Toward Thread-Level Speculation for Coarse-Grained Parallelism with Regular Access Patterns *

Ravi Ramaseshan and Frank Mueller

Dept. of Computer Science, Center for Efficient, Secure and Reliable Computing,
North Carolina State University, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

Abstract. Recent work on transactional memory (TM) bears promise to exploit multicore capabilities. TM extensions for thread-level speculative parallelism (TLS) have predominantly focused on integer benchmarks with short critical sections and exploit limited on-chip buffering space to store shadow values needed to potentially abort transactions. In contrast, scientific codes generally provide coarse-grained parallel regions with potentially shared memory accesses, which do not fit into size-limited shadow buffers. Hence, such codes represent a mismatch for TM-TLS.

This work contributes mechanisms to speculatively parallelize scientific codes with dense, non-scalar data references exploiting compilation techniques and runtime enhancements coupled with minor hardware enhancements to transparently support TLS.

A method to efficiently detect access violations to shared memory in speculatively parallelized regions is developed, much alike TM, yet with data footprints of arbitrarily large size. The mechanism for violation detection is based on runtime software and optional hardware support to efficiently capture regular access traces. Experimental evaluations assess the speculation overhead in presence and absence of access violations considering an environment with and without hardware support. The results show that this method is competitive to explicit parallelization or auto-parallelization, yet can be applied even when data dependency checks remain inconclusive at compilation time.

1 Introduction

Contemporary computer architecture design has shifted from pushing processor frequencies to increasing coarse-grained parallelism, mostly packaged as chip multiprocessors (CMPs) or multi-cores. While CMPs are projected to support 32 cores per chip or more, programming models for concurrency (multi-threading) and their interaction with hardware are being revisited. To exploit the potential speedup on these systems, course-grained concurrency within a program is a prerequisite. In this context, automatic and semi-automatic tools that help in identifying and exploiting concurrency are of grave importance.

On the Limitations of Auto-Parallelization: For scientific programs, statically provable *i.e.*, automatic parallelization under the compiler's direction have been successful for regular, well-structured, loop-nest-dominated codes, typically written in

* This work was supported in part by NSF grants CCR-0237570 (CAREER), CNS-0410203, CCF-0429653 and a grant by the Humboldt Foundation.

Fortran. However, cases such as — aliasing of variables, input-dependent execution patterns, complex control flow and use of third-party, binary-only libraries make it hard for the compiler to *prove* that a loop nest can be safely parallelized. Researchers have attempted to overcome these challenges using increasingly complex algorithms involving interprocedural analysis [8], abstract interpretation [23] and advanced symbolic analysis [5]. However, most of these attempts are very computationally expensive and can only handle benchmarks written in Fortran.

Thread-Level Speculation (TLS): TLS attempts to address scenarios where the compiler cannot statically prove the safety of parallelization. By speculatively parallelizing a loop, typically by executing fixed-sized chunks of the inner-most loop in parallel, performance can potentially be dramatically improved. However, if speculation fails, *i.e.*, data dependencies are actually violated by this chunking approach, the effect for execution has to be reverted to the state prior to speculation. For this reason, buffers are employed by TLS schemes to speculatively overwrite memory and also to allow for detection of dependence violation between speculatively executing threads. Most hardware TLS proposals support this by modifying the cache protocol. Examples include STAMPEDE TLS from CMU [26], Hydra TLS from Stanford [10] and the TLS machine from UIUC [15]. For each cache line, additional space is reserved to save the memory overwritten speculatively. Extra states in the cache protocol indicate if the cache line is speculative. Invalidations to speculative cache lines invoked by a less-speculative thread indicates violation of a *true* dependence (*i.e.*, a read-after-write dependence), causing the more-speculative thread to be squashed and restarted. TCC (Transaction Coherence and Consistency / Stanford [9]) instead proposes to buffer the entire speculative state inside the processor. At the end of speculative execution, speculative changes are broadcast to all processors to check for a dependence violation.

