Solving Large Problems Quickly: Progress in 1999

Todd C. Mowry Angela Demke Brown
Christopher B. Colohan Spiros Papadimitriou
J. Gregory Steffan Antonia Zhai

February 2000
CMU-CS-00-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This document describes the progress we have made and the lessons we have learned in 1999 under the NASA Grant NAG2-1230 entitled "Application-Specific Supercomputing". The long-term goal of this research is to accelerate large, irregular scientific applications which have enormous data sets and which are difficult to parallelize. To accomplish this goal, we are exploring two complementary techniques: (i) using compiler-inserted *prefetching* to automatically hide the I/O latency of accessing these large data sets from disk; and (ii) using *thread-level data speculation* to enable the optimistic parallelization of applications despite uncertainty as to whether data dependences exist between the resulting threads which would normally make them unsafe to execute in parallel. Overall, we made significant progress in 1999, and the project is going well.

This research is supported by NASA Grant NAG2-1230.

Keywords: B.3.2 Cache Memories, C.1.2 Multiple Data Stream Architectures (Parallel Processors), C.4 Performance of Systems, D.3.4 Compilers

1 Introduction

A number of important scientific and engineering applications suffer from the following two limitations. First, the largest problem size that one can solve in practice is constrained by the amount of physical memory in the system. Although one could (in theory) attempt to execute an application with a data set that is too large to fit in main memory (commonly refereed to as an "out-of-core" application) by simply relying on a paged virtual memory system to transparently migrate the data between main memory and disk, the resulting performance is typically so bad that programmers are faced with the formidable task of restructuring their code to use explicit I/O calls for the sake of achieving reasonable performance. Second, the task of decomposing applications which use sophisticated data structures (involving pointers and other forms of indirection) into parallel threads is quite onerous since their unpredictable memory access patterns make it difficult (and in many cases impossible) to statically prove that potential threads are in fact independent. This latter problem is particularly troublesome for large "legacy" codes where many years of effort have already been invested in simply getting a sequential version of the program to work correctly.

The goal of this research project is to overcome both of these limitations using the following two techniques. First, to achieve high performance on out-of-core data sets, we explore a fully-automatic scheme for prefetching I/O whereby the operating system and the compiler cooperate to combine the advantages of both explicit I/O (i.e. high performance) and paged virtual memory (i.e. portability and no burden on the programmer) without suffering from the disadvantages of either approach. Second, to overcome the problem of ambiguous data dependences between potential parallel threads, we explore a new mechanism called **T**hread-**L**evel **D**ata **S**peculation (**TLDS**) which enables either the programmer or the compiler to optimistically speculate that no dependences exist between the threads and run them in parallel.

1.1 Progress Toward Milestones

In our proposal, we set forth the following six technical milestones to be completed in 1999. (The seventh milestone was to complete and deliver this technical report, so we do not list it below.)

1. Complete the performance debugging of our implementation of I/O prefetching support within IRIX, and run out-of-core versions of the NAS parallel benchmarks with compiler-inserted prefetching on SGI Origin machines. Add new optimizations to the compiler to cope with the larger I/O latencies in these modern machines.

2. Complete the implementation of compiler, run-time layer, and operating system support for intelligently managing physical memory through "release" hints to maximize the performance of multiple out-of-core applications running simultaneously on the same machine. Evaluate the performance of this system using NAS parallel applications running on SGI Origin machines. Tune the compiler algorithm and run-time software layer appropriately to enhance performance.

3. Complete an initial implementation of thread-level data speculation support within a software DSM system. Evaluate this prototype on an SGI Origin machine. Develop performance monitoring support within this system to help us understand and optimize the performance of applications running on this system.

4. Complete the speculative parallelization (by hand) of the mesh generation application, and evaluate its performance using both simulations and the software DSM implementation of thread-level data speculation.

5. Begin the implementation of a compiler framework for automatically parallelizing irregular scientific applications so that they can exploit thread-level data speculation.

6. Establish ties with a technical contact within NASA to help us learn about the performance characteristics of applications that are of interest to NASA. This interaction will help facilitate technology transfer so that we can eventually apply our compiler techniques to real applications within NASA.

We are pleased to report that we have met all of our milestones (except for the fourth milestone, which was dropped based on feedback from our NASA contacts).

Section 2 describes our implementation of I/O prefetching support within IRIX, and presents the results of our experiments on the entire set of NAS parallel benchmarks (shown in Figure 5). We are pleased to observe that all of the applications are enjoying significant performance gains through prefetching, and this confirms our earlier experiments on the research prototype system at Toronto [5]. We did some basic tuning of the parameters that the compiler uses to model the memory hierarchy to get these results, although we think that we can do even better than this with more extensive modifications to the compiler algorithm. Section 2.7.2 describes the work that we have begun in this direction.

Section 2 also describes our implementation of *release* (as well as prefetch) hints within the compiler, run-time layer, and operating system, which was the goal of our second milestone. Figure 5 breaks down the contribution of releasing memory as opposed to simply performing prefetching without releasing for the NAS parallel benchmarks. We believe that further improvements are possible by making the run-time layer more intelligent, and we describe these efforts in Section 2.7.1.

Section 3.3 describes our initial implementation of thread-level data speculation within a software DSM system. We believe that our implementation is now complete and functional. We spent the past several months identifying and eliminating some fairly subtle bugs in the system, and this took more time than we had originally anticipated. We have verified that it performs well on simple test cases on an SGI Origin, and we have also added performance monitoring code to the system to understand its performance. We are now in the process of porting several real applications to this system to evaluate their performance. The lessons that we learn from these real applications will allow us to further tune our system.

The fourth milestone was abandoned once we got feedback from our NASA contacts that mesh generation was not an application of interest.

We are much further ahead on our fifth milestone (i.e. beginning an implementation of a compiler framework for automatically parallelizing codes using TLDS) than we had expected to be at this time. We have built several key pieces of the compiler, which are described in Section 3.2.

We have also completed our sixth milestone, which was to establish technical contacts within NASA for the sake of learning about applications that NASA cares about and facilitating technology transfer. We have acquired two large applications from NASA, and we are starting to evaluate the impact of our techniques on their performance. We expect to accelerate the pace of this interaction and technology transfer dramatically within this next year.

Overall, this has been a very productive year, and the overall project is slightly ahead of schedule at this point. In the following sections, we discuss our research progress in more detail.

2 Compiler-Inserted I/O Prefetching and Memory Management

Applications whose data requirements surpass the amount of physical memory available (which we refer to as "out-of-core" applications), spend a large fraction of their execution time waiting for I/O operations to complete. It has been shown that prefetching of I/O requests can be an effective way to overlap the I/O latency with other computation, resulting in dramatic improvements in the overall execution time. Successful prefetching, however, depends critically on determining what data should be prefetched, and at what time the prefetch should be initiated. Since making these decisions manually represents a substantial burden on the programmer, we have instead explored automated techniques based in static compiler analysis for obtaining the required information. Good performance for out-of-core applications also depends on making the right decision about what data should be replaced from memory to make room for the prefetched data. Again, application-specific knowledge can be exploited using similar compiler analysis to provide replacement hints to the operating system, thus improving global system performance.

Our approach combines static compiler analysis with operating system support for simple *prefetch* and *release* primitives (which provide hints about the pages that need to be brought into memory, and that can be replaced, respectively) and an efficient mechanism to share information between the operating system and the application. The out-of-core applications are written assuming unlimited virtual memory (i.e., without any explicit I/O calls), relying on the compiler to perform the necessary analysis and transformations to hide the latency of page faults. The run-time layer uses the information provided by the operating system to *filter* the requests inserted by the compiler, improving performance. An initial evaluation of this approach using a research operating system and multiprocessor at the University of Toronto demonstrated the enormous potential for improving the execution time of out-of-core applications [5]. To evaluate the approach on a

Table 1: Summary of results of compiler analysis phase

Locality Type	Prefetch Required	Release Possible
None	Before each reference	After each reference
Temporal	Only during first iteration	Only after last iteration
Spatial	Every p iterations (p is number of items per page)	After every p iterations
Group (leading ref.)	determined by locality of group	Never
Group (trailing ref.)	Never	determined by locality of group
Group (all others)	Never	Never

state-of-the-art system, we first needed to implement the required support in a modern operating system. We are using SGI's IRIX 6.5 for this purpose. Given this basic support, we can then begin to explore ways in which application-specific knowledge for resource-intensive out-of-core applications can be used to improve global system performance in a multiprogrammed environment.

In this section we will describe our automated system for I/O prefetching and memory management, including the implementations of the compiler, the run-time layer, and the operating system, as well as the interfaces between each of these major components. We will also present some initial performance results.

2.1 Compiler Implementation

The compiler algorithm consists of two major phases: (1) an analysis phase in which the references to be prefetched and released are identified, and (2) a scheduling phase in which prefetch and release instructions for these references are inserted into the code. We will briefly describe the analysis phase, then look more closely at the scheduling phase.

