Explore Source-level Software-only Thread-Level Speculation Parallelism

Chengyan (Chuck) Zhao
Department of Computer Science
University of Toronto
Toronto, Canada M5S 3G4
(416) 946-8879
czhao@cs.toronto.edu

ABSTRACT
In this paper, we describe a software only approach to explore Thread-Level Speculative Parallelism in applications' source level.

Categories and Subject Descriptors
Programming Languages, Compilers and Parallelism

General Terms
Algorithms, Performance, Design and Languages

Keywords
Thread-Level Speculation, Transactional Support and Source-level Parallelism

1. INTRODUCTION
Automatically parallelizing applications is a topic with long history since compiler appears. Despite its success in specific domains (e.g. Numerical Analysis related fields), it has only very limited progress in automatically parallelizing general-purpose applications. It is extremely difficult (if not entirely impossible) to perform such compile-time analysis when applications have memory accesses whose input values are based on run-time information or user input. Most compilers will render such cases as non-parallelizable because it cannot statically prove that there are not dependencies.

1.1 Thread-Level Speculation
Thread-Level Speculation (TLS) [4] [5] [1] [2] [3] is a technique to enable compilers to optimistically create parallel threads and execute them on regions with data-dependency ambiguities. This provides a mechanism to effectively explore and extract the dynamic parallelism from regions whose dependencies exist and depend on user input or run-time information. When execute on such regions, data changes will be speculatively buffered. Upon a successful execution, data changes will be made permanent called commit. If any conflict appears, its respective thread will be aborted with speculative data restored to its last commit point and execution resumed to retry the failed speculation. This process of recovery is called abort-and-retry. An example to demonstrate the process is given in Figure 1.

Figure 1 (a) uses the compression algorithm's kernel where 2 hash table entries are referenced using index1 and index2 inside a conditional loop at runtime. The work of iteration is partitioned and assigned to each thread called epoch. Thus the size of epoch represents the granularity. Three threads are executing simultaneously on their assigned epoch space. Thread 1 and 3 conflict on hash[5] as a true-dependency violation. Thus thread 3 is aborted and restarted to produce the correct result.

1.2 Software-only approach
Most of the past researches of TLS [4][5] tend to focus on hardware support, where different types of hardware extensions are proposed, including CPU Instruction Set Architecture (ISA), cache hierarchy, memory sub-system, etc. Since such hardware does not exist, all testings are performed on simulators. Our approach is to use a software solution where there is no need for specialized hardware. This makes it practical to conduct TLS related research utilizing existing parallel machines and perform tests on real parallel architectures.

1.3 Source-level approach
Past work [6] [7] also tends to focus on adding support into compiler backend to utilize such proposed hardware extensions. We propose a source-level approach of using TLS where necessary supports are inserted into source programs rather than compiler backend. The benefit is to provide a quick feedback on the feasibility of software-only TLS approach, and identify the performance critical issues for future improvements.

1.4 Goal
Our goals for this work are 3 fold. First, we want to explore the feasibility of Source-level TLS on a software-only approach. Second, we want to remove locks used in the program to achieve higher parallelism. Third, based on our testing and analysis, we will propose the important directions for further research in this field.

2. Related Work
Steffan et al [2] [3] [4] [8] proposed a hardware extension on CPU's cache hierarchy to support TLS. Each cache line was extended to have additional 3 to 5 bits to hold speculative states, and a state-transition table was built to check all possible violations. At any time, only 1 out of the scheduled N threads is executing in non-speculative mode where all others in speculative mode. Upon conflict, threads in speculative mode always abort and restart immediately. Various approaches have been proposed to optimize epoch size, thread scheduling and inter-thread communications.

Hammond et al. [9] [10] proposed hardware-support of TLS in a different direction where it is based on the Stanford Hydra single-chip Multi-processor (CMP). For their CMP design, 3 independent cores are built into the same chip. All cores operate independently. All cores communicate with memory in a snooping bus where at any given time, only one core could commit and all other cores snoop. The committing core has the priority to finish commit and any snooping core would abort upon discovering a conflict. Different from Steffan's approach, all threads start in speculative mode and wait until commit stage to decide its exact behavior.

Cintra et al [11] proposed using a hierarchy of Memory Disambiguation Tables (MDT) to support TLS across NUMA multi-processors composed from speculative Chip-Multi-Processors. They used hardware to enforce a hierarchical ordering on the threads one level inside the speculative CMP and an other level across the CMPs.

The only software evaluation of TLS we aware of was conducted by Papadimitriou et al. [12] where they were motivated by the success of TreadMarks (TMK) [13] distributed-shared memory system. They parallelized bucketsort using TLS with 512K number of elements to sort with 2048 buckets. Memory-access tracking was on the granularity of Virtual-memory Page level provided by TMK. They ended with huge numbers of page threshing and performance roughly on the range of 2 sec vs. 5 min.

In order to make a direct comparison, we also pick bucketsort algorithm and parallelize it using TLS. We intentionally make the size of unsorted array and buckets the same as in Papadimitriou's report. Rather than relying on TMK to provide virtual memory tracking, we develop our own memory tracking data structures where we could effectively track memory access on word level[1] or even a finer granularity.

