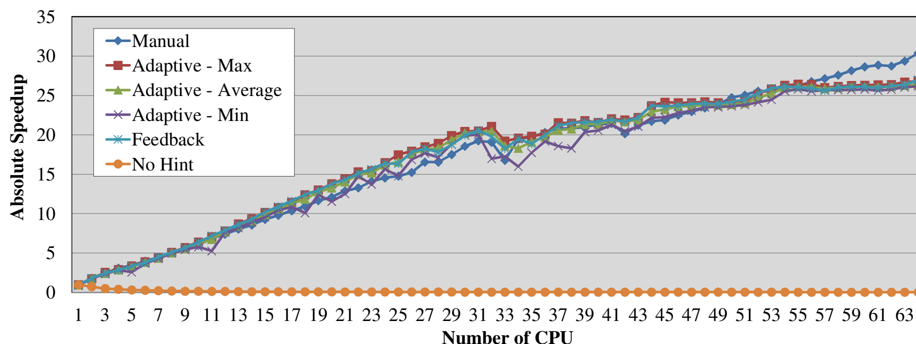

Fig. 3. Speedup Results - Molecular Dynamics

For the md benchmark, all versions show close performance, with little variance in the adaptive runs. Overhead of heuristics is negligible, which demon-

strates satisfactory efficiency and applicability of the heuristics. The adaptive and feedback versions generally show higher performance between 8 and 55 cores than the manual version, but lower with 1 to 7 and 56 to 64 cores, due to different cache behaviours affected by the heuristics.

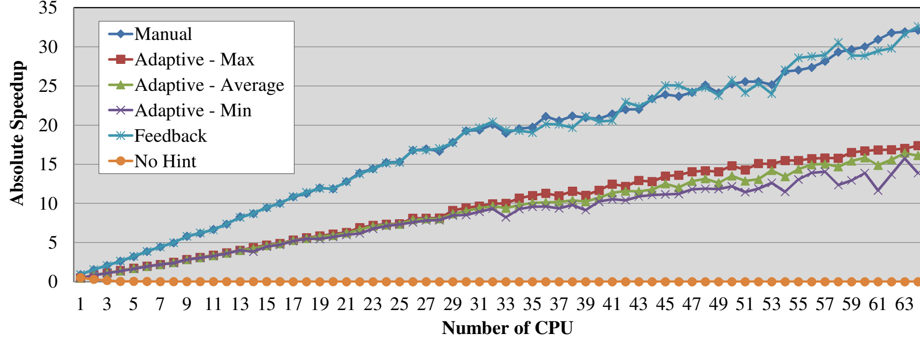

Fig. 4. Speedup Results - Mandelbrot

On the other hand, mb is the least efficient benchmark with respect to the heuristics, due to its small innermost loop body. However, the normalized speedups of the adaptive version is relatively stable with the number of cores, and there is not much variance between each run, which guarantees worst-case performance. The nest-loop-disabling option is used for the benchmark, which improves the speedups by 3 times.

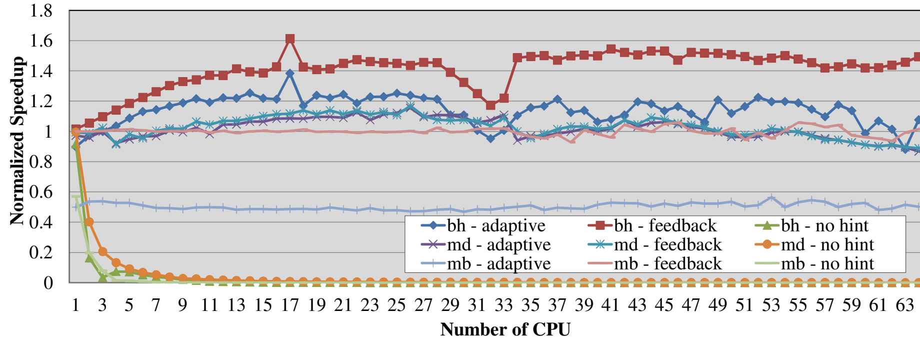

Fig. 5. Performance comparison, speedups normalized to the manual versions

Though the feedback-based selection log file is chosen to be the best speedup one over 10 runs, all the log files select the same fork/join points as the manual version for mb and md. For bh, any of the log files produces a feedback version with significantly higher speedups over the corresponding adaptive ones. Besides, the feedback versions can be further applied feedback-based selection to produce even better feedback versions.

To understand what the best parameters are for the benchmarks, we also experimented with different parameter configurations to see their influence on the performance. The results are illustrated in Figure 6, with speedups normal-

ized to the average speedups of the adaptive versions. For all experiments, we set $\eta_{threshold} = 0.5$ to indicate overhead should not take more time than useful work. We then set the default parameters $T_{overhead} = 1000000$, $N_{warmup} = 10$ and adjust one of them. The speedup data is computed by geometric means over 10 runs on 64 processor cores.