Software speculative parallelization proposes to maintain speculative state entirely in software. The LRPD test from Rauchwerger *et al.* [22] and Rundberg *et al.* [24] maintain *shadow* memory for all potential arrays that can participate in speculation. Additional state at the level of individual data elements (*e.g.*, a single array position) supports dynamic memory renaming, speculative privatization and reduction. As with past hardware TLS proposals, these schemes still maintain speculative information at a fine-grained level. They are much slower than hardware TLS and require large amounts of memory to serve as shadow memory. Moreover, they depend on the *compiler* to save and restore potentially overwritten memory state and to allocate and manage the shadow memory.

Transactional Memory (TM) Design: In response to the increasing on-chip parallelism, academics and chip designers have been exploring the benefits of TM as an alternative to protecting data with locks (or semaphores). Transactions are shown to be well-suited for small critical sections when concurrent accesses to the same shared data are the exception, which is the case for integer and client/server workloads. Past work on parallelization for TM has focused on bounded transactions in terms of data size with extensions to unbounded protocols [11]. The former are common in hardware TM while the latter dominate in pure software solutions or hybrids. In high-performance computing (HPC), software transactional systems are not an option due to their cost for maintaining shadow space and, optionally, versioning in memory. A Hybrid approach

provides a fast solution while transactional state remains on chip (*e.g.*, in transactional caches) but becomes slow when spilling into main memory. Extensions of TM for TLS have been proposed in Stanford's TCC, which provides a transparent method to speculative parallelize loop nests, as discussed before.

Critique of Past Work: The key weakness with most past approaches (one exception being TCC) is the reliance on fine-grained cache-line level speculation state management. As the *size* of the speculative region grows, the number of speculatively written cache lines may increase beyond the capacity of the cache. Evicting a cache line in a speculative state causes the speculation to be aborted. In case of TCC, the limiting factor is the size of the in-processor write buffer that stores the speculation state; the speculation is aborted if this write buffer overflows.

Due to these design limitations, the size of the speculation region is necessarily small. There are several downsides to this. First, the overhead of speculation management instructions (*e.g.*, spawning a speculative thread, waiting to commit) can be significant. Past studies report overhead ranging from 3.7% to 30.6% [26]. Moreover, due to fine-grained speculation, there is contention between processors for the same cache lines, resulting in cache misses. Past work reports that anywhere from 1.67% to 65% of the cache misses in speculative execution accessed data that was in the other processor's cache [26]. Both these factors have a detrimental effect on the net speedup obtainable with speculation.

More critically, the limitation to small regions typically forces speculation to be only applied to the innermost loop, lest the speculative data overflow the cache and cause an abort. Kejariwal *et al.* [13,14] report only 1% gain of TLS over explicitly threaded codes or hardware enhancements for Spec 2006 for inner-most loops, which are extremely constrained assumptions. In practice, the working sets of scientific programs can be much larger than the size of the cache. This restriction to innermost loops may disallow speculative parallelization in cases where the inner loop cannot be parallelized. And the cost of thread creation and speculation can be much lower than in their study if assisted by hardware.

Consider a doubly nested loop with a dependence given by the direction vector (=,<). In this case, the outer loop can be parallelized, but the inner loop must execute sequentially. In fact, even a dependence of (<, <) can be speculatively parallelized by our approach if actual access traces reveal that a potentially carried dependence of the outer loop, as determined statically by the compiler, does not occur dynamically, effectively reducing it to (=,<).

Contributions: In contrast to prior work, we propose to explore the potential of speculative parallelization beyond inner-most loop combined with low-cost dependence violation detection and memory shadowing. This is not possible with the limited speculation regions of current TLS proposals.

Our focus is on scientific codes, which are a poor match for TM-TLS systems as such systems provide large, parallelizable regions with many non-scalar accesses that are potentially shared. This approach exploits the fact that scientific programs tend to be dominated by regular and predictable access patterns. Instead of maintaining fine-grained speculation information per cache line, we develop *compressed* representations of memory access patterns that are suitable for dependence checking without decom-

pression. Our idea improves over past TLS work by allowing potentially orders-of-magnitude larger speculation regions than possible with a cache-line centric approach.