The rest of the paper is organized as following. Section 2 introduces related work. Section 3 shows our chosen problem for TLS analysis, steps to parallelize it, challenges we faced, techniques and solutions we used. Section 4 discusses testing setup, different testing results under various parallel architectures with a detailed performance analysis on an optimal configuration. Section 5 points out future research directions. We conclude the paper in Section 6.

```
while (condition){
… …
    a = hash[index1];
… …
    hash[index2] = b;
… …
```

Figure 1 (a) Pseudo-code example for TLS

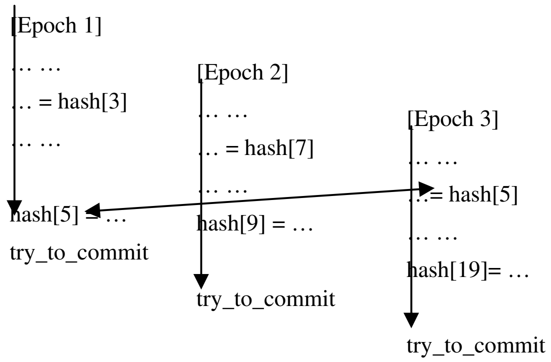

Figure 1. (b) Thread-Level Speculation

3. Problem Statement

In order to exercise TLS, we need a program that has regions of dependency uncertainty. This region cannot be parallelized through statically compilation analysis, but would be ideally suitable for our TLS approach. Another consideration is that most programs spend a significant portion of their execution time in loops. So we would perform TLS parallelization on loops as our starting point. Thus, we pick Bucket Sort as our test example.

3.1 Bucket sort

3.1.1 Bucket Sort algorithm

Bucket sort's algorithm is illustrated in Fig-2.

Bucket sort is used to sort N positive integers which all fall into the range between 0 and M-1. We call M the bucket size and the elements to be sorted are given in array a. The main algorithm consists of three loops, where we mark them as loop 1, loop 2 and loop 3. Loop 1 is used to initialize the bucket array. Loop 2 is used to build entropy of unsorted data into buckets and loop 3 is to go through the bucket entropy and generate the result in ascending order. For better understanding the algorithm, a demo is given in Fig-3.

[1] We performed our tests on slackware Linux where each word is 4 byte integer.

3.1.2 Bucket sort demo

Fig-3 shows the steps and intermediate results when performing bucket sort's algorithm. Step 1 initializes buckets array (of size 8) to be all zeros. Step 2 build source's distribution into buckets. Step 3 will traverse the buckets array. When it is not zero, result will be generated using the current bucket index as reference.

3.1.3 Bucket sort algorithm analysis

Bucket sort uses array indirect reference where source data are used as index to build bucket entropy and bucket index is used to generate result. It has the worse time complexicity of O(M * N). However, since we only have N numbers to sort, this worst case will never happen. The average time complexicity is O(M + N) shows that this is actually a linear sorting algorithm.

```
unsigned int M = number_of_buckets;
void bucketsort (unsigned int a[], unsigned int N){
  int k,j=0;
  for (k=0;k<M;k++){ buckets[k] = 0;}  // loop 1
  for(k=0;k<N;k++){ ++buckets[a[k]];} // loop 2
  counter = 0;
  for(k=0;k<M;k++){                    // loop 3
     for(j=0;j<buckets[k];j++){
          a[counter] = k;  counter++;}
  }
```

Fig-2 Bucket sort algorithm

```
source a: { 1 3 7 3 4 }
size of N: 5, size of bucket: 8
// step 1: init bucket array
buckets { 0, 0, 0, 0, 0, 0, 0, 0}
// step 2: build source entropy into buckets
buckets: { 0, 1, 0, 2, 1, 0, 0, 1}
// step 3, generate result from buckets, in ascending order
buckets: { 0, 1, 0, 2, 1, 0, 0, 1}
a:        {     1,    3, 3, 4,     7}
```

Fig3. Bucket sort demonstration

3.2 Parallelization analysis

We focus our interests on loop 2 where unsorted entropy is built into buckets. It contains array in-direct references and not statically parallelizable because the potential data dependencies are carried in the source array and totally depends on its input and distribution.

Loop 2 will be parallelized using PThreads and execute on shared-memory multi-processor machines. We first do a loop blocking such that the original single loop is converted to a nesting of 2 loops, as illustrated in Fig-4. The inner loop was unrolled U times, which is the block size. We use a number of P parallel threads. Each thread is scheduled to execute on the unrolled U iterations as its level of granularity. Transactional memory support is from Vista library [14] which provides a low-overhead interface for transaction commit and recovery. We track memory access on each bucket before any attempt to read and after any attempt to write. If N is not evenly divisible by U, there will be a leftover with size (N – |N / U| * U). Since the leftover is less than the amount of an iteration, we will just use Thread 0 to execute the clean up sequentially.

```
// loop 2 before parallelization
for (k = 0; k< N; k++){ ++buckets[ a [k] ]; }

// loop 2 after parallelization
for (k= 0; k< N; k += U){
  for(i=0; i< U; i++){
    ++buckets[ a [ k * U + i] ];
    }
}
// left over clean up by Thread 0
```

Fig-4 TLS loop-2 parallelization

3.3 Parallelization process

3.3.1 Thread speculation and controls

Fig-5 shows our initial thread speculation and controls after TLS parallelization.

