

CpE 514
Computer Architecture

◆ A technical paper on:

**Thread-Level Data Speculation – A Key Technique for
Thread-Level Parallelism on Single-Chip Multiprocessors and
Beyond.**

◆ Guided by:

Prof. J. Stepleton

Computer Science Department

Stevens Institute of Technology

Hoboken, NJ

◆ Submitted by:

Amit D. Lakhani

SID: 999-02-8706

Dt.: 12 / 04 / 2000

Stevens Inst. of Technology

Fall 2000

Thread-Level Data Speculation - A Key Technique for Thread-Level Parallelism On Single-Chip Multiprocessors And Beyond

Abstract:

Due to the increasing transistor density on a single-chip processor computer architects are in a constant effort to utilize maximally the resources available on a processor. To fully utilize the efficiency of these single-chip processors, compiler execution of parallel threads looks too shortly feasible. But it has minimal success in executing non-numeric application due to their “weird” access patterns. This paper explains a newer technique - Thread-Level Data Speculation (TLDS) to rectify this. It is also demonstrated that through modest hardware augmentations single-chip hardware can support TLDS by directing its cache coherence scheme to detect dependence violations and use the primary data caches to buffer speculative state.

Introduction:

It has been a constant effort by computer architects to explore and utilize all the resources of the processors as “transistor-density” on a single chip goes on increasing and also to put these usage into improved performance. A number of options have been proposed, including integrating main memory onto the processor chip supporting wider instruction issue, and executing multiple threads of control in parallel. While these options may be mutually exclusive in the short term due to transistor constraints, in the long term we will eventually have enough resources to potentially combine several of these options. While there have been several proposals, perhaps one of the most compelling options is to integrate *multiple processors* on a single chip. Also, this is a good practice looking from the VLSI perspective because their distributed nature allows the bulk of interconnections to be localized rather than avoiding the delays in long wires.

Thus, due to diminishing returns of wider instruction issue and increased memory on chip, it seems that it will be only matter of time when processors start developing the performance benefits of parallel threads with compiler support. Simply

stated, a **Thread** is simply a small part of a larger program, which may or may not work independently. Thus, a large application can be divided into simple dependent or independent threads. The coarsely grained **Thread-Level Parallelism (TLP)** exploited by these “single-chip multiprocessors” is largely orthogonal to the more fine-grained **Instruction-Level Parallelism**, which wide-issue machines attempt to exploit within a single thread. Hence, with enough transistors, it may be possible to aggressively exploit both forms of parallelism in a complementary fashion.

At this level, an obvious question does come to mind what are the performance benefits of using single-chip multiprocessing. For a multiprogrammed workload, computational throughput will clearly improve as the independent tasks execute in parallel on multiple processors rather than time-sharing a single processor. However, to reduce the execution time of a single application, that application must contain parallel threads. Unfortunately, despite the fact that conventional multiprocessors have been commercially available for quite some time, only a small fraction of the world’s software has been written to exploit parallelism. Hence if single-chip multiprocessing is going to succeed and become ubiquitous, the key question is how do we convert the applications we care about into parallel programs? A key problem in this approach is also how to determine whether data dependencies occur between parallel threads? Well, to address this problem in numeric codes, a considerable research has focused on analyzing array accesses, a citable example is *The Doall Test*, but these even gets complex when accessing non-numeric codes due to their complex access patterns including pointers to heap-allocated objects and complex flow-control. Well, instead of this we can relax the constraints on the compiler by allowing it to view parallelization solely as a cost/benefit trade-off i.e. the compiler believes that the threads are *likely* independent, and it can go ahead and parallelize them without worrying about violating program correctness. This technique of predicting the outcome speculatively is better known as **Speculative Execution**. In this paper, it is a constant attempt to explore a technique that tries to implement this.