To this extent, we have developed a pure software implementation for coarse-grained speculative parallelization leveraging past work on compressing regular access patterns. We shall leverage this work to implement trace compression in software and to evaluate the benefits and limitations in an experimental environment. These experiments distinguish the cost of (a) memory shadowing, (b) online trace generation and compression, (c) dependence violation checking and (d) actual execution for the parallelized loops. These results allow us to assess the benefits for a hypothetical hybrid hardware-software thread-level speculation scheme. Through simple and low-cost hardware extensions, regular access patterns can be recognized and compressed in hardware. A set of dependence tests can subsequently be executed either in hardware or in software depending on their respective complexity. Overall, our method can be more aggressive in parallelization than existing TLS techniques, yet is shown to be competitive in overhead to explicit parallelization or auto-parallelization. It can be applied even when data dependency checks remain inconclusive at compilation time or allow only TLS at the inner-most loop nest.

2 Design and Implementation

In a nutshell, our TLS design can be described as follows. First, the program is run with a small profile data set. Instrumentation (in hardware or software) observes if loop-carried dependencies exist during the execution of the target loop nest. If no such dependencies are seen, the compiler inserts code to *speculatively* parallelize the loop nest on a multiprocessor machine, and the program is run with the full data set. For the speculatively parallelized loop nest, each processor executes its assigned chunks of iterations in parallel. Hardware or software mechanisms ensure that if the speculation fails, the state of the machine (registers and memory) can be restored to a safe state and re-executed serially. If speculation failure is infrequent, the program may achieve a significant speedup in execution with this scheme.

Figure 1 shows the details of our proposed speculative parallelization scheme consisting of an analysis step that identifies potential speculative parallelization candidates and the actual execution step.

In the analysis step, all loops of an application are inspected by the compiler's auto-parallelization framework, a modified version of ORC [1, 2, 28]. Loops that do not contain definite loop-carried dependencies are marked as potential candidates for speculative parallelization. The compiler activates instrumentation for tracing memory accesses and the loop iterations of such loops. This program is run with a small data set. For each speculation candidate loop-nest, the corresponding dependence graph is rebuilt using the trace information.

Subsequently, all loops without a loop-carried dependence in this trace-based dependence graph are marked for speculative parallelization. The program is now run with the actual data set. The program executes non-speculatively till it encounters the beginning of a speculative region. Within this region, one or more speculative loops may execute in sequence. Each of these loops is parallelized, enhanced by trace compression capabilities to generate power regular section descriptors (PRSDs) and, on exit from the