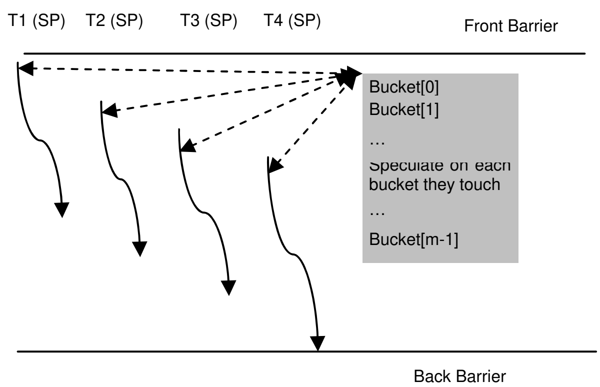

Fig-5 Threads speculation and controls

Assuming we have 4 parallel threads (P = 4,T1 .. T4), each will be scheduled to execute on its respective U unrolled iterations. All threads are created in advance and waited in Front Barrier. Once all threads arrived, they will pass front barrier and start execute its share of iterations in speculative mode. Transactional memory backup happens on individual thread level where each thread is responsible for backup all buckets it is about to access. Upon detection of any conflict as shown in Fig-6, (we will give out details on conflict detection in next section), the thread with higher ID will be marked to abort. All threads will continue to the attempt_to_commit stage. Depends on its state marker, it will then commit or abort. Committing threads will simply have its stored memory released and changes become permanent. Aborting threads will recover from its backup memory and re-execute its iteration.

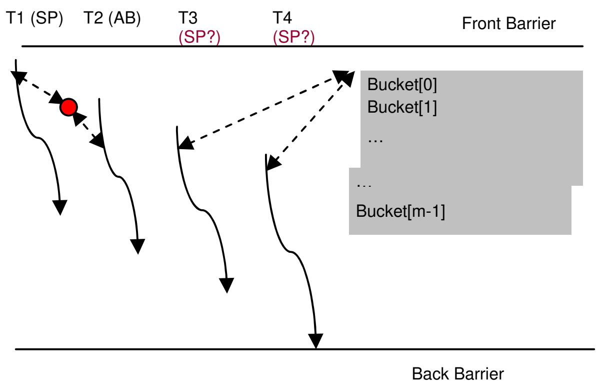

Fig-6 TLS Thread-abort control

This approach seems to be promising to us in the very beginning. It detects conflicts, aborts and restarts the conflicting threads successfully. Unfortunately, it suffers from the problem of not-always sorted result. The reason of this uncertainty is that the aborts only happen among conflicting threads. This information is not propagated to other threads thus they are not aware of the buckets that could potentially have to_be_aborted values. As shown in Fig-6, Thread T1 and T2 conflict on bucket[1]. According to our schema, T2 will be aborted upon reaching the attempt_to_commit stage. However, T3 and T4 both have the potential to access bucket[1] whose speculative value is bad and will be aborted. Unfortunately, there is no way for T3 or T4 to be aware of this situation, thus any attempt on bucket[1] will end up with bad temporary result. This bad result will then be propagated to the end and make the result array unsorted.

There are two potential solutions for this issue. First, we could improve the memory-tracking data structure where an additional marker is placed on each bucket, saying whether this bucket is still reliable for the rest of the iteration. This way, the potential-bad bucket information could be propagated to all other threads. However, this approach will inevitably involve additional overhead. Second, we can make the thread-aborting mechanism more conservation. If there is any conflict in the current iteration, all threads will be aborted. Upon abortion, Thread 1 will resume to execute the aborted iteration sequentially for the iteration. This approach avoids the increased memory-tracking overhead and does not increase the number of aborts. For our testing implements this conservative approach which guarantees the correctness of the results. Fig-7 shows this revised approach.

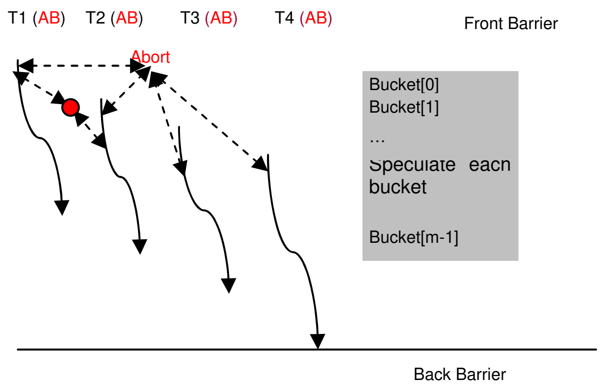

Fig-7 Revised Thread-abort control

3.4 Memory-tracking data structure

T1: Read (0 | 1)
T2: Read (0 | 1)
T3: Read (0 | 1)
… …
TP: Read (0 | 1)

Fig-8 Memory-access record tracking

Fig-8 shows our memory-access tracking logic for each individual bucket. Each of the available P threads has one digit to represent its status for the particular bucket. Before any thread T attempts to read the bucket, its corresponding Read Bit in the above record is marked as 1. After T finished writing the value into bucket, it will try to clear its respective read mark from the tracking record. If the record shows more than one read mark on, a collision is detected and the corresponding thread(s) will be aborted. Rather than Virtual Memory Page based tracking performed by TMK, our memory-access tracking can perform on any granularity we deserve. Currently, the implementation works on 4-byte long integer array on Linux.

3.5 Challenges and Frequently-used techniques

Practicing TLS on source level imposes several challenges unusual to application programmers, including: P-thread management, Transactional management, TLS support and debugging multi-threaded parallel code. We present such problems and give our partial solutions.

3.5.1 Privatization

One of the most frequently used techniques is privatization, where a scalar variable is privatized into a one-dimensional array for make storage private for each thread. Inter-thread communications are simulated as further privatize the one-dimensional array into a two-dimensional array, where threads indexed on rows represent senders, threads indexed on columns represent receivers and data of the array represent messages.