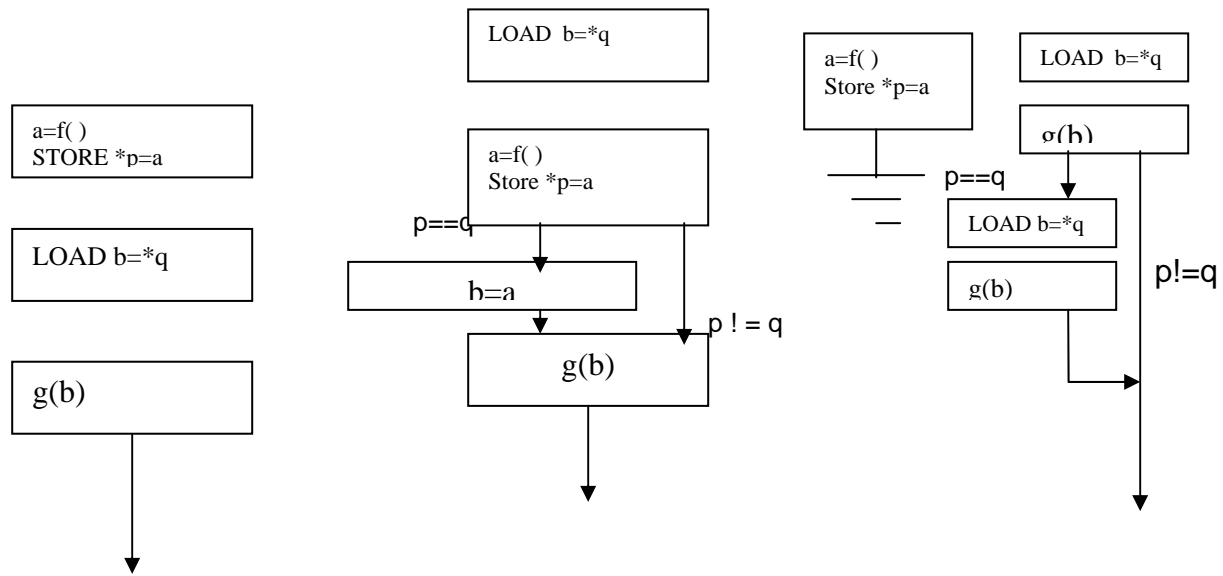


Figure 1.

(a) Original Execution

(b) Instruction Level Parallelism

(c) Thread Level Parallelism

Thread-Level Data Speculation

Thread-Level Data Speculation is similar to instruction-level parallelism, except that the load and store are executed by separate threads of control which run in parallel, as can be seen in the Figure 1. To maximize parallelism, we want to perform loads as early as possible so that operations, which depend on them, can be executed concurrently. Hence it is often desirable to move a load ahead of an earlier store, which is safe provided that they access different memory locations. Since analyzing memory addresses in non-numeric applications is difficult, a potentially attractive option is for the compiler to *speculatively* move a load ahead of a store, and resolve whether this was safe at run-time. If the speculative load turns out to have been unsafe, then a recovery action is taken to restore the correct program state. This technique is known as **Data Speculation**, and it works well when the unsafe cases are sufficiently rare that the overhead of recovery is small relative to the benefit of increased parallelism. Figure 1 is an explanation of the above technique, using instruction-level parallelism as well as thread-level parallelism. In this example, the compiler is uncertain whether the pointers

p and q point to the same address, but nevertheless it has speculatively moved the load ahead of the store. At run-time, we can verify the safety of this speculation through either a simple software check or with special hardware support, such as “memory conflict buffer”.

A given speculative load is safe provided that its memory location is not subsequently modified by another thread such that the store should have preceded the load in the original sequential program. When such dependence violations are detected, a recovery action is taken such as partially re-executing the thread, which performed the failed speculative load.

Related Work:

Although a lot of work has been done to improve and implement instruction level data speculation only two significant works have been noted in Thread-level Data Speculation- ***the Privatizing Doall Test(PD test) and Wisconsin Multiscalar Architecture***. *The PD test* is completely software based allowing the compiler to parallelize the loops without fully disambiguating all memory references. It has a concept of *shadow arrays* of each shared variable to track read and write accesses. At the end of each parallel execution of loop the shadow arrays are examined. If any cross-iteration data dependencies are violated, the loop is re-executed sequentially. Otherwise, we know that the execution was correct. Well, this directly brings the point that it cannot be used in non-numeric codes due to their various access patterns like pointers to heaps etc. also, it doesn't achieve any parallelism in presence of single cross-iteration RAW dependence.

In *Wisconsin Multiscalar architecture* a tightly-coupled ring architecture assigns threads around the ring in program order, provides a hardware mechanism for forwarding register values between processors, and uses a centralized structure called the “address resolution buffer” (ARB) to detect data dependences through memory. When an unsafe speculation is detected, a purely hardware-based mechanism squashes computation in reverse order around the ring until it can be safely restarted. But here too, the ARB is a relatively large, centralized structure, which must be checked on all loads and stores. A centralized approach has the danger of increasing load latency due to long wire delays. Instead, we would prefer a more distributed approach where loads and stores can be satisfied directly from their own primary caches.