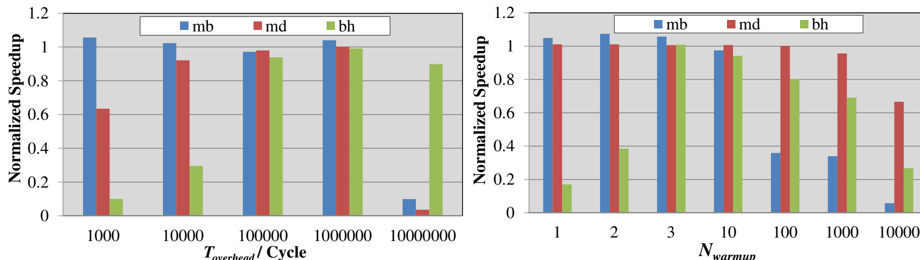

Fig. 6. Parameter Results - $T_{overhead}$ and N_{warmup}

It can be seen that the performance is not very sensitive to the parameters of the heuristics, which is encouraging. The benchmarks have relatively stable performance with $T_{overhead}$ between 100000 and 1000000 CPU cycles and N_{warmup} between 3 and 10. In general, the performance degrades outside these ranges, except for mb, which prefers smaller thread overhead and less warmup runs. The significant performance drop of N_{warmup} from 10 to 100 is because a large number of warmup runs prevents the nest-loop-disabling optimization. It is also remarkable that memory-intensive benchmarks (bh) tend to suit larger $T_{overhead}$ than computation-intensive ones (mb, md).

6 Conclusions and Future Work

In this paper we proposed an adaptive fork-heuristics for software TLS, which inserts all potential fork/join points and purely relies on the heuristics and runtime system to disable the inappropriate ones. These adaptive heuristics have ability to utilize the real parallel execution environment information to maximize performance. In addition, we proposed a feedback-based selection technique to achieve ideal speedups.

Experiments on three scientific computing benchmarks on a 64-core machine demonstrate that the adaptive fork-heuristics are both highly effective and efficient. All benchmarks achieve more than 88% and 50% speedups of the manual version for the programs parallelized by feedback-based selection and adaptive fork-heuristics, respectively. Moreover, the feedback version of Barnes-Hut are 49% faster than the manual version due to exploitation of cache efficiency. Experiments also show the encouraging fact that the heuristics are not overly parameter sensitive.

In future work, we will exploit more accurate cost-benefit models and hints that are more stable and even less parameter dependent, and evaluate with more benchmarks. We will also implement the necessary compiler optimizations to enable fully automatic parallelization with adaptive fork-heuristics.

References

1. LLVM (low-level vitrual machine). http://llvm.org
2. Cao, Z., Verbrugge, C.: Language and architecture independent software thread-level speculation. In: LCPC'12: Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing. pp. 270–272 (2012)
3. Cao, Z., Verbrugge, C.: Mixed model universal software thread-level speculation. In: ICPP'13: Proceedings of the 42nd International Conference on Parallel Processing. pp. 651–660 (2013)
4. Chen, M.K., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. In: ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture. pp. 434–446 (Jun 2003)
5. Dou, J., Cintra, M.: A compiler cost model for speculative parallelization. ACM Transactions on Architecture and Code Optimization 4(2), 12 (2007)
6. Du, Z.H., Lim, C.C., Li, X.F., Yang, C., Zhao, Q., Ngai, T.F.: A cost-driven compilation framework for speculative parallelization of sequential programs. In: PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. pp. 71–81 (Jun 2004)
7. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: PPoPP'06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 158–167 (Mar 2006)
8. Liu, Y., An, H., Liang, B., Wang, L.: An online profile guided optimization approach for speculative parallel threading. In: Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture (ACSAC '07). pp. 28–39 (2007)
9. Luo, Y., Packirisamy, V., Hsu, W.C., Zhai, A., Mungre, N., Tarkas, A.: Dynamic performance tuning for speculative threads. Proceedings of the 36th annual international symposium on Computer architecture (ISCA '09) pp. 462–473 (2009)
10. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In: PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 269–279 (Jun 2005)
11. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C.: The STAMPede approach to thread-level speculation. ACM Transactions on Computer Systems (TOCS) 23(3), 253–300 (Aug 2005)
12. Wang, S., Dai, X., Yellajyosula, K.S., Zhai, A., Yew, P.C.: Loop selection for thread-level speculation. In: Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing (LCPC). pp. 289–303. Springer-Verlag (2005)
13. Whaley, J., Kozyrakis, C.: Heuristics for profile-driven method-level speculative parallelization. In: ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing. pp. 147–156 (Jun 2005)
14. Xu, F., Shen, L., Wang, Z., Guo, H., Su, B., Chen, W.: Heuspec: A software speculation parallel model. In: ICPP'13: Proceedings of the 42nd International Conference on Parallel Processing. pp. 621–630 (2013)