3.5.2 Debugging multi-threaded program

Since our TLS approach only needs to parallelize loop 2, the entire TLS work falls into a single function after parallelization. Multiple threads will concurrently execute within the same function. This makes debugging particular difficult. DBX and DDD understand P-Thread library, but does not work well under this multi-threaded environment. TotalView [14] understands P-Thread library too, but fails to map from executable back to source code. This renders debugging in assemble-language level and is not useful for us neither. We find a reasonable approach is to use the combination of Barriers with conditional printfs on threadIDs. Barriers will make threads synchronize and wait, while conditional printfs will print out the information we are interested for a particular thread. This preserves the parallel executing nature of threads and works well for the rest of the project lacking effective debugger support.

4. Test Setup and Performance Analysis

In this section, we describe our testing environment setup followed with detailed performance analysis.

4.1 Test Setup

We conduct the testing on 2 different platforms: dual-CPU Athlon MP machine running Slackware Linux [2] and 4-CPU Sun box running Solaris 5.8 machine[3]. To explore the effectiveness of TLS parallelism, we render our testing configurations according toTable-1. Each test with a particular parameter is repeated three times and averaged the execution time to leverage potential disruption of workload on the testing machine.

Table 1. Testing Configuration

Parameter	Starting value	Ending value	Increment
P	1	2 * max_CPU_num	1
M	32	2048	*2
U	4	1024	*2

P: Number of Parallel threads

M: range of buckets

U: unroll factor, a.k.a. epoch size

[2] cluster102.ece.toronto.edu

[3] ra.eecg.toronto.edu and seth.eecg.toronto.edu

4.2 Performance on 2P Linux machine

Fig-9 and Fig-10 shows the performance and abort_ratio evaluations on 2-CPU Linux testing.

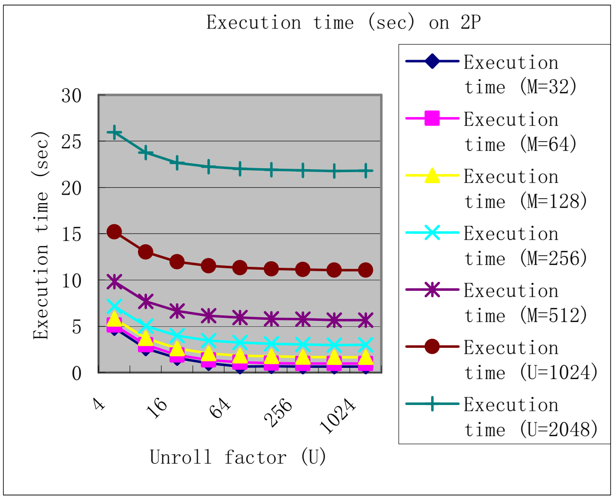

Fig-9 Performance evaluation on 2P Linux machine

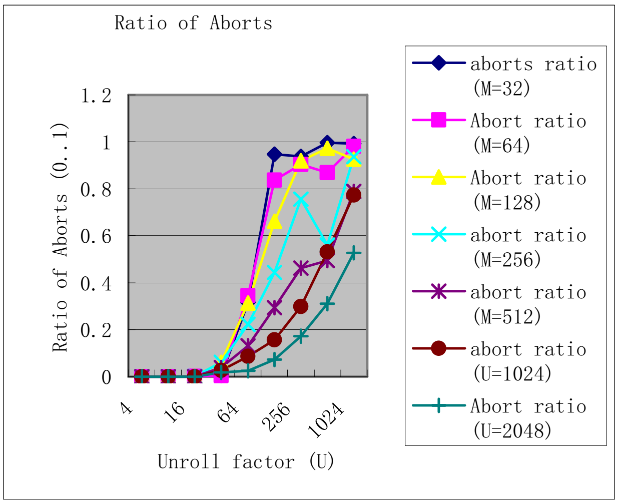

Fig-10 abort ratio evaluation on 2P Linux machine

Fig-9 shows the performance with various factors of U unrolls and bucket size M given 2 parallel threads.

Apparently from the figure, the higher number of unrolls, the better performance. This is because the increased size of U represents the increased granularity for the amount of work threads could execute in parallel. The higher granularity will directly lead to better performance.

Fig-9 also shows that increased bucket size will reduce performance. This is because of our memory-tracking logic. We trace memory change on bucket's level. Linear increment in bucket size will result a linear increment in the size of our memory-tracking structure, plus quadratic increment in the time spent in searching and updating it.

Fig-10 shows the abort ratio with the increment of unroll factor U and bucket size M. It is clear that increment of Unrolls will result a direct increase in the abort ratio. It is because the increased U will directly transfer to the granularity. Threads with higher granularity and the same working set size will have a higher possibility to conflict. When U is close to 2048, the abort ratio is close to 1, which means most threads are conflicting against each other.

Contradicted to our original thinking, TLS achieves its best 2P performance when conflict rate is very high. The optimal configurations are shown in Table-2.

4.3 Performance on 4P Solaris Machine

Performance figures for 4-CPU Solaris are given in Fig-11 to Fig-14.