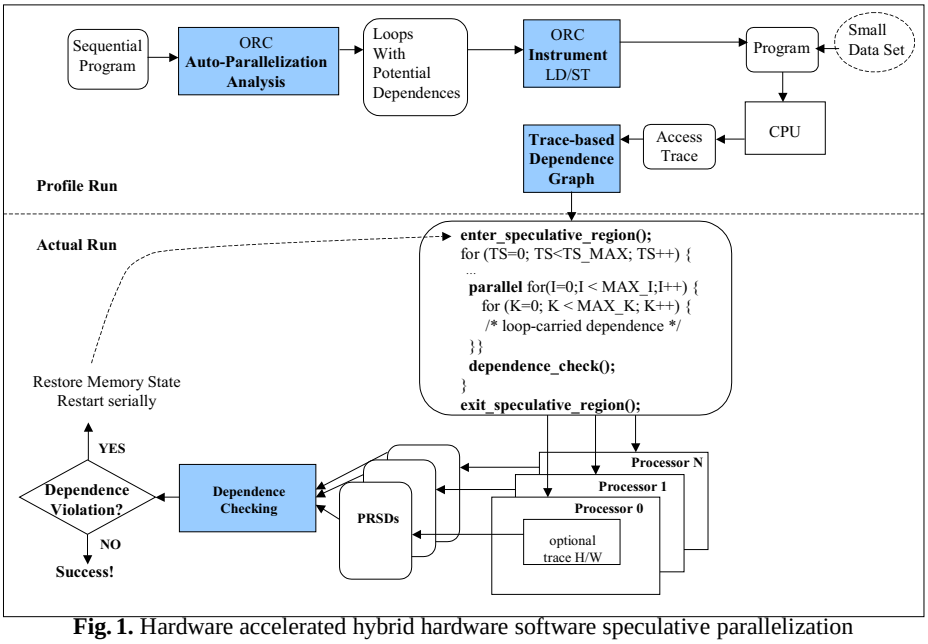

Fig. 1. Hardware accelerated hybrid hardware software speculative parallelization

parallelized loop, by a dependence violation test. The constraint on the speculative region is that it must fully enclose any speculative loops, yet it may begin at a higher loop level than the speculative loops themselves. In Figure 1, this region encloses a coarser timestep loop that contains the parallelized loop while the dependence in the loop body could be carried by either of the inner two loops. The components of the framework are further elaborated in the following.

Analysis Step

We have enhanced the ORC compiler to selectively instrument loops with potential dependencies [21]. The instrumented application is then executed on a uniprocessor using a small input as a training run. Accesses to memory and loop entry/exit points are subsequently traced exploiting an online compression scheme to generate PRSDs as outlined below. This access trace is utilized to break dependencies in the compiler-generated dependence graph that actually did not occur at runtime. The modified ORC compiler then generates a speculative program from this relaxed dependence graph, as outlined in Figure 1.

Software Tracing

Software tracing occurs during the execution of the speculatively parallelized program. Accesses are traced on a per-thread basis to determine if data dependencies that actually occur during parallel execution are being violated. We represent the memory access pattern seen by the hardware at runtime using Power Regular Section Descriptors (PRSDs) [17, 18]. PRSDs represent hierarchically nested memory accesses such as those frequently generated by access points in loop nests. A PRSD is described by the following recursive production rules:

PRSD→<Base_Address, PRSD_Body>

PRSD_Body→<Stride, Length, PRSD_Body> | ϵ

Base_Address is the first address described by the PRSD. Each PRSD describes Length number of instances of a child PRSD (PRSD_Body), each of which is strided at a distance of Stride. The child PRSD may itself contain children PRSDs. In this manner, PRSDs can efficiently describe regular accesses that occur in deeply nested loops.

For example, consider the following code, assuming row-major layout and array elements of 8 bytes and matrix dimensions of 2000 square.

```
for(i=0;i < 2000;i++)      PRSD_A: <Base_A,<16000,2000,<8,1000>>>
 for(j=0;j < 1000;j++)     PRSD_B: <Base_B,<8,2000,<16000,1000>>>
  A[i][j]=B[j][i]+C[i][j]; PRSD_C: <Base_C,<16000,2000,<8,1000>>>
```

The inner j-loop initially accesses 1000 consecutive elements of matrix A of row 0 represented concisely as RSD $\langle$Base_A[i],$\langle$8.1000$\rangle\rangle$ for $i = 0$, *i.e.*, the first iteration of the i-loop. For $i = 1$, 1000 consecutive elements of A are accessed again, yet, this time in row 1. Effectively, elements 1000-1999 of row 0 were skipped (not accessed). Hence, the offset between A[0][0] and A[1][0] is calculated as 16.000 bytes, and PRSD_A is created to denote both the 1000 accesses of the j-loop (as seen in the RSD above) within the 2000 iterations of the i-loop and a 16.000 bytes offset for the array accesses between consecutive iterations of i. Thus, when the code shown on the left is executed, the 3 PRSDs shown on the right are created on-the-fly as the access stream is generated. In this case, 6 million accesses are represented as a 3-level PRSD requiring less than 100 bytes of memory. More significantly, the compressed representations (PRSDs), generated within each thread, are still in a format that can be analyzed to determine if data dependencies were violated.