2.1.1 Analysis Phase

A data item has *reuse* if it is referenced multiple times. Reuse is thus an intrinsic property of a given data access pattern. In contrast, *locality* only results when subsequent references find the data item still in memory. Hence, it is a function of the size of memory, the volume of data accessed between reuses, and the page replacement policy used by the operating system. Since the operating system's replacement policy is beyond the compiler's ability to analyze, we simply assume that a page is likely to be replaced if the amount of data accessed between reuses is greater than the size of physical memory. This assumption would be true for a strict least recently used (LRU) replacement policy. If memory were unlimited, then reuse and locality would be identical; in reality, the references with data locality are a subset of those with data reuse.

To properly identify what needs to be prefetched, we must distinguish three types of data reuse (and three corresponding types of locality), each of which needs to be handled in a different manner. *Temporal reuse* occurs when a particular reference accesses exactly the same data location in different iterations. *Spatial reuse* occurs when a particular reference accesses different data locations found on the same page. *Group reuse* occurs when different references access data locations found on the same page.

Given the relationship between reuse and locality, the locality analysis algorithm is comprised of two main steps:

1. Discover the intrinsic data reuses within a loop nest through *reuse analysis*. This would be equivalent to solving the locality analysis problem if memory were unlimited.

2. Given that we have a *finite* memory, determine the set of reuses that actually result in locality. This is accomplished by computing the *localized iteration space*, which is the set of nested loops that access less data than the specified memory capacity. Data locality is then computed by intersecting the intrinsic data reuses with the localized iteration space. i.e.

$$\text{Data Reuse} \cap \text{Localized Iteration Space} \Rightarrow \text{Data Locality}$$

These steps produce a mathematical description of locality in a *vector space* representation. The analysis phase of the compiler is essentially the same as that used for cache prefetching, with one exception. For

(a) Code Before Unrolling

```
for (i = 0; i < n; i++) {
  if ((i mod 64) == 0)
    f(i);
  g(i);
  if ((i mod 64) == 0)
    h(i);
}
```

(b) Code After Unrolling

```
for (i = 0; i < n; i+=64) {
  f(i);
  g(i);
  g(i+1);
  g(i+2);
  ...
  g(i+63);
  h(i);
}
```

(c) Code After Strip-Mining

```
for (j = 0; j < n; j+=64) {
  f(j);
  for (i = j; i < j+64; i++)
    g(i);
  h(j);
}
```

Figure 1: Example of unrolling and strip-mining a loop by a factor of 64

references with group reuse, the *trailing reference* (i.e. the last reference to touch a particular data item) is identified in addition to the *leading reference* to determine when releases are possible. More details on the compiler analysis can be found in Mowry's thesis on cache prefetching [4] and Demke's thesis on I/O prefetching [1]. A summary of the results of the analysis phase for determining which references to prefetch and release is given in Table 1. Note that for references with no locality, it is possible to release after each reference, however, in our current implementation we do not actually generate any releases for these references. Since we have no locality information for these types of references, it is also possible that the same data will be reused again in the near future, and we prefer to try to retain the data in memory for this case.

In practice, a number of considerations complicate the computation of data locality. First, symbolic loop bounds make it difficult to determine the exact amount of data accessed in a loop nest. Aggressive constant propagation can resolve some of these cases, but in other instances the compiler must simply assume unknown loop bounds to be either small (always localized) or large (never localized). Second, the actual amount of memory available at run-time may be quite different from that specified at compile-time, due to the resource demands of other applications. For prefetches, this problem can be partially handled by under-estimating the amount of available memory at compile-time and relying on the run-time layer to reduce the overhead of unnecessary prefetches generated by this approach. For releases, however, underestimating the amount of available memory causes the compiler to insert releases for references that are known to have reuse, but are believed to have no locality. That is, the compiler expects the data to be replaced before the reuse occurs. To handle these cases, we encapsulate the inherent reuse information discovered by the compiler by including a *priority* parameter in release calls. The run-time layer can then use these priorities to determine whether a released address should be returned to the operating system immediately, or buffered until physical memory actually becomes scarce. They also provide a means for the run-time layer to decide which data addresses should be released first when there are multiple references from which to choose.

2.1.2 Scheduling Phase

There are two main steps in the scheduling phase. First, the results of the locality analysis are used to perform *loop splitting* to isolate references that are expected to incur page faults. This technique allows prefetches to be inserted only when the analysis suggests they are necessary, reducing the overhead of always prefetching. Each type of locality requires a different loop splitting technique.

For references that have spatial locality with respect to a given loop, page faults will only occur when page boundaries are crossed. Similarly, a page can only be released after the last item on that page has been used. If the number of data elements in a page is p, this will occur whenever "`loop_index` mod $p = 0$". For cache prefetching, p was typically a small value, and the transformation was accomplished by *unrolling*

(a) Code Before Peeling

```
for (i = 0; i < n; i++) {
  if (i == 0)
    f(i);
  g(i);
  if (i == n-1)
    h(i);
}
```

(b) Code After Peeling

```
f(0);
g(0);
for (i = 1; i < n-1; i++) {
  g(i);
}
g(n-1);
h(n-1);
```

Figure 2: Example of peeling the first and last iterations of a loop

(a) Original Loop

```
for (i = 0; i < 1000000; i++)
  A[i] = 0;
```

(b) Software Pipelined Loop

```
prefetch_block_exc_aligned(&A[0], 188);                /* Prolog */

for (i1 = 0; i1 < 229376; i1 += 16384) {      /* Steady State */
  prefetch_block_exc(&A[i1 + 786431], 4);
  for (i = i1; i < i1 + 16384; i++) {        /* Strip-mined loop */
    A[i] = 0;
  }
}

for (i = 229376; i < 1000000; i++)            /* Epilog */
  A[i] = 0;
```

Figure 3: Example of how software pipelining is used to schedule prefetches the proper amount of time in advance. For this example, 786,431 iterations are required to hide I/O latency.

the loop p times. For I/O prefetching however, p is much larger and unrolling is no longer reasonable. The appropriate transformation is to *strip-mine* the loop by a factor of p, so that prefetches and releases can be inserted once every p iterations. One additional complication arises with I/O prefetching and multi-dimensional loops, however. Since the number of items on a page is typically large, it is possible that p is greater than the number of iterations in the innermost loops, making strip-mining impossible. A simple example of strip-mining and unrolling a loop is given in Figure 1.

References with temporal locality with respect to a given loop are only expected to page fault during the first iteration of that loop, and can only be released during the final iteration. The appropriate transformation is to *peel* the first and last iterations so that prefetches can be inserted during the first iteration and releases can be inserted during the last iteration. An example of peeling a simple loop is given in Figure 2

The overall transformation required for a particular reference is determined by combining the transformations needed for each type of locality that the reference has with respect to the surrounding loops.

The second step in the scheduling phase is to use *software-pipelining* to prefetch data early enough to hide the latency of the page fetch. This technique allows us to overlap the prefetches for a future iteration with the computation of the current iteration. Given that loop iterations are used as the unit of scheduling in the pipelining algorithm, two key issues need to be resolved to schedule prefetches effectively: finding the prefetch distance and choosing the pipelining loop.

Given the amount of latency that needs to be hidden, the problem of finding the prefetch distance (in terms of the number of iterations of the pipeline loop) is simply a matter of determining how much computation is needed. If the latency is expressed as a number of cycles, and we assume that each instruction takes a single cycle, then the number of iterations required, d is given by:

$$d = \left\lceil \frac{l_m}{s + l_p} \right\rceil \quad (1)$$

where d is the prefetch distance, l_m is the expected I/O latency, s is the number of instructions in the shortest path through the loop body and l_p is the software overhead introduced by adding a prefetch instruction to the loop body. The prefetch overhead latency parameter, l_p, is used to more closely approximate the time spent executing a loop iteration *after* a prefetch request is inserted in the loop. This consideration is especially important for short loop bodies since the time spent issuing the prefetch can be a significant fraction of the time to execute an iteration. In general, it is impossible to determine exactly how many instructions will be executed, however we choose the shortest path to ensure that prefetches are issued early enough. The ceiling of the ratio is used to ensure that all of the latency is hidden.

As an example, consider the code in Figure 3. For this example, we have chosen parameters that correspond to our experimental architecture (Irix 6.5 on an Origin 200): the page size is 16384 bytes, the I/O latency is 2,000,000 cycles, and the prefetch latency is 10,000 cycles. Since the A[i] reference has spatial locality, the `i` loop is first strip-mined into loops `i1` and `i` using a block size of four pages. Using equation 1, the compiler then determines that 47 iterations of the outer `i1` loop (with a prefetch call added) are needed to fully hide the I/O latency. To initialize the pipeline, a *prolog* is constructed to prefetch the first 188 pages in a single block prefetch call. Next, the *steady state* loop is executed. In this loop, blocks of four pages are prefetched in every iteration of the `i1` loop (to be used in the `i` loop 47 `i`-iterations later), and the current block of four pages are used in the inner `i` loop. Finally, the *epilog* loop performs the final iterations of computation, in which all the data has already been prefetched and only the original computations are performed.