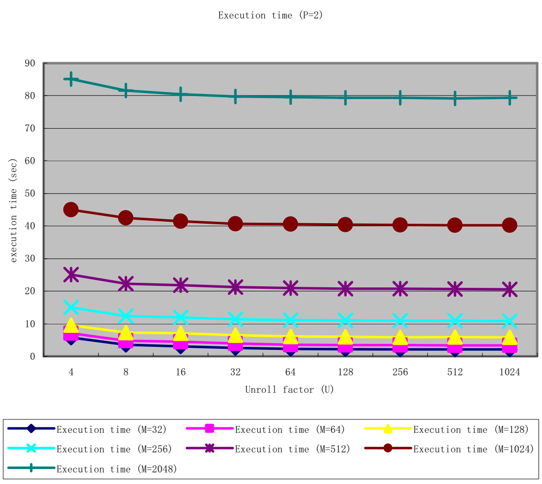

Fig-11 execution time for P=3 on 4P Solaris

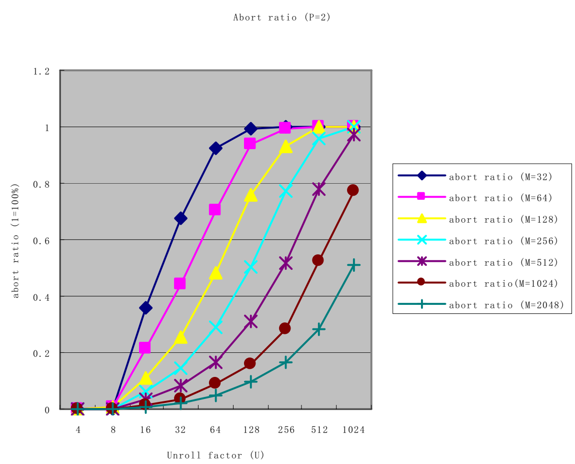

Fig-12 abort ratio for P=3 on 4P Solaris

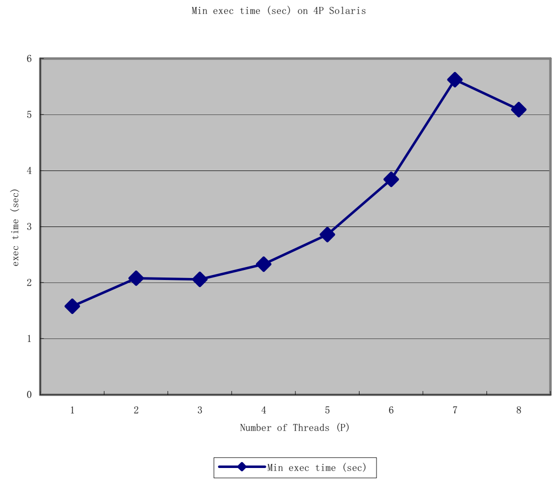

Fig-13 minimal execution time of various P on 4P Solaris

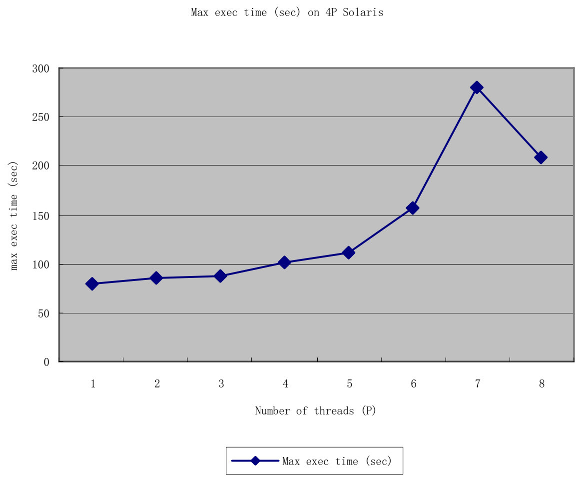

Fig-14 maximal execution time of various P on 4P Solaris

Fig-11 and Fig-12 show very similar behavior comparing with those for 2P Linux. Number of unrolls plays a vital factor in performance. The bigger the unroll number, the better the performance. In addition, the aborts ratio will increase dramatically when unroll factor U increases. However, this increased abort ratio does not translate to performance reduction. Our optimal performance on 4P Solaris (in Table-2) is achieved when abort ratio is actually 100%.

Fig-13 and Fig-14 give execution time for P ranging from 1 to 8 on the 4-CPU Solaris machine. Notice in Fig-13 that when P is less than or equal to 4 (number of physical CPUs), best performance curve is almost flat or sub-linearly. When P is over the number of physical CPUs, the execution time will increase super-linearly. Fig-14 of the max execution time curve shows the same trend. It roots in the way parallel threads are scheduled. When the number of threads is less than the number of CPUs, each thread would ideally be scheduled to a different CPU, maximizing potential parallelism. When the number of threads overreaches the number of CPUs, there is no way to schedule all these threads on different CPUs, thus multiple threads will be scheduled on the same CPU, only accumulating overheads.

Fig-14 also shows that the maximal execution time for each configuration of P is un-believably long. The performance gap between best and worst configurations of the same P could be around 2 degrees of magnitude. It strongly suggests that granularity and bucket partition play a vital factor in getting the best parallel performance for a given number of parallel threads.

4.4 Performance on 8P SGI Origin Machine

Since PThread did not work on 8P SGI Origin[4], we instead re-write our PThread programs using ANL[5].

When exhaustively searches through the entire empirical configuration space, 8-CPU SGI Origin machine shows similar behaviors when comparing to 2P Linux and 4P Solaris. The similar figures are given in Fig-15 and Fig-16.