Optional Hardware Tracing

Software tracing imposes a considerable overhead on application performance, as quantified in the experimental results. Instead, traces may be generated in hardware building on existing architectural features. PRSDs are generated for each access point incrementally using a constant amount of working memory and a bounded number of operations per memory access. We propose a constant-size trace buffer associated with the performance monitoring unit (PMU) and the prefetch unit. PRSD generation in hardware then amounts to (1) recognizing strided accesses and (2) employing a logic for monitoring/logging memory references. Hardware prefetchers already realize (1) for short stride lengths. Performance monitoring units already implement (2) as trap-based facilities to capture latency-constrained memory loads within the pipeline (Itanium-2), through precise event-based sampling (Pentium) and instruction-based sampling (AMD Barcelona). We supplement these features by a fixed-size buffer to store PRSDs of approximately 30 entries (*e.g.*, for 6 non-scalar references in 5 loop nests) and the logic to detect/extend PRSDs as striding references are encountered. Upon buffer overflow, outer-most PRSDs could be spilled to memory, as is the case with current PEBS functionality. When leaving parallelized regions, the entire buffer is explicitly spilled. The PRSD algorithm is described in detail by Marathe *et al.* [16, 18]. Overall, additional circuit requirements should be equivalent to a small subset of existing chip area for performance monitoring, but the details of area and power implications are beyond the scope of this paper. The objective of this paper is to assess the remaining performance impact of our TLS scheme when tracing is supported in hardware.

Dependence Testing

When the program starts executing the speculatively parallelized loop, the loop iterations are partitioned by multiple threads running on different processors. Access tracing (in software or hardware) results in the on-the-fly construction of PRSDs, which are flushed when necessary to a memory buffer private to each thread. At the end of speculative execution, the PRSD traces from each thread are inspected to determine if data dependencies were violated by the speculative parallelization.

The *single-bounds interval test* determines if regions of accesses from different threads are disjoint. To this extent, PRSD generation is augmented by tracking the upper and lower watermark for a given access point, expressing them as a single interval for the entire PRSD. By arranging intervals of watermarks in an AVL tree, the region overlap test has a complexity of $O(n \log n)$ where n is the total number of intervals over all threads, *i.e.*, n insert operations on the AVL tree are required, each of which may take up to $\log n$ comparisons. If the tree is traversed to determine the location of a new interval i and an existing interval j originating from a different thread is encountered that overlaps in range, a potential data dependence is found. (Notice that intervals need to be tagged with the last thread ID that performed an insert on the respective range to determine the origin of intervals in terms of threads.)

This check requires implementation in software due to its algorithmic complexity. The overhead of the dependence check depends on the number of threads, t, and the number of PRSDs generated per thread, p. The latter can be affected by limiting the scope of speculation. The former can be influenced by parallelizing the dependence check in which case the serial complexity of $O(n \log n)$ for $n = p \times t$ (where p is averaged over all threads) can be reduced to $O(\log p + \log t)$ [19], but this optimization is beyond the scope of this paper. Our framework supports the specification of a set of dependence tests, but details are omitted here because additional tests have not been implemented yet.

If no potential dependence is found and speculation succeeds, then the results of computation are immediately available, *i.e.*, unlike some TM-TLS proposals, there is no "commit" overhead. If speculation fails, then the system is restored to a safe state before entering the speculative region as explained next.

Entering Speculation Regions

In the process of speculative execution, memory locations are being modified. The content of these locations needs to be saved since modified values require recovery upon failed speculation. Past approaches leveraged cache shadowing techniques or maintained fine-grained copies of old values in memory. Shadow caches limit the size of speculation while fine-grained shadow memory, sometimes also employed when shadow caches overflow, may impose significant overhead due to on-demand copying at data-type granularity upon each and every write operation.