The problem of choosing the pipelining loop is unique to I/O prefetching. For cache prefetching, prefetches were software pipelined around the innermost loop nest that changed the value of the array indexing function (i.e. the first loop that changed the address referenced). For example, in Figure 4(a) the `m` loop would be chosen as the pipeline loop. This approach presents a problem for I/O prefetching, since the loop that would ordinarily be chosen may not have enough iterations to hide the I/O latency, preventing the pipelining from being effective.

Figure 4(b) shows the result of choosing `m` as the pipelining loop. Since the `m` loop has spatial locality, we would like to request blocks of four pages with each prefetch. Since each iteration of the `m` loop accesses only 20 bytes of data, it would take 3277 `m` loop iterations to cover a four page block, which means that strip-mining is not performed since only five iterations are available. Using equation 1, the compiler determines that prefetches need to be issued roughly 278,000 `m` loop iterations ahead, causing a block prefetch for 68 pages to be inserted for the prolog section of the pipeline, as shown in Figure 4(b). Now, since the spatial locality of the `A[i][j][k][l][m]` reference suggests that we only need to prefetch every 3277 `m` loop iterations, and there are only five iterations in the code, the steady state and epilog sections of the pipeline are not created. The effect is that when new pages are prefetched at all, the request is issued just before entering the `m` loop as part of the prolog, but are never early enough to be useful. The fundamental problem is that the pipeline never gets into the steady state.

To cope with this problem, we modified the scheduling part of the algorithm to consider the amount of data actually used in what would ordinarily be chosen as the pipeline loop. If the pipeline loop has spatial locality, and the total data traffic across *all* iterations of that loop is less than a block (i.e. four pages in our experiments), then we choose the next surrounding loop nest as the pipeline loop instead. We apply this heuristic recursively until a loop that accesses more than a block of data, or the outermost loop is found. The result of this modification is shown in Figure 4(c), where prefetches are software pipelined across the `j` loop, rather than the `m` loop. It is now possible to schedule prefetches early enough to hide all the latency.

2.2 Operating System Implementation

We have implemented support for user-level paging directives (i.e. prefetch and release) within the SGI IRIX 6.5 operating system. IRIX 6.5 supports a Memory Management Control Interface, which consists of policy modules that allow users to select various policies for page size, allocation, migration, and replication. A policy module may be connected to any range of an application's virtual address space, down to the level of a single page. We have defined a new policy module—called "PagingDirected"—that allows a user-level process to

(a) **Sample Code Without Prefetching**

```
for (i = 0; i < 64; i++) {
  for (j = 0; j < 128; j++)
    for (k = 0; k < 128; k++)
      for (l = 0; l < 5; l++)
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;
}
```

(b) **Code After Software Pipelining (original)**

```
for (i = 0; i < 64; i++) {
  for (j = 0; j < 128; j++)
    for (k = 0; k < 128; k++)
      for (l = 0; l < 5; l++) {
        prefetch_block_exc_aligned(&A[i][j][k][l][0], 68);    /* Prolog */
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;
      }
}
```

(c) **Code After Software Pipelining (new)**

```
for (i = 0; i < 64; i++) {

  prefetch_block_exc_aligned(&A[i][0][0][0][0], 68);        /* Prolog */

  for (j0 = 0; j0 < 40; j0 += 10) {                  /* Steady State */
    prefetch_block_exc(&A[i][j+89][0][0][0], 4);
    for (j = j0; j < j0 + 10; j++)
      for (k = 0; k < 32; k++)
        for (l = 0; l < 5; l++)
          for (m = 0; m < 5; m++)
            A[i][j][k][l][m] = 0;
  }

  for (j = 40; j < 64; j++)                          /* Epilog */
    for (k = 0; k < 32; k++)
      for (l = 0; l < 5; l++)
        for (m = 0; m < 5; m++)
          A[i][j][k][l][m] = 0;

}
```

Figure 4: Example of Software Pipelining with Small Loop Bounds.

invoke prefetch and release operations on pages of its address space associated with this policy. In addition, the PagingDirected policy module shares information about memory usage with the application through a single 16KB page. This page is allocated by the operating system and mapped read-only into the application's address space when the PagingDirected policy module is created. The page is used primarily as a bitmap, indexed by virtual page number, in which bits are set to indicate that the corresponding page is in memory, and cleared otherwise. The operating system can also pass other information to the user-level via the shared page, such as the total amount of memory currently used by the process and the amount of additional memory still available.

When the PagingDirected policy module receives a request to prefetch a page, it performs actions similar to those that occur for a page fault, with two notable exceptions. First, if there is no free memory available to allocate for the prefetched data, the prefetch request is discarded immediately. Second, when the request completes, the prefetched page is not fully validated and no entry is made in the TLB. The second feature prevents mappings for prefetched (and not yet referenced) pages from displacing TLB entries which are still in use.

Requests to release pages are handled by passing the released addresses to a new system releasing daemon—called the releaser—which is similar in function to the paging daemon, but is specialized to reclaim only the pages specified by the application. When a release request is made, the PagingDirected policy module clears the bits for the pages and enters the request in the releaser's work queue. The releaser handles requests from each prefetching/releasing application as they are received, first checking the bit vector to make sure that the pages have not been referenced again (either by a prefetch or a real reference) between the time that the application made the request and the time that the request is handled. The releaser then performs all actions needed to free the pages, including the allocation of swap space and writing back dirty pages if necessary. Released pages are placed at the end of the free list, so that they will not be reallocated for another purpose immediately. This strategy gives pages that were released too early a chance to be rescued from the free list.

All updates to the shared page are handled by the operating system. When the PagingDirected policy module is created, all bits in the shared page are initially set. When the application attaches the policy module to a region of its virtual address space, the bits corresponding to those addresses are all cleared. Thereafter, bits are set whenever a physical page is allocated for a virtual page associated with this policy module, either due to prefetch requests or ordinary page faults. Bits are cleared when pages are reclaimed, either by an explicit release request or due to default page replacement activity. Note that since the base page size in Irix 6.5 is 16KB, we are able to represent 2GB of memory using a granularity of one page per bit, which is sufficient for a 32-bit address space (the upper half of the address space belongs to the operating system and can't contain user data). For 64-bit address spaces that are expected to be sparsely populated, we need to consider a multi-level bit vector scheme in which shared pages are allocated for the portions of the address space actually in use.

2.3 Run-time Layer Implementation

The role of the run-time layer is to improve performance by adapting the prefetch and release requests inserted by the compiler based on actual run-time conditions. It needs to consider both the results of the static analysis performed by the compiler (available via the prefetch and release calls inserted) and the dynamic information provided by the operating system via the shared page. The run-time layer is also responsible for making the actual calls to the operating system when necessary.

To achieve the full benefit of prefetching, we need to be able to both fetch data asynchronously (so the application can continue after issuing the prefetch) and take advantage of any available parallelism in the underlying disk subsystem. The run-time layer accomplishes these requirements by creating a number of *pthreads* [2] that make the actual calls to the PagingDirected policy module and wait for the prefetches to complete. When a prefetch request inserted by the compiler is intercepted by the run-time layer, the bitvector is first checked to see if a prefetch is really needed. Then, if necessary, the request is placed on a work queue and one of the prefetching threads is signaled to handle the request. The prefetching threads simply remove requests from the queue and issue them to the PagingDirected policy. This pthreads-based approach to achieving asynchronous prefetching is very similar to the implementation of the asynchronous I/O library in IRIX.

For releases, the current implementation of the run-time layer uses a very simple buffering scheme, together with the bit vector to improve performance. Rather than trying to buffer releases until the system runs low on free memory, the run-time layer simply attempts to filter out the obviously bad release requests. There are two cases that are checked: releases for pages that are not currently in memory, and multiple releases for the same page. The first case occurs when the compiler has not inserted releases early enough and the system paging daemon has already reclaimed the memory being released. We use the shared bit vector to detect this case, only issuing release requests to the operating system when the corresponding bit is already set. The second case occurs when symbolic loop bounds cause the compiler to release too aggressively. We detect this case by buffering the last block of pages released and comparing that address with the address of the current release. If the addresses are the same, no action is taken, but if the addresses differ, then the previous block is actually released and the current block is buffered.

2.4 The Compiler/Run-time Layer Interface

The role of the compiler is to analyze the out-of-core program and decide what data needs to be prefetched,

Table 2: Operations inserted by the prefetching compiler.