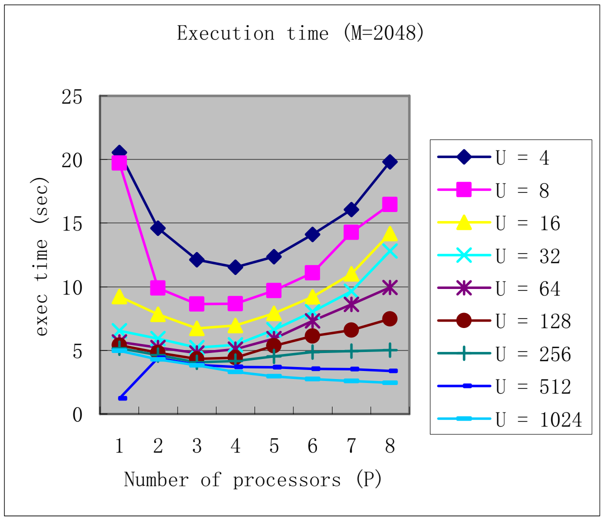

Fig-15 8P SGI performance with fixed bucket size M = 2048[6]

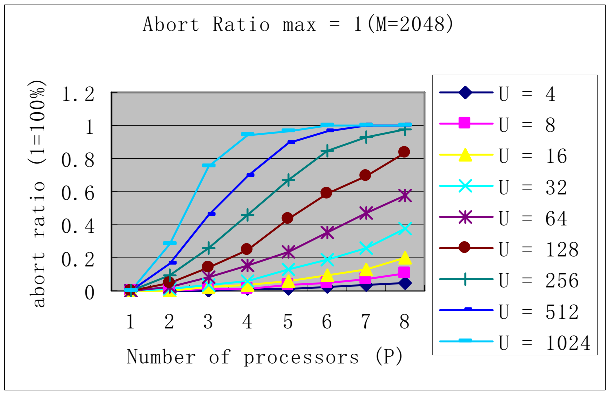

Fig-16 8P SGI abort ratios (1=100%) with fixed bucket size M = 2048

As shown in Fig-15, execution time on 8P SGI has a parabolic-like curvature when having M fixed to 2048 (simulating Papadimitriou [14]'s configuration with 1-additional configure parameter U floating). Performance increases when U increases and peaks when P = 4. When U increases, the granularity gets larger thus the overall performance increases. Since the increment in the number of ANL processes will increase the level of parallelism at the expense of synchronization overhead, P=4 seems to be a good balancing point.

Fig-16 shows the abort ratio diagram on 8P SGI. It is very similar to Fig-10 for 2P Linux and Fig-12 for 4P Solaris.

---

[4] The quartet.eecg.toronto.edu is a SGI Origin machine running IRIX 5.3 and no additional OS upgrades will be possible. SGI's PThread support starts with IRIX 6.2. Current version lacks kernel support, thus third party PThread library (e.g. MIT PThread open-source library) will only do context switching for PThreads, rather than scheduling them to run on different CPUs.

[5] ANL Parallel Processing Macro, introduced at http://www.google.ca/search?q=cache:24jBzD3ZeqkJ:www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15740-f98/public/asst/asst5/ANL_tutorial.pdf+ANL+Parallel+Processing+Macro&hl=en

[6] Process creation overhead has been removed when doing the timing analysis

The increased epoch size will have a higher possibility for different processes to collide, and this tread increases when with the number of parallel processes.

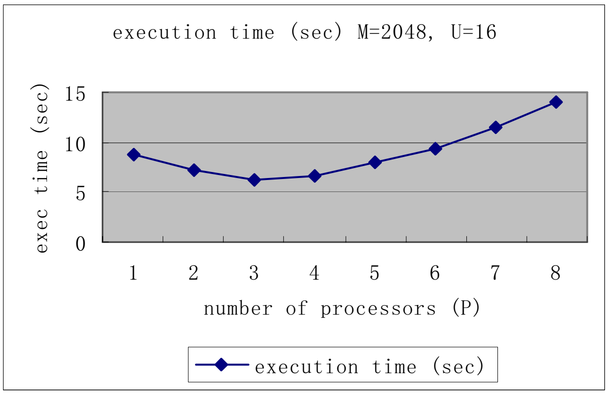

Fig-17 8P SGI performance when M=2048, U=16

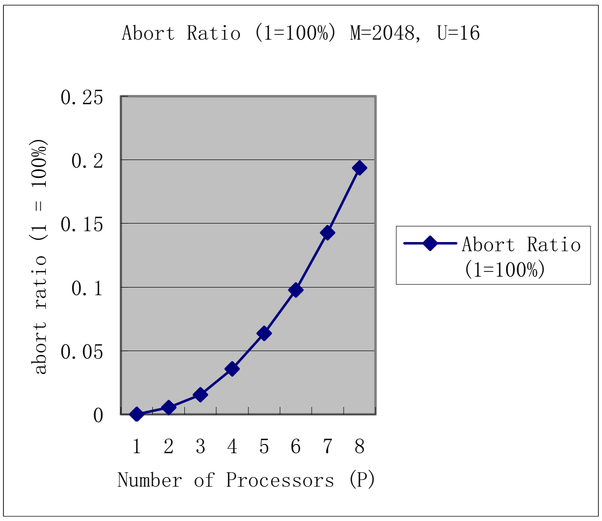

Fig-18 8P SGI abort ratio when M=2048, U=16

Fig 17-18 present a particular configuration on 8P SGI with both M fixed to 2048 and U fixed to 16.