Our scheme exploits virtual memory with the copy-on-write mechanism inherent to process creation. This has the advantage that old values are preserved through bulk transfers into a shadow memory region, which is more efficient than on-demand fine-granular copies and does not impose limitations on the scope of speculation as shadow caches do. When entering the scope of a speculative region (or even at a coarser level), a child process is forked and immediately suspended. As a side effect, the operating

system (OS) marks the pages of parent and child as read only. When a page is written to (in the parent), a page protection fault is caught by the OS. Subsequently, a copy of the page is created in physical memory and mapped onto the same virtual page as before. At this point, write access to this virtual page is enabled for parent and child so that, upon return from the OS, the write is re-issued in the parent. Upon successful speculation, results are already available in the parent while the child process is simply killed. If a potential violation is detected, the parent is aborted (killed) while the child is activated and resumes with sequential execution from the point of entry to the speculative region. Since the child is privy to the memory state prior to speculation, correct results are guaranteed.

3 Experimental Results

Our experiments were conducted on a dual processor 900 MHz Itanium™2 machine running Linux. We configured our benchmarks to be compiled for two threads. We used ORC version 2.1 for our framework with customized extensions (such as the OpenMP runtime system and some enhancements within the parallelization framework at the HIR level to be able to compile and parallelize the benchmarks subject to evaluation).

For our evaluation, we selected five benchmarks from a modified version of the NAS Parallel Benchmark [4] suite, a widely used suite to evaluate the performance of parallel computers. Specifically, we use the NAS (OpenMP C) parallel benchmark suite (version 2.3) by the Omni OpenMP Compiler group for our experiments. The source code of each benchmark was modified by removing all OpenMP directives from them. In addition, we also replaced the complex number implementation used in the `FT` benchmark to use that provided by the C Standard Library. This modification was required to circumvent a deficiency in our branch of ORC that performs reduction analysis on structure elements incorrectly. The benchmarks were compiled at the highest level of optimization (`-O3`). In our research, we are focusing on do-all rather than do-across auto-parallelization. Hence, we used the do-all `-apo` flag and turned do-across auto-parallelization off. We also switched inter-procedural analysis and binary dead-code elimination off to bypass further bugs in our branch of ORC.

Potential of Speculative Parallelization

To analyze the potential of our speculative parallelization framework, the benchmarks were traced and analyzed offline using the class S data set and the speculation experiments were run using the class A data set. In order to estimate the potential of speculative parallelization, we compare the execution times of the conservatively parallelized program against that of the speculatively parallelized program without the speculative runtime overheads of tracing, PRSD compression and dependence violation checking.

The objective of the first experiment is to study the capabilities of our speculative parallelization framework. The goal of speculative parallelization is to simply turn it on regardless of potential violations due to data dependencies. Hence, we are assessing here if speculative parallelization can achieve at least as good of a result as compile-time parallelization for codes that are known to be statically analyzable in terms of data dependencies.

Table 1 summarizes the results and shows that our speculative framework is competitive with compile-time parallelization. Not only that but the `LU` benchmark shows

potential beyond conventional compile-time parallelization as speculation uncovers additional potential for savings. Further analysis of the benchmark codes further indicates additional potential for speedup by the speculative framework if a coarser level of parallelization (with more work per thread) was chosen at an outer-more loop level.

Table 1. Limit of Speculative Parallelization

NPB	Wall-clock time (seconds)		Speedup
	Static Parallel	Speculative Parallel	
BT	1007.37	1009.00	0%
EP	119.50	120.46	0%
FT	50.35	50.30	0%
LU	406.42	371.60	9%
SP	357.20	357.44	0%

Speculative Execution Overheads

The objective of the second experiment is to distinguish additional costs due to our software speculative framework and discuss their impact under the assumption that partial hardware support exists (particularly for PRSD trace generation). We then have to distinguish the cost of (a) memory shadowing, (b) online trace generation and compression, (c) dependence violation checking and (d) actual execution for the parallelized loops. Hence, we perform the following experiments with the speculatively parallelized program: (A) Without tracing, PRSD generation, forking and dependence violation checking; (B) with tracing overheads but without PRSD generation, forking and dependence violation overheads; (C) with tracing and PRSD generation overheads but without forking and dependence violation checking; (D) with tracing, PRSD generation and forking overheads but without the dependence violation checking overhead; (E) with all the overheads but no mis-speculation (oracle dependence checking) and (F) with all the overheads including potential mis-speculation.