Objectives:

Actually the approach of this paper is three-fold - to quantify the potential performance advantages of TLDS, to propose cost effective hardware support for TLDS, and to gain insight into the compiler support necessary to effectively exploit TLDS. Thus, we would exploit whether adding support for TLDS improves performance through increased thread-level parallelism? Secondly, can we provide the necessary hardware support for TLDS in a cost-effective way? And finally, what compiler support is needed to effectively support TLDS?

2. An Example: Compress

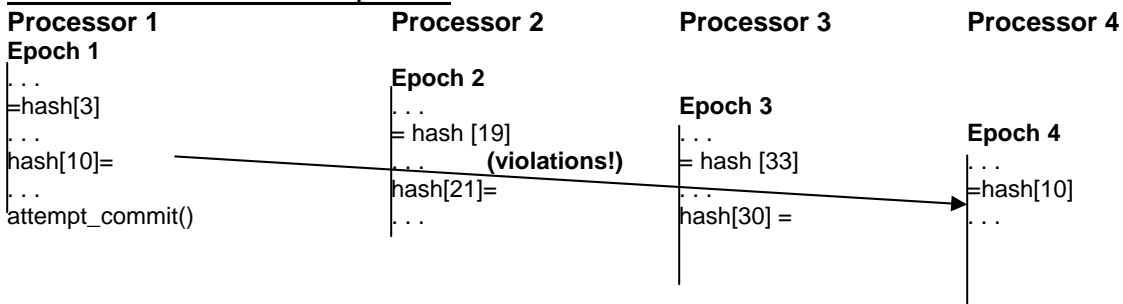
Before we illustrate how TLDS applies to a real application, we briefly introduce some terminology. We say that TLDS parallelism is extracted from a *speculative region*, which consists of a collection of dynamic instruction sequences called **epochs**. For example, with loop-level parallelism, we would say that the loop is a speculative region, and the individual loop iterations would be epochs. Since TLDS parallelism also applies to structures other than loops (e.g., recursion), we have adopted this more general terminology. For TLDS to be effective, each epoch must contain enough work to amortize the costs of thread management and data communication, and the speculative regions must constitute a significant fraction of overall execution time.

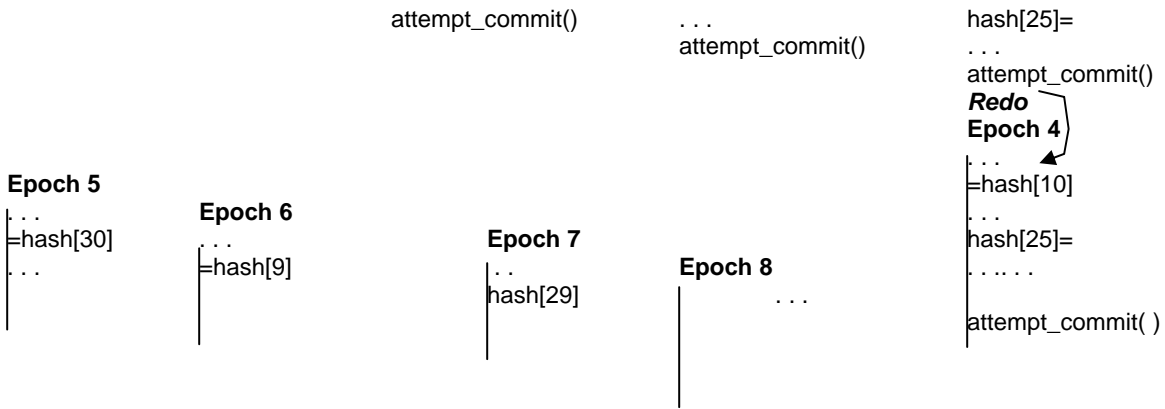
Figure 2

Pseudo code representation of compress

```
while ((c = getchar()) != EOF) {
  /* perform data compression */
  in_count++;
  ...
  ... = hash[hash function(c)];
  ...
  hash[hash function(...)] = ...;
  ...
  if (...) {out_count++; putchar(); ...}
  if (free_entries < ...) {free_entries = ... }
  ...}
}
```