Fig-17 shows similar behavior given in Fig-15, as performance peaks when P=4. We give Fig-18 to explicitly demonstrate the abort ratios when local performance peaks. It is not necessarily to have a very high abort ratio for the peak performance. In this case, the abort ratio is roughly 4% for local optimal M=2048, U=16 and P=4. Unfortunately, when abort ratio is usually very high when globally searching through the empirical configurations.

4.5 Best configurations and 4P Solaris profile analysis

Table-2 Best configurations for global optimal performance

Number of Max Physical CPU	M (Number of Buckets)	U (Unroll factor)	P (Number of optimal threads)	Best execution time (sec)	Abort ratio (%)
2P Linux	32	1024	2	0.629	99.2
4P Solaris	32	512	3	2.065	100
8P SGI	128	512	8	1.96	100

Table-2 shows the optimal configurations for 2P Linux, 4P Solaris and 8P SGI platforms where best performance is achieved after globally searching through the empirical configurations. Since the behaviors on 2P Linux and 4P Solaris are quite similar, we next give a detailed analysis of profiled data for the configuration achieving optimal performance on 4P Solaris in Fig-19[7].

.

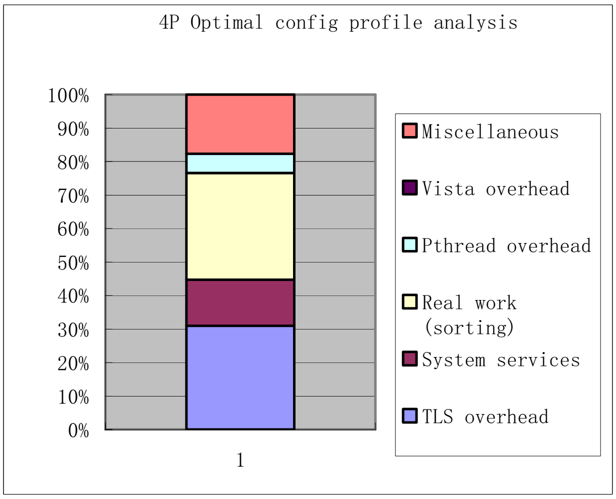

Fig-19 4P Solaris optimal configuration profiling

First, TLS overhead occupies a significant portion of the entire execution time – 31%. This is largely due to the checking for the number of read marks in detecting collision. Since this checking happens sequentially and frequently (each thread needs to check it after any bucket

[7] Profile-data analysis on8P is not given because profiler is not installed on the SGI machine.

update), there would be enough room for future improvements.

Contradicted to our original belief, PThread overhead is very low (5.7%), with 3.4% for mutex_lock and 2.3% for mutex_unlock respectively. Overheads involved in PThread create, destroy, self, barrier and synchronization management are all ignorable. Since PThreads lack the ability of explicit inter-thread communication, such communication overhead has been counted into the major sorting work after privatizations.

Vista library is proved to be a low-overhead interface. We detected ignorable 0% of vista overhead in the optimal 4P configuration. This is because the revised collision detection and recovery approach only needs to backup buckets memory once per iteration. Further optimize vista library seems to be not useful in our testing case.

For the overall best performance among the testing platforms, we achieved optimal 0.629 sec on 2P Linux machine vs. 0.059 sec for the original sequential implementation tested on the same platform. The slow-down gap is roughly on the range of 8 – 15 times.

4.6 Future Open Areas of Research

Work in this project identified the following open research topics directly related to software-only TLS.

4.6.1 Epoch size

Epoch size, a.k.a. granularity that each parallel thread performs its task on, is critical for optimal performance. All of our testings show that increased granularity will lead to improved performance. However, it is not always possible to increase granularity in real applications. Further, when considering cache hierarchy, we would prefer to put speculative data into L1 or L2 caches for all threads, thus there would be an upper limit for the size of epoch. The optimal value of the size remains an open issue.

4.6.2 Thread scheduling and communication

Our testing shows good load balancing because of the random distribution for the generated bucketsort source data. However, general parallel threads do not necessary have good load balancing due to the nature of its respective trace footprint and data locality. Since all threads are synchronized within the same iteration, it would be necessary to schedule threads across this iteration barrier for better performance. Inter-thread communication is difficult because PThread interface is flat and there is no way to send a message from thread1 to thread2 directly. Currently, we use a passive communication model where each thread checks its messages at attempt_to_commit stage and acts accordingly. Improvement in this issue would possibly need a revise on the PThread programming model.

4.6.3 Number of Parallel threads

As shown from our testing, having more threads than the number of physical CPUs is undesirable, as some extra threads would have to be scheduled on the same CPU thus accumulating additional overheads. However, when CMP becomes available, we might have shared-memory multi-processors built on top of CMPs. There would be a huge difference between scheduling threads within different cores in the same CPU and scheduling them across physical CPU boundary. Finding an optimal maximal thread number would need further investigation.

4.6.4 Collision number and rate

Our testing seems to be insensitive to the number of collisions and its ratio. This might be due to the fixed number of unsorted elements. When U increases, the number of iterations will decrease. Thus increased abort ratio will actually result in decreased number of collisions. However, in general-purpose applications, this might not be the case, where increased number of conflict ratio might actually result in increased number of collisions. How will such programs behave is an issue remain to be seen.