These experiments were run with the speculative program also being compiled for class 'S'. With the above five experiments, we calculated the costs of each individual overhead. Table 2 shows the wallclock measurements of each experiment.

Table 2. Speculation Overhead Experiments

Experiment	Wall-clock time (seconds)				
	BT	EP	FT	LU	SP
SpecNoOverhead	1.13	7.53	1.28	0.26	0.37
SpecInstr	5.55	13.73	5.62	3.02	1.54
SpecInstrPrsd	84.29	14.79	6.48	309.59	210.40
SpecInstrPrsdFork	98.87	14.78	6.51	310.58	229.89
SpecInstrPrsdForkOracleCheck	105.67	14.80	6.52	520.48	248.66
SpecInstrPrsdForkCheck	111.11	14.78	6.52	516.13	308.49

Figure 2 shows the relative costs of each overhead during the speculative run normalized against the execution time considering all overheads. The overheads of the speculative run indicate a substantial contribution by tracing and PRSD compression followed by dependence violation checks (for selected benchmarks) as they dominate the overheads for most benchmarks. Recall the discussion on promoting tracing and PRSD generation to hardware monitoring. By shifting the burden of this costly task, data (trace) generation suitable for dependence checking becomes feasible. It still remains the responsibility of our software framework to perform dependence violation

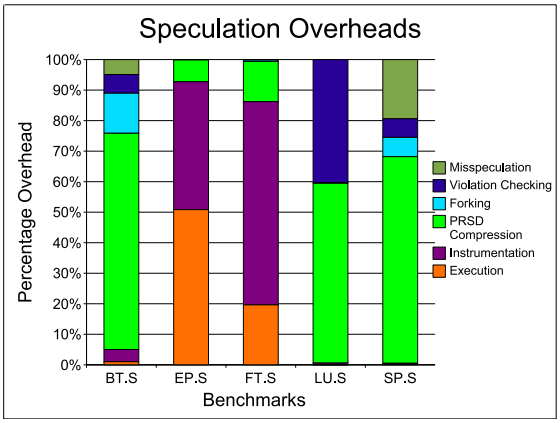

Fig. 2. Relative Costs of Speculation Overheads

checking. This may result in program slowdowns (compared to the version without dependence violation checks) for two reasons. The first is because of the the different levels of dependence checks, and the second is due to the invested time in executions resulting in incorrect speculation. Another source of overhead is due to retaining program state prior to entering speculation, which is due to forking. The current cost of forking depends much on the write set of pages of a given benchmark. Furthermore, there is still room for improvement since forking currently takes place at the entry to a speculative region. By coarsening the state capture to a lower nesting level (outer loops), this overhead can be further reduced, which is currently being investigated.

4 Related Work

Compressed access patterns have been proposed for *compile-time* analysis in past work, such as for array privatization and reduction detection [12, 27]. Unfortunately, complex control flow and input-dependent symbolic variables may make it impossible to reason about such compressed representations at compile time. In response, Rus *et al.* propose a *Hybrid Analysis* (HA) [25]. With HA, the compressed accesses (descriptors) are formed at compile time using sophisticated symbolic analysis. The descriptors may include symbolic terms whose values are unknown at compile time. The actual *use* of these descriptors for deciding if an optimization (*e.g.*, parallelization) is safe is deferred till runtime when the values of symbolic variables are known and can be plugged in. The need for sophisticated symbolic analysis (to form and aggregate the descriptors) limits the applicability of this approach to a subset of Fortran codes. In contrast, we do not need symbolic code analysis as the access descriptors are built on-the-fly based on observed run time addresses. We exploit Power Regular Section Descriptors (PRSDs) [17, 18], which provide the means to represent regular access patterns in constant space regardless of the number of loop nests and iterations using an effective online detection and compression technique. Hence, our scheme is more general and applicable to a larger set of programs than static schemes.