Single page, prefetches only
`void prefetch(void *prefetch_address, unsigned priority, unsigned tag);`
`void prefetch_exc(void *prefetch_address, unsigned priority, unsigned tag);`
`void prefetch_indirect(void *prefetch_address, unsigned priority, unsigned tag);`
`void prefetch_exc_indirect(void *prefetch_address, unsigned priority, unsigned tag);`
Block prefetches
`void prefetch_block_aligned(void *prefetch_address, unsigned num_pages, unsigned priority, unsigned tag);`
`void prefetch_exc_block_aligned(void *prefetch_address, unsigned num_pages, unsigned priority, unsigned tag);`
`void prefetch_block(void *prefetch_address, unsigned num_pages, unsigned priority, unsigned tag);`
`void prefetch_exc_block(void *prefetch_address, unsigned num_pages, unsigned priority, unsigned tag);`
Single page prefetch and release
`void prefetch_release(void *prefetch_address, void *release_address, unsigned priority, unsigned tag);`
`void prefetch_exc_release(void *prefetch_address, void *release_address, unsigned priority, unsigned tag);`
Block prefetch and release
`void prefetch_block_release(void *pf_addr, void *rel_addr, unsigned num_pages, unsigned priority, unsigned tag);`
`void prefetch_exc_block_release(void *pf_addr, void *rel_addr, unsigned num_pages, unsigned priority, unsigned tag);`
Release only
`void release(void *release_addr, unsigned int priority, unsigned int tag);`
`void release_exc(void *release_addr, unsigned int priority, unsigned int tag);`
`void release_block(void *release_addr, unsigned int num_pages, unsigned int priority, unsigned int tag);`
`void release_exc_block(void *release_addr, unsigned int num_pages, unsigned int priority, unsigned int tag);`
End of loop marker
`void end_of_loop_nest(unsigned int loop_number);`

Table 3: Arguments passed to PagingDirected policy on user invocation

Argument	Description
`int op`	operation to perform, see Table 4
`vrange_t pf_range`	range to prefetch, base address and length
`vrange_t rel_range`	range to release, base address and length
`unsigned int type`	type of request (i.e., regular, indirect, exclusive), for statistics collection
`unsigned int tag`	static request identifier, for statistics collection

when the prefetch should be initiated to hide the latency, and what data can be replaced. Unfortunately, the compiler cannot always make these decisions with 100% certainty for reasons that will be discussed in Section 2.1. Thus, the interface between the compiler and the run-time layer must allow the compiler to make the results of its static analysis of the program available to the run-time layer during execution when more accurate decisions can be made.

The compiler expresses the decision to prefetch or release a particular data reference by inserting a procedure call in the program at the appropriate point to hide the expected latency of the prefetch. Descriptive procedure names indicate if the request is for a prefetch or release only or both, if the request is for a single page or a block of pages, and if the requested data will be modified (exclusive requests). This approach allows the run-time layer to optimize the actions performed for each case. Arguments to the procedure indicate the address of the requested data (both prefetch and release), the number of pages used in a block request, a unique tag identifying the static request number, and a priority for retaining released pages. The compiler also inserts a procedure call to mark the end of a particular loop nest, allowing the run-time layer to possibly perform some activity when leaving a given loop. The various procedure calls inserted by the compiler are listed in Table 2.

2.5 The Run-time Layer/Operating System Interface

The interface between the run-time layer and the operating system consists of a single additional system call which allows user-level programs to invoke operations on policy modules. The operating system then passes the request to the PagingDirected policy module, which in turn performs the specified actions. Table 3 lists the arguments passed to the operating system for use by the policy module. Table 4 lists the possible values for the *op* argument.

Table 4: Possible operations performed by PagingDirected policy module

Operation	Description
`PMOP_PREFETCH`	prefetch page (or range of pages)
`PMOP_RELEASE`	release page (or range of pages)
`PMOP_PREF_REL`	both prefetch and release, one call
`PMOP_INITBV`	initialize shared page
`PMOP_GETSTATS`	copy prefetch statistics up to user level
`PMOP_PFSTATS_SETUP`	record beginning of prefetch for statistics collection

Table 5: Application Characteristics on Irix

Name	Input Data Set	Memory Required: Absolute	Memory Required: % of Available	Original Execution Time (mins)
BUK	2^{24} 20-bit integers	206 MB	275%	13.2
CGM	sparse matrix with 15,167,342 non-zeros	206 MB	275%	48.5
EMBAR	2^{24} random numbers	134 MB	179%	20.3
FFT	256x128x128 matrix of complex numbers	235 MB	313%	34.1
MGRID	256x256x256 matrix	452 MB	600%	23.4
APPLU	5x5x62x62x62 matrices	219 MB	292%	17.1
APPSP	110x110x110 matrices	213 MB	284%	90.5
APPBT	5x5x64x64x64 matrices	189 MB	252%	48.4

2.6 Performance Results

We performed experiments on our Irix implementation using out-of-core versions of the NAS Parallel benchmark suite. The only modification made to the benchmarks for these experiments was to increase the size of the data sets used. Table 5 summarizes the characteristics of the versions used for these experiments.

2.6.1 Impact on Out-of-core Execution Time

In Figure 5, we show the execution times of the out-of-core programs, normalized to the original case. For each benchmark we show three bars: the original, unmodified program (**O**) and the program compiled to use both prefetching alone (**P**), and the program compiled to use both prefetching and releasing (**PR**). Each bar is broken down into four components. The top section is the time that the program was stalled waiting for a page to be fetched from the swap space. The second component (from the top) is time spent waiting for resources that are unavailable. This component includes time waiting for locks, time waiting for the CPU, and time waiting for memory. The third component is the time spent executing system code, and is primarily the time spent in the fault handling code, since these applications use no other system services. The bottom section of each bar is the time spent executing user code.

Because we use separate pthreads to issue the prefetch requests, the prefetch service time does not appear in the execution time of the main application. Since we are using a multiprocessor, many of the prefetches can be serviced in parallel. Although the prefetch threads may compete with the main application for CPU time, it is an extremely small effect since these threads perform very little computation and spend most of their time waiting for I/O's to complete.

Figure 5 shows the overall performance results. I/O stall reductions range from 23% to over 99%, with 6 of the 8 applications achieving reductions of over 65%, both when prefetching is used alone and when it is combined with releasing. These reductions are quite promising, given that Irix is a full-featured commercial operating system which already attempts to automatically prefetch page faults whenever possible, reducing the original I/O stall time component in applications with reasonably predictable access patterns. Despite the efforts undertaken by the operating system significant amounts of I/O stall remain, which our system is able to reduce dramatically.

The surprising result shown here is the dramatic effect that prefetching and releasing can have on resource contention in the Irix system. The primary source of contention in the Irix system is locks on memory objects in the process's virtual address space which are needed whenever the address space changes. In particular, they are needed by prefetches, releases, page faults, and the system daemon attempting to reclaim memory.

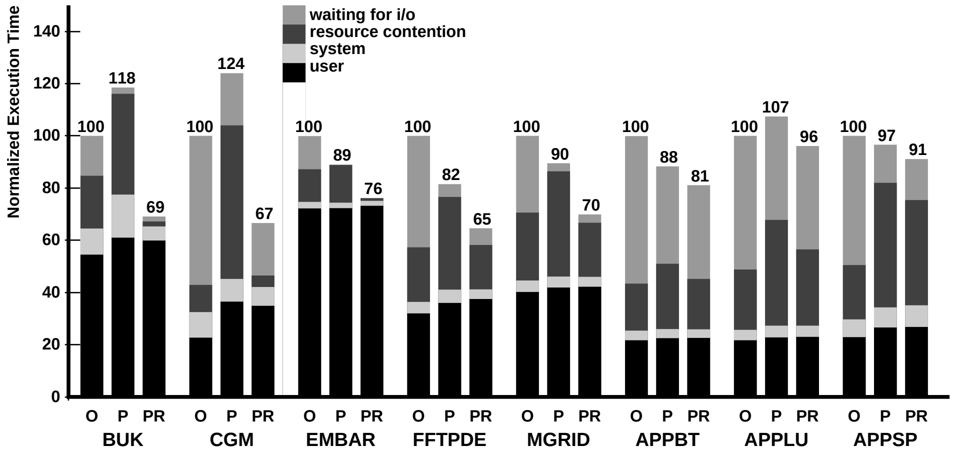


Figure 5: Impact of prefetching and releasing on the execution times of the out-of-core applications. (**O** = original; **P** = with prefetching only; **PR** = with prefetching and releasing.)

Even without prefetching and releasing, there is a fair amount of contention between the system paging daemon reclaiming memory and the out-of-core application page faulting. When prefetching alone is used, memory is consumed much faster and the contention between the paging daemon and the application is greatly increased. When releasing is added, however, these hot locks become less contended. Because memory is freed explicitly through releases which only require the locks for a very short period of time, the system paging daemon does not need to run as frequently and doesn't interfere with the out-of-core process's paging activity. In all cases, the contention when releasing is used together with prefetching is less than that which occurs for prefetching alone. In fact, releasing can also dramatically reduce contention over the original case, as for Buk, Cgm, and Embar. Contention is reduced by a small amount over the original cases for Fftpde and Mgrid. These five applications represent cases where the compiler and run-time layer are largely successful at prefetching and releasing. For the remaining three cases, however, prefetching and releasing are less successful. Since memory is not being freed effectively by the application, the paging daemon is still required to reclaim pages and the amount of contention for locks and physical memory increases, nearly doubling for Appsp in the worst case. Applu, Appsp and Appbt all contain multi-dimensional loops with symbolic bounds, in which the inner dimensions are relatively small. This situation is extremely difficult for the compiler to analyze statically, and results in sub-optimal prefetching and releasing decisions. The current run-time layer is able to discard bad decisions made by the compiler, but lacks the information needed to actually make the correct decisions. To overcome these difficulties, the compiler needs to produce code that can be specialized by the run-time layer as important quantities such as loop bounds become known during execution.