4.6.5 Speculative Region

Out testing only focus on TLS parallelizing loops, thus no representation of other parts of the program's constructs. Oplinger et al. [7] suggests that it is not sufficient to parallelize loops only and proposed to extend TLS regions over functions and other program constructs. This is an interesting topic as the extended approach will result in an increase in epoch size. However, such attempts will need to have relatively accurate functions' return-value predictions in order to maintain a low abort count. Finding an efficient and relatively accurate prediction algorithm would be crucial for further advancement in this direction.

4.6.6 Transactional overheads

Transactional control overhead is ignorable in our testing, because the revised pessimistic thread-abort scheme will only need to backup speculative data once in the front barrier per iteration. However, this pattern could be significantly different for general-purpose applications where memory backup is frequent and backup memory is fragmented. It would be necessary to optimize Vista library to reduce some transactional control overheads.

4.6.7 Parallel programming model and tools

PThread is one of the most frequently used Parallel programming diagrams. It also serves as the compilation target of OpenMP. However, PThread is provided as a set of plain APIs. This lack of thread control structure makes thread communication and value optimization difficult. TLS requires explicit and efficient inter-thread communications and synchronizations. A possible approach is the Object-Oriented Programming Model (e.g.

Java Threading architecture) where threads are represented as objects rather than through plain APIs. Message passing and thread's control are explicit and easy, but the overheads involved in OOP model could be prohibitive. It remains an open issue on how to construct such an efficient control model with low overhead, as well as effective parallel programming and debugging tools.

4.6.8 Source-level Software-only TLS is feasible

As shown in our testing, Source-level Software-only approach for Thread-Level Speculation is feasible. We can guarantee correctness with suggestions on further performance improvements. However, the increased complexicity of TLS control, parallel debugging combined with transaction support could render the task unattractive even for experienced application programmers. Thus we suggest such work be automated by compilers and leave programmers a set of pragmas as hints and inputs for compilers to perform TLS parallelization.

4.6.9 Enhanced hardware support is coming

Another motivation of this work is the confirmed product delivery from both Intel and AMD, where both companies promised to deliver dual-core and even multi-core CPUs beginning in 2nd half of 2005[8]. It would be interesting to see what kind of supports have been built into such products, and how our software-only approach could evolve and utilize such hardware enhancements and reduce software overheads.

5. Conclusions

In this paper, we studied Source-level Software-only approach for Thread-level speculation. TLS makes it possible to parallelize loops where static compilers failed to parallelize. Software-only approach removes the dependency on hardware extensions and source-level approach reduces the time and efforts needed for adding tedious compiler-back supports. We focus our efforts on TLS parallelizing loop 2 of bucket sort. Initial testing result is promising. We reduced the performance gap from 150 times slower [12] to within one degree of magnitude, plus a few promising approaches proposed for further performance improvements.

6. REFERENCES

[1] Zhai A., Colohan C.B., Steffan G. and Mowry T., Compiler Optimization of Scalar Value Communication between Speculative Threads. Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002, San Jose.

[2] Steffan G., Colohan C.B., Zhai A. and Mowry T., *Improving Value Communication for Thread-Level Speculation,* Proceedings of the Eighth International Symposium on High-Performance Computer Architecture, February 2002, Cambridge.

[3] Steffan G., Colohan C.B., Zhai A. and Mowry T, *A Scalable Approach to Thread-Level Speculation*. Proceedings of the 27th International Symposium on Computer Architecture, June 12-14, 2000, Vancouver, British Columbia, Canada.

[4] Steffan G. and Mowry T., *The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization*. Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, February 2-4, 1998, Las Vegas, Nevada.

[5] Steffan G., *Hardware Support for Thread-Level Speculation, Ph.D. thesis,* Technical Report CMU-CS-03-122, School of Computer Science, Carnegie Mellon University, April 2003.

[6] Oplinger J. and Lam M.S., Enhancing Software Reliability using Speculative Threads, Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, October 2002

[7] Oplinger J., Heine D. and Lam M.S, In Search of Speculative Thread-Level Parallelism, Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99), Newport Beach, CA, October 1999

[8] Oplinger J., Heine D., Liao S.W., Nayfeh B.A., Lam M.S. and Olukotun K., Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor, Stanford University Computer Systems Lab Technical Report CSL-TR-97-715, February 1997

[9] Hammond L., Carlstrom B., Wong V., Hertzberg B., Chen M., Kozyrakis C., and Olukotun K., Programming with Transactional Coherence and Consistency (TCC), Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, October 9-13, 2004

[10] Hammond L., Carlstrom B., Wong V., Hertzberg B., Chen M., Kozyrakis C., and Olukotun K., Transactional Memory Coherence and Consistency, Proceedings of the 31st Annual International

[8] Sun Micro Systems has similar plan announced. PowerPC has dual-core chips available on the market place already.

Symposium on Computer Architecture, München, Germany, June 19-23, 2004

[11] Cintra M. Martinaze J.F., Torellas J., Architectural support for scalable speculative parallelization in shared-memory multi-processors, in proceedings of ISCA 27, June 2000

[12] Papadimitriou P, Mowry T., Exploring Thread-Level Speculation in Software: The Effects of Memory Access Tracking Granularity, CMU Technical Report, 2001

[13] Keleher P., Dwarkadas S., Cox A.L., and Zwaenepoel W., TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proceedings of the Winter 94 Usenix Conference, pp. 115-131, January 1994

[14] Lowell D. E. and Chen P., Free Transactions with Rio Vista, *16th ACM Symposium on Operating Systems Principles, October 1997*