Past work on parallelization for TM has focused on bounded transactions in terms of data size with extensions to unbounded protocols [3, 6, 11, 20]. The former are common in hardware TM while the latter dominate in pure software solutions or hybrids.

XTM utilizes memory protection but lacks trace compression capabilities to speed up violation detection [7].

5 Conclusion

We have enhanced the ORC compiler to selectively instrument loops with potential dependencies. The access trace obtained by executing the instrumented program is utilized to calculate a relaxed dependence graph that our modified ORC compiler uses to generate a speculative program. Further, we have developed a pure software implementation for coarse-grained speculative parallelization that leverages past work on compressing regular access patterns using PRSDs.

While the cost of PRSD generation dominated the overheads incurred by our framework, regular access patterns can be recognized and compressed through simple and low-cost extensions in hardware, which would eliminate them as overhead. By coarsening the state capture to a lower nesting level (outer loops), the overhead due to dependence violation checking can be further reduced. We also believe that by selecting a coarser level of parallelization (with more work per thread) at an outer-more loop level, we would not only be able to mask this overhead but to enable speedups with our framework. We are currently investigating both these techniques.

Overall, our method can be more aggressive in parallelization than existing TLS techniques, yet is shown to be competitive in overhead to explicit parallelization or auto-parallelization. It can be applied even when data dependency checks remain inconclusive at compilation time or allow only TLS at the inner-most loop nest.

References

1. Open research compiler for itanium processor family. http://ipf-orc.sourceforge.net/.
2. Open64 compiler. http://sourceforge.net/projects/open64.
3. C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *International Symposium on High Performance Computer Architecture*, pages 316–327, 2005.
4. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
5. W. J. Blume. Symbolic analysis techniques for effective automatic parallelization. Technical Report UIUCDCS-R-95-1913, 1995.
6. W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *Architectural Support for Programming Languages and Operating Systems*, pages 347–358, 2006.
7. J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *Architectural Support for Programming Languages and Operating Systems*, 2006.
8. M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.
9. L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6):92–103, 2004.

10. L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.
11. T. Harris, A. Cristal, O. S. Unsal, E. Ayguad, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.
12. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
13. A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using spec cpu 2006. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–225, 2007.
14. A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism: The dl version of this paper includes corrections that were not made available in the printed proceedings. In *International Conference on Supercomputing*, page 24, 2006.
15. V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.
16. J. Marathe. Metric: Tracking memory bottlenecks via binary rewriting. Master's thesis, North Carolina State University, July 2003.
17. J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, and A. Yoo. METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *International Symposium on Code Generation and Optimization*, pages 289–300, Mar. 2003.
18. J. Marathe, F. Mueller, T. Mohan, S. A. McKee, B. R. de Supinski, and A. Yoo. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Transactions on Programming Languages and Systems*, 29(2), Apr. 2007.
19. M. Medidi and N. Deo. Parallel dictionaries using avl trees. *J. Parallel Distrib. Comput.*, 49(1):146–155, 1998.
20. K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: log-based transactional memory. *hpca*, 0:254–265, 2006.
21. R. Ramaseshan. Trace-based dependence analysis for speculative loop optimizations. Master's thesis, North Carolina State University, June 2007.
22. L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.
23. L. Ricci. Automatic Loop Parallelization: An Abstract Interpretation Approach. In *PARELEC '02: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, page 112, Washington, DC, USA, 2002. IEEE Computer Society.
24. P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3, Oct. 2001.
25. S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 274–284, New York, NY, USA, 2002. ACM Press.
26. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–24, 2000.
27. P. Tu and D. A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.
28. C. Wu, R. Lian, J. Zhang, R. Ju, S. Chan, L. Liu, X. Feng, and Z. Zhang. An overview of the open research compiler. In *Languages and Compilers for High Performance Computing*, pages 17–31, 2004.