2.7 The Next Step

In the immediate future, we will be working on improving our automated system for virtual page fault prefetching in two important ways. The first item focuses on improving the way in which the run-time layer handles release requests. The second item focuses on improving the scheduling of prefetch requests by the compiler. Over the long term, we will also be exploring further ways to specialize the code generated by the compiler at run-time.

2.7.1 Extensions to the run-time layer

As we discussed in Section 2.1.1, the compiler determines when it becomes *possible* to release a particular page, based on the type of locality of the references that access that page. In determining the locality of the references, however, the compiler assumes that the operating system will be using an LRU replacement policy, and that the amount of memory available will be the amount specified to the compiler. Inaccuracies in these assumptions may lead to the compiler inserting releases for pages that could actually be kept in

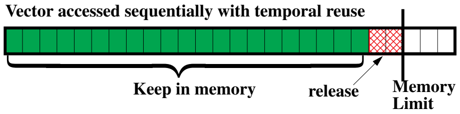


Figure 6: Most recently used replacement for a large vector with temporal reuse

memory. By buffering the release requests, the run-time layer can delay releasing memory until free memory actually becomes scarce. In some cases, it may not be necessary to perform any releases at all. In other cases, it may be possible for the run-time layer to implement an application-specific replacement policy that performs better than LRU for that application. To allow the run-time layer to make these decisions, we pass the results of the compiler's reuse analysis as a priority parameter in the release calls.

As an example of how the run-time layer could implement a better replacement scheme, consider the simple case of a program that repeatedly touches a very large array sequentially. This is essentially the access pattern in EMBAR. If the entire array does not fit in physical memory, and we use an LRU replacement policy, then each page of the array will need to fetched from backing store on each pass through the array. A better approach is to retain the initial portion of the array in memory, and discard the most recently used pages to make room for the data that is about to be accessed. The situation is illustrated in Figure 6. In this case, the run-time layer knows that the data will be reused, based on the priority given by the compiler. Releases are buffered until the operating system indicates that memory needs to be freed, at which point the most recently released pages are returned to the operating system first.

A second situation in which we can improve the replacement policy based on application-specific knowledge is choosing between multiple releasable references. The simplest case is two references, where one has reuse and the other does not (neither has locality). The run-time layer can choose to buffer the releases for the reference with reuse, while immediately issuing the releases for the other references. If free memory still becomes scarce, the run-time layer can then begin to issue some of the buffered requests. For multiple references with different types of reuse, we want to keep multiple queues of buffered releases, based on the priority assigned by the compiler. Releases would then be issued from the lowest priority queue when memory needs to be freed. We are currently working on the implementation of this multiple release buffer approach.

2.7.2 Improvements to the compiler scheduling algorithm

Recall in Figure 4(c), the compiler has chosen the j-loop as the software pipelining loop, based on the heuristic described in Section 2.1.2. A prolog is inserted before the j-loop to initialize the prefetching pipeline, and an epilog is inserted after the j-loop to drain it. Note, however, that this sequence of filling and draining the pipeline is executed on every iteration of the surrounding i-loop! Each time, some I/O stall is experienced until the prolog prefetches arrive in memory. A better approach is to pipeline across all the loops, initializing the pipeline with a prolog before the outermost loop, and only draining the pipeline at the end of all the loops.

This approach creates problems for generating the prefetch address, however. In the example of Figure 4, the prefetch distance is computed in terms of iterations of the pipeline loop, and the prefetch address is generated by simply adding an offset to that loop index in the array address computation. If we instead pipeline across all the loop nests, we need to compute the prefetch distance (and hence the addressing function) as offsets for each loop index. Essentially, we want to compute how many iterations of each loop are needed to completely hide the latency, beginning with the outermost. The equations for calculating the offset for each loop index are given below, d_i is the depth or offset for loop index i, d_0 is the offset for the outermost loop index, l_m is the total latency that needs to be hidden, while l_i is the latency not hidden by the surrounding offsets, s_i is the number of instructions in the shortest path through loop i.

$$d_0 = \left\lfloor \frac{l_m}{s_0 + l_p} \right\rfloor$$

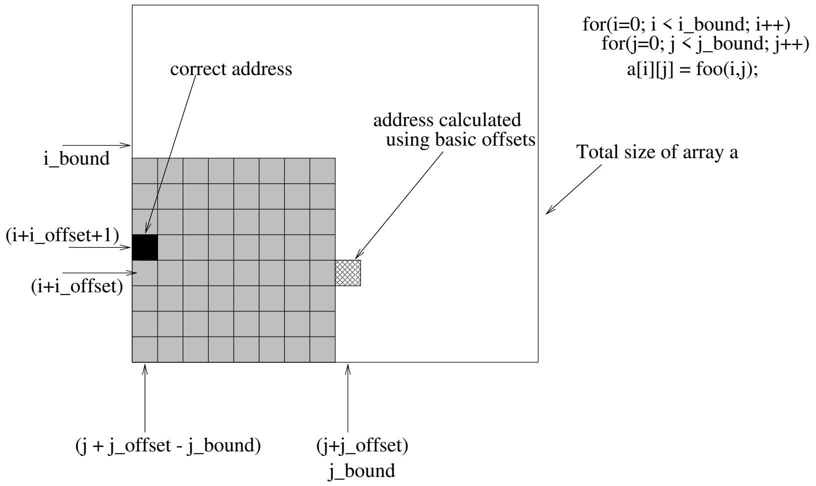


Figure 7: Computing the correct prefetch address with multi-dimensional software pipelining.

$$l_i = l_{i-1} \bmod (s_{i-1} + l_p)$$

$$d_i = \left\lfloor \frac{l_i}{s_i + l_p} \right\rfloor$$

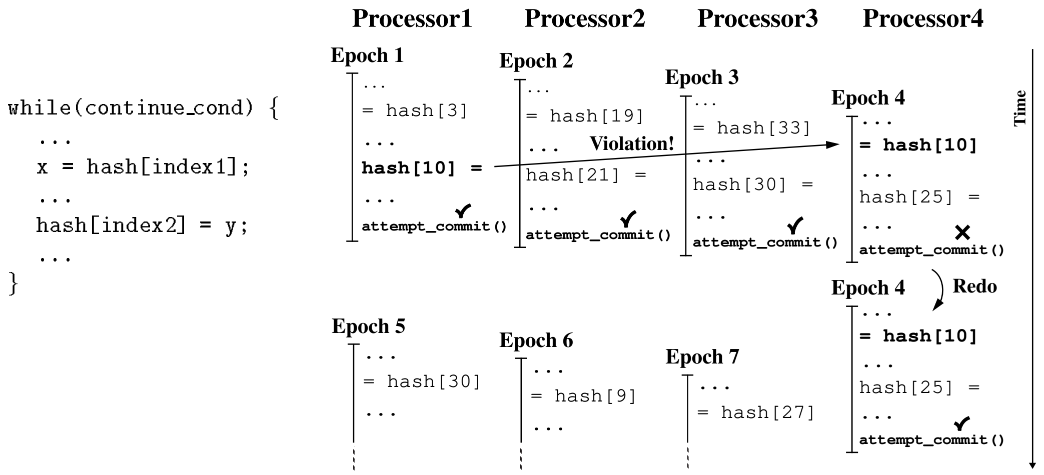
Note that in the equations above, we take the floor of the depth, rather than the ceiling, since we are using the interior loops to account for the leftover latency. For the innermost loop, we still want to take the ceiling.

While these equations give us the basic offsets, we need to take extra care when the loop index plus the offset exceeds the upper bound of the loop. With a single pipelining loop, this occurred at the point where we switched from the steady state to the epilog and began draining the pipeline. Now, however, we do not want to drain the pipeline until all the indexes plus offsets are at their upper bounds. Instead, when the index and offset exceed the bound, we need to adjust the offset and the offsets for all the surrounding loop to compute the correct address. Figure 7 shows a simple two dimensional example in which only a portion of the array is being accessed. In this figure, i_offset and j_offset are the amounts added to each of the loop index variables to calculate the address to prefetch. As we execute the inner loop, eventually j+j_offset will reach the upper bound of the region we want to access. At this point, `a[i+i_offset][j+j_offset]` is the element shown in the hashed box, which is outside the region we want to access. The correct element to prefetch is actually the first element of the next row, which we get by incrementing the offset for i by one, and decrementing the offset for j by `j_bound`. That is, we want to prefetch `a[i+i_offset+1][j+j_offset-j_bound]`. Ideally, we want the compiler to generate code to adjust the offsets and keep the prefetch address calculation correct.

Presently, we have "hand-compiled" several simple loops to perform software pipelining across all loop nests. We are working on using these hand implementations as a model for the compiler implementation.