TLDS execution of compress





The `compress` application in the SPEC92 and SPEC95 benchmark suites is a good candidate for exploiting TLDS. Over 99% of execution time is spent in a single while loop, which reads each input character and performs the compression. A compiler cannot statically prove that loop iterations are independent because they are not. The input characters are used to index a hash table, which is modified. Hence when two character sequences hash to the same entry, there is a true *read-after-write* (RAW) data dependence. Figure 2 shows a pseudo-code representation of this code. Fortunately, due to the nature of a hash table, consecutive characters rarely access the same hash table entry—therefore there is an opportunity to extract parallelism during the iterations *between* actual dependences. Since a single-chip multiprocessor has a relatively small number of processors, we do not need a large gap between data-dependent iterations to keep the machine busy. Figure 2(b) illustrates how `compress` can be parallelized using TLDS, where each epoch (i.e. loop iteration) is executed as a separate thread. Since the threads are speculative, any stores, which they perform must be buffered until it is certain that their results are needed. The fourth epoch, however, reads the value `hash[10]` *before* it is modified by epoch 1—since this violates the original program semantics, we must recover by re-executing epoch 4. In addition to the hash table accesses, `compress` contains several other sources of RAW data dependencies. Thus we see that a range of different compiler optimizations can potentially enhance TLDS, and we will quantify their benefit in the next section.

From the above example, two important effects come out which might limit performance - *data dependences through memory caused by ambiguous load and store addresses, and synchronization and communication latency when we explicitly forward scalar values between epochs to partially overlap execution.* We examine each of these in the following sections:

Relaxing Memory Data Dependences:

Data dependences between epochs can occur either through registers or through memory. Register dependences are generally easier for the compiler to analyze since the storage locations are not ambiguous. As illustrated earlier in Figure 2, TLDS can exploit parallelism whenever gaps exist between data-dependent epochs. We quantify this potential by computing the *run lengths*, which are the number of consecutive epochs within a speculative region delimited by *performance-limiting* read-after-write (RAW) data dependences. While RAW dependences can potentially disrupt parallelism (forcing a processor to re-execute an epoch), this is not always the case because when one of the epoch say E_i is executing another epoch say E_j , where $j \leq (i - T)$, where T is the number of speculative threads, will already have *committed* its state to memory. Hence a RAW dependence of distance $d > T$ will not limit performance.

Forwarding data between Epochs:

Although these scalar dependences cannot be fully eliminated (unlike the B and O cases), we can potentially accelerate these cases by explicitly *forwarding* the values between epochs. In addition to forwarding scalar memory dependences, we must also forward any register. In addition to a data transfer mechanism, forwarding data also requires a form of producer-consumer synchronization. The performance of a speculative region that requires forwarding is limited by the *critical path length*, which is the sum of the non-overlapped portions of each epoch plus the latency of forwarding these values between epochs. The performance with forwarding depends on how aggressively we attempt to minimize the non-overlapped portions of each epoch. In addition to using fine-grain rather than coarse-grain synchronization, we can potentially improve performance further by rescheduling the code to move as many instructions out of the non-overlapped portion of an epoch as possible.

Architectural Support for TLDS:

After studying the theoretical aspect we need to concentrate on the implementation of TLDS. Actually, our goals are twofold. *First*, we would like to support an aggressive form of TLDS while requiring only minimal hardware modifications to a generic single-chip multiprocessor. *Second*, we do not want to sacrifice performance in

single-threaded applications or applications that do not exploit TLDS—hence we will avoid complex, centralized structures which can increase primary data cache access latencies. Therefore the starting point for our design is a single-chip multiprocessor where the Level 2 cache is physically shared and the individual Level 1 caches are kept coherent to provide a shared memory abstraction. Experiment results have shown that although fast communication is necessary but using the level 2 shared cache is a viable approach.

Thread Management:

A number of mechanisms are required by TLDS to manage and coordinate the parallel threads. The important issues in this matter are understated. *First*, we need a way to create parallel threads and schedule the epochs onto them. *Second*, since dependence violations are detected by comparing epoch numbers, a mechanism is needed such that each thread's epoch number will be visible to the hardware. One way to accomplish this is for software to explicitly pass epoch numbers to the hardware through a new instruction. *Third*, we need to distinguish speculative versus non-speculative memory accesses, since only speculative operations must be buffered or checked for dependence violations. Rather than creating new flavors of all memory references in the instruction set, we can instead use explicit instructions to dynamically indicate whether a thread is speculative or not—when a thread is speculative, all of its memory references will be interpreted as being speculative. Finally, we need a mechanism for recovering from failed speculation. We propose that software performs the bulk of the recovery process, and that hardware simply provides two key pieces of functionality-(i) detecting data dependence violations and notifying software when they occur, and (ii) buffering speculative stores so that software does not have to explicitly roll back their side effects on memory. These are some of the basic steps required to manage threads and thus implement TLDS. Various other schemes like the cache coherence can also be included in the coarse-granulated approach, but it would also involve a programming overhead and although the instructions to be added for such a scheme may be small, it would cause considerable burden on the programmer.

Compiler support of TLDS

The compiler clearly plays a crucial role in exploiting TLDS. In addition to selecting regions of the code to speculatively parallelize and inserting the appropriate TLDS primitives, compiler has an important role in *optimizing* the code by removing data dependences and maximizing parallel overlap if we are to achieve the full potential of TLDS.

The first step in compiling for TLDS is choosing the appropriate speculative regions to parallelize. Our goals here are twofold: (i) maximizing the fraction of total execution, which is parallelized, and (ii) achieving the best speedups within each speculatively parallelized region. To maximize program coverage in a cost-effective manner, the compiler could also make use of control-flow profiling information to focus its efforts on the sections of code, which account for the largest fractions of total execution time.

First, we would like to choose regions where the epochs are large enough to amortize the costs of thread management and communication. The compiler has considerable flexibility in adjusting the epoch sizes within a region. If the epochs are too large, the compiler can statically split them up into smaller pieces (e.g., divide a loop body in half). If the epochs are too small, the compiler can merge consecutive epochs to form larger epochs (e.g., unroll a loop and fuse consecutive iterations).

Second, we would like to maximize the probability of successful speculation by avoiding regions with problematic data dependences. After selecting candidate regions to speculatively parallelize, the compiler should then *optimize* their performance in the following ways. The first step is eliminating as many dependences as possible—e.g., induction variables, reduction operations, and dependences inside library routines. Next, it should insert code to explicitly forward these values between epochs, and (most importantly) *reschedule* the epochs to minimize the critical path.

Finally, the compiler must insert calls to the TLDS primitives and create the recovery code. Note that although the hardware restores the memory state during recovery by discarding any speculative stores, it is the responsibility of software to restore any necessary register state. The compiler can reduce the recovery overhead by reexecuting only the portion of the epoch, which depends on speculative load results,

and by initiating the recovery process early either through an interrupt mechanism or by polling the violation flag early.

In summary, although automatically parallelizing non-numeric codes is still a non-trivial task, it is at least *feasible* with TLDS, in contrast with the hopelessly restrictive model of statically proving that threads are independent.

Conclusions.

As was the approach to obtain a breakthrough in the compiler technology by automatically parallelizing code fragments, we have investigated a technique called TLDS, but the compiler however cannot prove that the dependences do not exist. Still experimentally it is proved that TLDS can have a speedups ranging from 15% to nearly fourfold in seven of ten cases. Also, if larger speedups are required a combination of finely grained Instruction level parallelism alongwith TLDS can be implemented for extensive speedups.

For the implementation of the TLDS scheme only modest hardware is required. For example, the cache coherence scheme can be extended to detect true RAW dependences, also the cache can be itself used to buffer speculative memory accesses. Also due to the distributed nature of this hardware, degradation from present performance is not anticipated. In future compilers can be thought of having full support for automatic parallelization for non-numeric applications using TLDS, the goal that is setup in this paper.

References:

- 1) J. G. Steffan, C. B. Colohan, and T. C. Mowry. *Architectural Support for Thread-Level Data Speculation*. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.
- 2) Fourth Annual Symposium on HPCA (High-performance computer Architecture) – 4, 1999. website: <http://www.computer.org/tab/tcca/HPCA-4>
- 3) *Readings in Computer Architecture*- Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, Morgan Kaufmann publishers.
- 4) Muchnick, Steve S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers Inc., pages 134-760.

- 5) M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley Longman.
- 6) *Computer Architecture- a quantitative approach*, David Patterson and John L. Hennessy, *Chapter – 4 and pages 650-750 on cache coherence*
- 7) M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University.