3 Thread-Level Data Speculation

Thread-**L**evel **D**ata **S**peculation (**TLDS**) [10] and other similar techniques [6, 8] allow the compiler to automatically parallelize portions of code in the presence of statically ambiguous data dependences, thus extracting parallelism between whatever dynamic dependences actually exist at run-time. To illustrate how TLDS works, consider the simple `while` loop in Figure 8(a) which accesses elements in a hash table. This loop cannot be statically parallelized due to possible data dependences through the array `hash`. While it is possible that a given iteration will depend on data produced by an immediately preceding iteration,



(a) Pseudo-Code **(b)** TLDS Execution

Figure 8: Example of TLDS execution.

these dependences may in fact be infrequent if the hashing function is effective. Hence a mechanism that could speculatively execute the loop iterations in parallel—while rewinding and re-executing any iterations which do suffer dependence violations—could potentially speed up this loop significantly, as illustrated in Figure 8(b). Here a *read-after-write* (RAW) data dependence violation is detected between *epoch 1* and *epoch 4*, so *epoch 4* is rewound and restarted to produce the correct result. This example demonstrates the basic principles of TLDS—it can also be applied to regions of code other than loops.

In this example we assume that the program is running on a shared-memory multiprocessor, and that some number of processors (four, in this case) have been allocated to the program by the operating system. Each of these processors is assigned a unit of work, or *epoch*, which in this case is a single loop iteration. We timestamp each epoch with an *epoch number* to indicate its ordering within the original sequential execution of the program. We say that *epoch X* is *"logically earlier"* than *epoch Y* if their epoch numbers indicate that *epoch X* should have preceded *epoch Y* in the original sequential execution. Any *violation* of the data dependences imposed by this original program order is detected at run-time through the TLDS mechanism. Finally, when an epoch is guaranteed not to have violated any data dependences with logically-earlier epochs and can therefore commit all of its speculative modifications, we say that the epoch is *homefree*. We provide this guarantee by passing a *homefree token* at the end of each epoch. Further examples of the use of TLDS, and an exploration of the interface between TLDS hardware and software, can be found in an earlier publication [9].

In the remainder of this section, we will describe the TLDS system in detail, including the API, compiler support, and run-time support. We investigate two types of run-time support: a software DSM (distributed-shared memory) system for exploting coarse-grain parallelism, and aggressive hardware support that expoits more fine-grain parallelism.

3.1 TLDS API

We now define an interface between TLDS-compiled software and the speculative run-time support. This interface must be flexible enough to support different modes of execution. Another goal is for APIs for the software-only and hardware runtime systems to be similar so that the compiler may easily target both.

3.1.1 Hardware Run-time Support API

To begin, we assign primitives to all TLDS operations. These primitives are simplified to the point that they each can be implemented as a single RISC instruction. This way, our interface is flexible and we can defer design decisions until later. This interface is shown in table 6.

Operations	Description
`thread_descriptor_t tlds_fork()`	Create a new thread to execute an epoch.
`void tlds_end_thread()`	End the current thread.
`void tlds_set_sequence_number(sequence_number_t sn)`	Set the sequence number of the current thread.
`int tlds_become_speculative()` `void tlds_become_nonspeculative()`	Delimit start and end of speculative region.
`void tlds_wait_for_homefree_token()` `void tlds_pass_homefree_token(thread_descriptor_t td)`	Homefree token passing.
`void tlds_commit_speculative_writes()`	Release buffered writes to other epochs.
`int tlds_set_cancel_handler()` `void tlds_cancel_thread(thread_descriptor_t td)`	Allow epochs to be terminated.
`void tlds_suspend_speculation()` `void tlds_resume_speculation()`	Allow non-speculative stores in a speculative region.
`void tlds_violate_thread(thread_descriptor_t td)`	Trigger a violation.
`sp_t tlds_save_sp()` `void tlds_restore_sp(sp_t saved_sp)`	Stack management.
`void tlds_forwarding_frame(void *forward_struct)` `void tlds_forwarding_size(int size)`	Define region for forwarded values.

`void tlds_wait_for_value(unsigned offset)`
`void tlds_send_value(thread_descriptor_t td,`
`unsigned offset,`
`int value)`
	Forward values between epochs.

Table 6: Primitive Hardware TLDS operations.

In our hardware implementation, we implement each of these primitives as a single RISC instruction. We assume the availability of a low-cost, lightweight fork operation that allows us to quickly spawn new threads. This allows us to associate a new thread with every epoch, and identify an epoch by it's thread descriptor (the handle in many of our primitives).

3.1.2 Software-Only Run-time Support API

While the interface shown in Table 6 is efficient for our hardware implementation and for exploiting fine-grain parallelism, we require a more efficient interface for the software DSM implementation. We bundle these primitives into higher level functions, and assume that a single thread will execute many epochs. Conceptually, we assume that epochs will communicate less often and at a larger granularity. Table 7 defines our high-level primitives for the software-only support for TLDS. We will now define each of these primitives in detail.

During initialization, one process is spawned for each processor in the system. After allocating initial shared memory regions is performed, each process should execute the main loop of the system by calling `tlds_start()`. This is responsible for fetching and executing epochs.

Shared memory should be managed by calls to `tlds_mem_alloc()` and `tlds_mem_free()`, which are the equivalents of `malloc()` and `free()` for shared *speculative memory* (we will use that term to distinguish the shared memory image provided by the software-TLDS layer from the actual shared memory used for the central data structures in its implementation). Since the allocation of a memory block is performed by one of the processors, some mechanism is needed to initially broadcast a pointer to the first shared block. This can be done with the `tlds_mem_send()` and `tlds_mem_recv()` functions, which normally need to be called only once, during the initialization phase.

In order to have well-defined entry-points, the code for each epoch should be enclosed within a procedure[1]. An epoch can be created with `tlds_epoch_new()`. This can be thought of as equivalent to thread initialization in the single-thread model of execution. Each epoch is represented by an opaque data structure, and can be accessed from any processor (see 3.3.1 for details). Forwarding buffers are associated with each epoch, both for data that are forwarded during epoch creation (forwarding frame), as well as at a later stage of execution. A number of forwarding flags (for synchronization between the sender and recipient) are also associated with each epoch. All these need to be declared during the epoch creation stage, along with the epoch sequence number. Sequence numbers can be obtained and reserved with `tlds_seq_get()`.

[1]The compiler uses forwarded pointers to data in enclosing scopes, therefore this design choice does not impose any other restrictions.

Operation	Description
`tlds_init(argc, argv)`	Initialization of processes, shared-memory data structures and synchronization primitives.
`tlds_shutdown()`	Cleanup
`tlds_start()`	Start main epoch execution loop
`tlds_mem_alloc(size)` `tlds_mem_free(ptr)`	Shared memory management
`tlds_mem_send(ptr, len)` `tlds_mem_recv()`	Initialization of pointers to shared memory
`epoch = tlds_epoch_new(seq, &proc, fwd_frame, fwd_frame_size, fwd_data_size, fwd_num_flags`	Creation of new epoch data structure (contains forwarding buffers and flags).
`tlds_epoch_fork(epoch)`	Insertion of epoch into pending queue
`tlds_epoch_begin()`	Initialization of twin lists for current epoch
`tlds_epoch_end()`	Move epoch into finished queue and jump to beginning of main epoch execution loop
`tlds_epoch_fwd_set(epoch, ofs, data, len)`	Copy data into forwarding buffer of designated epoch
`tlds_epoch_fwd_set(epoch, ofs, ptr, len)`	Copy data from forwarding buffer
`tlds_epoch_fwd_wait(epoch, nflag)`	Wait for forwarding flag of designated epoch to be set (synchronization primitive)
`tlds_epoch_fwd_signal(epoch, nflag)`	Set forwarding flag
`tlds_epoch_cancel(epoch)`	Cancel epoch (invokes cancel handler)
`tlds_epoch_kill(epoch)`	Destroy designated epoch and free all related data structures (blocking operation)
`tlds_epoch_fail_handler(epoch, &hnd, arg)`	Set callback to invoke upon dependence violation
`tlds_epoch_cancel_handler(epoch, &hnd, arg)`	Set callback to invoke upon cancellation
`tlds_epoch_check_fail()`	Force early check for dependence violation
`tlds_epoch_check_cancel()`	Force early check for cancellation
`tlds_seq_get(nreserve)` `tlds_seq_next(seq)`	Sequence number management

Table 7: Software-TLDS operations.

After an epoch has been created, it can be scheduled for execution with `tlds_epoch_fork()`. In terms of the single-thread model, this can actually be regarded as thread creation[2], although in the actual implementation this operation appends the epoch to the pending queue (see 3.3.1).

Within each epoch procedure, any accesses to speculative memory have to be enclosed within calls to the `tlds_epoch_begin()` and `tlds_epoch_end()` pair of functions. These perform the necessary initial and final work (respectively) needed to track memory accesses. The latter also terminates the current epoch (returning execution to the main loop of the system).

The pair of `tlds_epoch_fwd_set()` and `tlds_epoch_fwd_signal()` are used to forward values during epoch execution. The operations `tlds_epoch_fwd_get()` and `tlds_epoch_fwd_wait()` should be used at the receiving end. Note that (in the current implementation) the latter blocks execution, until the desired value has been received.

The functions for setting violation and cancellation handlers, as well as canceling and killing an epoch are relatively straightforward. Epochs are never preempted, since this would impose a large overhead. Checks for violation or cancellation are always performed when an epoch finishes. However, an early check can be forced, if necessary, by calling `tlds_epoch_check_cancel()` or `tlds_epoch_check_fail()`. These can be inserted by the compiler and/or programmer before starting time-consuming work.

[2] Thread creation and initialization normally occur together, thus the correspondence to the RISC-like API for the compiler is rather straightforward (see also table 8).

Mechanism	Hardware API	Software API

Thread creation	`proc:`
	`  tlds_forwarding_frame(fwd_frame)`
	`  tlds_forwarding_size(fwd_frame_size)`
	`  tlds_fork()`
	`  tlds_set_sequence_number(seq)`
		`epoch = tlds_epoch_new(seq, &proc,`
		`      fwd_frame, fwd_frame_size, ...)`
		`tlds_epoch_fork(epoch)`

Epoch definition	`tlds_become_speculative()`	`tlds_epoch_begin()`

	`tlds_wait_for_homefree_token()`
	`tlds_become_nonspeculative()`
	`tlds_commit_speculative_writes()`
	`tlds_pass_homefree_token()`
	`tlds_end_thread()`
		`tlds_epoch_end()`

Epoch management	`tlds_violate_thread(td)` `tlds_cancel_thread(td)`	`tlds_epoch_cancel(epoch)` `tlds_epoch_kill(epoch)`
Value forwarding	`tlds_send_value(td, ofs, data)`	`tlds_epoch_fwd_set(epoch, ofs, data, len)` `tlds_epoch_fwd_signal(epoch, flag)`
	`tlds_wait_for_value(ofs)`	`tlds_epoch_fwd_wait(epoch, flag)` `tlds_epoch_fwd_get(epoch, ofs, ptr, len)`

Table 8: Correspondence between APIs.

3.1.3 API Correspondence

Both the hardware and the softare-only interfaces implement the same high-level TLDS mechanisms. This makes it easy for our compiler to target either platform. Table 8 shows the correspondence between the software and hardware APIs.

In the next section we describe our compiler infrastructure, and how the compiler uses these API primitives to speculatively parallelize a program.

3.2 The TLDS Compiler

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, the 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, and breaking that region into epochs. The compiler optimizes the efficiency of TLDS execution by:

- selecting epochs that are large enough so that the parallelism we exploit via multiple threads of execution can offset the overhead cost associated with thread creation;
- defining the epoch boundaries so that the number of data dependence violations between epochs is minimized;
- scheduling synchronization between frequently occurring data dependences to reduce the number of data dependence violations;
- and scheduling instructions and synchronizations in so that the distance between the corresponding producer and consumer of each cross-epoch dependence is minimized.

In the remainder of this section we describe the compiler passes and other tools used to compile a program for TLDS execution. We begin by explaining the profiling passes, and then show the two frameworks for generating TLDS code: the interactive framework that we currently use, and the fully-automatic framework that we are working towards.

3.2.1 Control Flow and Data Dependence Profiling

To generate efficient TLDS code, the compiler needs to understand certain properties of the target application: the frequency that certain regions are executed; the amount of potentially parallel computation in consecutive epochs; and the locations of highly-probable inter-epoch data dependences. It is not always

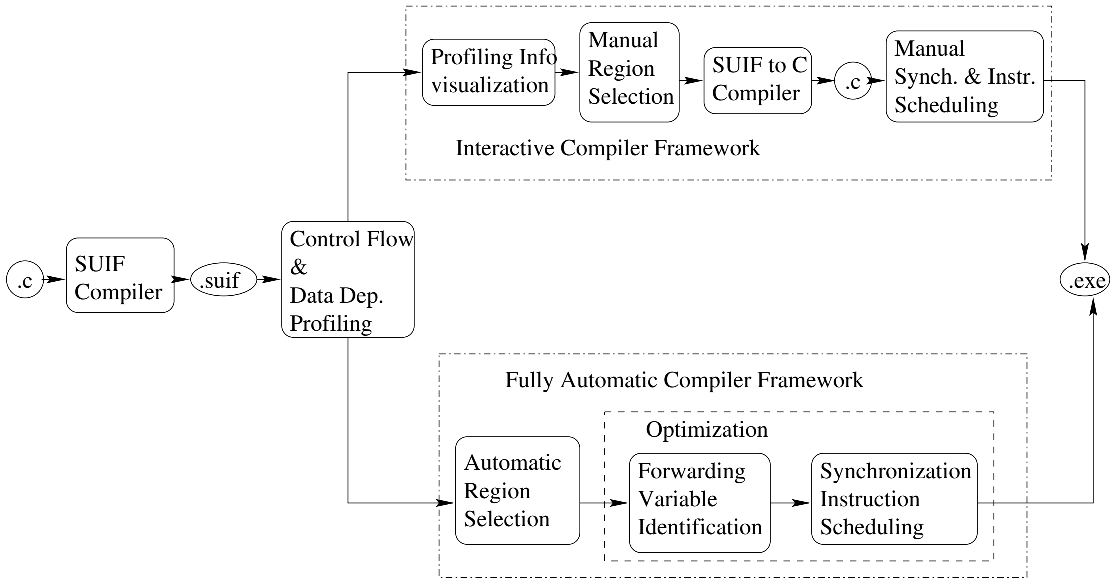


Figure 9: Two Compiler Frameworks

possible to implement these metrics using static analyses, so we must use profiling information as well. Our profiler collects two types of information: control flow behavior and data dependence behavior. The profiler is implemented with the following stages:

1. Assign a unique identification number to each object of interest (control flow constructs, uses and definitions of scalar variables, and individual loads and stores);

2. Insert calls to profiling routines at the appropriate locations in the original program, parameterized by the unique identification number;

3. Compile and execute the resulting program, collecting statistical data.

4. Annotate the original program with the statistical data for use by future compiler passes.

3.2.2 An Interactive Compiler

The first stage of our compiler development is a semi-automatic interactive compiler, where programmer intervention is required for two key components: region selection and instruction scheduling. In subsequent work, both processes will be automated. Figure 9 shows both compilation systems. The manual portions of the compilation process are aided by tools as follows:

- A visualization interface displays the control flow and data dependence information. This interface allows the programmer to manually select epochs to be parallelized.

- After the application is parallelized, it is translated back to C source code form. At this point, the programmer is required to schedule instructions and synchronizations to minimize the synchronization distance and optimize the performance.

3.2.3 An Automated Compiler

In the subsequent work, region selection and instruction scheduling will be automated as shown in Figure 9. A fully automated compilation system will allow us to speculatively parallelize benchmarks with large amounts of source code, and will also allow us to find more speculative regions.

The automatic region selector takes two forms of input: the original program, and the profiling information. The region selector computes an estimation of epoch size from control flow information, and an

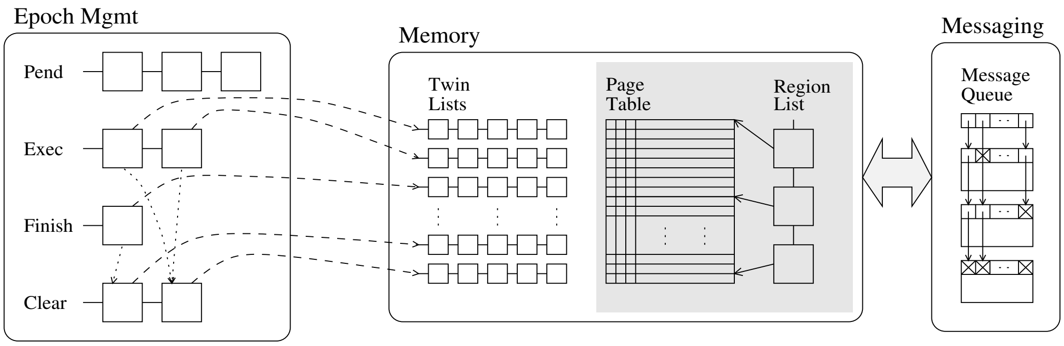


Figure 10: High-level view of software-only implementation. Shaded structures are *not* located in shared memory. The messaging subsystem is used to broadcast necessary updates, as well as during initialization (and for any other exchange of information that do not fit in central epoch queues and linked structures.

estimation of independent computation from data dependence information. The optimization stage contains two components. First, the synchronization identification step recognizes frequently occurring data dependences between epochs and inserts synchronizations as needed. Second, the instruction and synchronization scheduling step moves the producers of inter-epoch dependences as early as possible in the epoch and moves the consumers of inter-epoch dependences as late as possible in the epoch.

In the next two sections, we describe two implementations of TLDS runtime support: a software-only implementation built on top of a software DSM system, and a more aggressive hardware implementation that extends an invalidation-based cache coherence scheme.

3.3 Software-only Implementation

One way to provide support for TLDS is through a software-only library, implemented on top of already available hardware platforms. In order to facilitate speculation, we need a mechanism to track speculative reads and writes. The virtual memory hardware can be used to that effect. This approach was inspired by software-DSM systems, which provide the abstraction of shared memory across a collection of machines that do not already support a shared memory image in hardware. There has been a significant amount of research in this area and a number of successful systems have been built [3, 11, 7].

Each of these systems augments the virtual memory hardware by trapping to user-level code in response to access violations on shared pages. By setting the page protections appropriately, it is possible to track accesses to regions of shared memory, and by exchanging messages between different threads, the local copies of the shared memory image in each processor can be brought up-to-date.

However, our goal is not to provide support for shared memory on cheap platforms that do not already support it (for instance, networks of workstations), but rather to facilitate TLDS on systems that do not already support it. Therefore, we chose to build our software TLDS layer on top of a hardware shared-memory multiprocessor, such as the SGI Origin. The main advantage is the significantly reduced communication latency, which improves our chances of success.

In the following sections we will give an overview of the implementation and describe the current status and future work.

3.3.1 Implementation

As we have already mentioned, our base platform for software-TLDS is a hardware shared-memory multiprocessor. The implementation essentially employs a work-queue model and is built upon a central data structure, which is located in shared memory.

The main component of this data structure is a set of four queues. At any point in time, all epochs currently in the system can be found in one of these queues. Each queue contains epochs that are in various stages of execution:

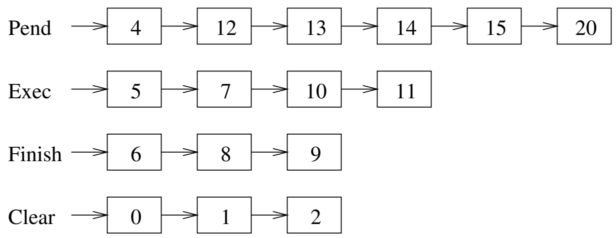


Figure 11: Central queue data structure (with cancelled/failed and restarted epochs).

- **Pending execution:** This queue contains epochs that have been scheduled for execution. Upon completion of an epoch, each processor fetches the next available epoch from this queue and begins executing it.
- **Executing:** This is simply a list of all epochs that are currently being executed by some processor in the system.
- **Finished execution:** Epochs are moved into this queue as soon as their execution has been completed, without checking for dependence violations (this may not be immediately possible). Upon completion of an epoch and before fetching the next, each processor accesses this queue and performs all necessary dependence checks for as many epochs as possible (an epoch is checked if all preceding ones haven't violated any dependences).
- **Clear to commit:** If the check for dependence violations is successful, an epoch is moved into this queue. An epoch needs to be kept in the system, until all processors have committed the changes made by it. Before beginning execution of an epoch, each processor aggressively updates its state, by traversing all epochs in this queue. A small "garbage collection" check is executed during this traversal, to destroy epochs whose state has been committed by all processors (and which are no longer pointed to as "last seen" epochs, as explained later).

The work performed on each queue comprises the the main loop of the system, which is started by `tlds_start()` (the actual implementation uses longjumps). A global lock is used to ensure mutual exclusion during accesses to the queues.

Epochs are also destroyed by kill operations, besides during the garbage collection phase. A kill operation blocks until all processors have finished executing their current epochs. Then, the state of each local image of the shared memory is restored to that of the epoch immediately preceding the one being killed. Finally, all epochs after that are purged from the system. We chose this implementation for various reasons. First of all, we expect kills to occur infrequently. Also, doing otherwise would introduce a large amount of complexity (and thus overhead) in managing the queues (and especially in the garbage collection function, which is frequently executed). Last but not least, we would not expect epochs succeeding a killed one to contain much useful work that would justify the extra overhead.

It should be evident that the epoch data structure (which is opaque to the programmer/compiler) is central. This is the data structure allocated by `tlds_epoch_new()` and contained in the queues described above. Besides the obvious information (such as sequence number, parent epoch sequence, executing processor, pointer to epoch procedure, etc.), the following information is associated with each epoch structure:

- **Forwarding frame:** This is simply a buffer containing a copy of the data forwarded to the epoch during creation; it is part of the epoch structure.
- **Forwarding buffer and flags:** This is space reserved for forwarding data after epoch creation. A flag is associated with each data item in the forwarding buffer (used for synchronization between sending and receiving epochs). These are also parts of the epoch structure and are allocated during its creation.
- **Twin list:** This is a pointer to a list of copies of all pages written by an epoch during its execution. Whenever a page is first written, a clean copy of the page is stored. When an epoch finishes execution, a copy of the final contents of each page is also made. The twin list is also allocated in shared memory.

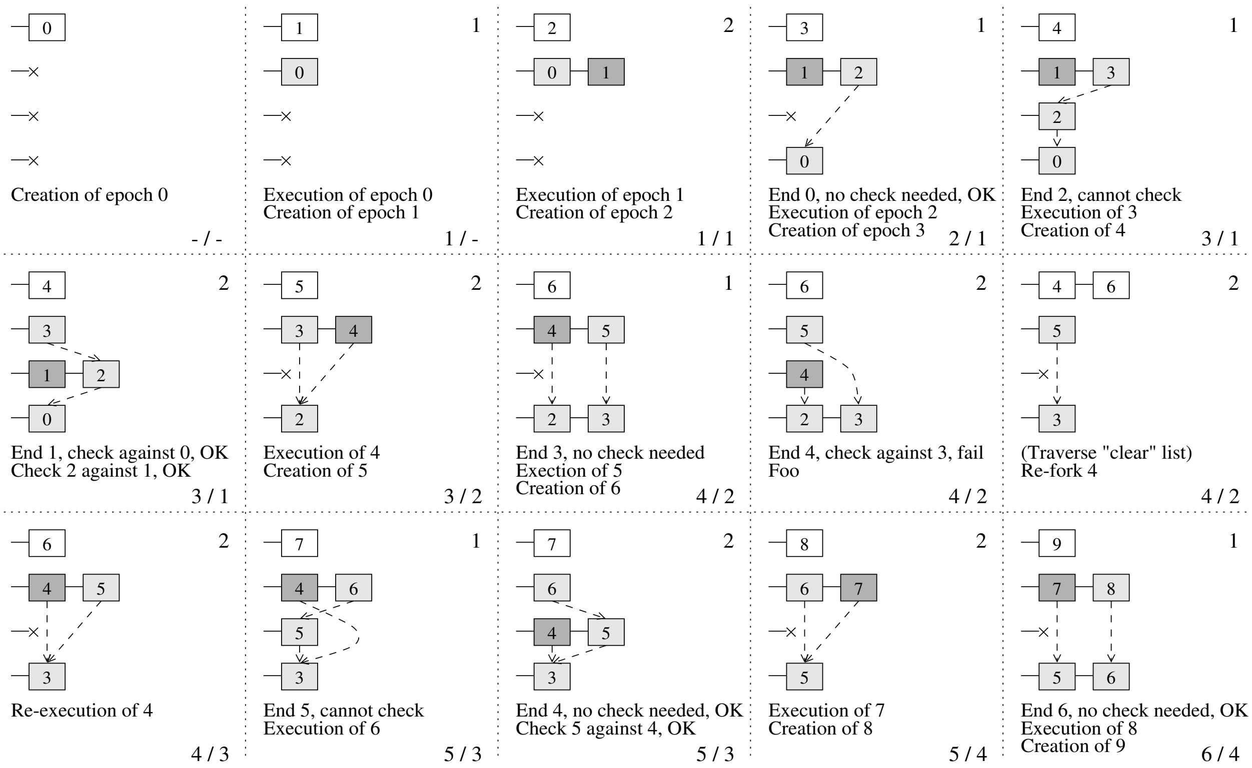


Figure 12: Central data structure during progress of execution, on a system with two processors. The "last seen" pointer is shown with a dashed line. The iteration number of the main loop for each processor is shown at the bottom right of each picture. The processor performing the actions listed in each picture is shown on the top right. Epochs executed on the first processor are lightly shaded, whereas epochs executed on the second processor are shaded dark.

All this information is used for dependence checking, as well as updating memory state (both during committing as well as undoing an epoch); a more detailed explanation will be provided later.

- **Last seen epoch:** This is simply a pointer to the epoch which was last committed on the processor that executed this epoch. This information is obviously used when checking for dependence violations (only epochs after the "last seen" one need to be checked against).

The above discussion brings us to the memory subsystem, which is the part whose design is based upon software-DSM systems. Each processor keeps a private copy of the shared speculative memory. Thus, each epoch reads and modifies only the local copy of the processor which executes it. The virtual memory system is used to track these accesses. Initially, each page is both read- and write-protected. Upon the first read (write) operation, a segmentation violation signal (SIGSEGV) occurs, which is trapped to user-level code. The signal handler sets a read (write) dirty bit and the read (write) protection is removed from the page, until the epoch completes execution. Thus, for each epoch, only the first read and the first write accesses will cause an expensive signal handler invocation.

We have incorporated one more important idea from software-DSM systems. Practically all such systems implement *multiple-writer* protocols. Whenever a page is first written, a *twin* is created, ie. a clean copy of its initial contents. By comparing the initial and final contents of the page, we can achieve two things. First, we can actually track writes at a word (or even byte) granularity and thus reduce *false sharing* problems. Also, if multiple processors (in our case, epochs) have modified a certain page, it is possible to merge these modifications. Software-DSM systems always use *diffs* to exchange page modifications between processors. This has the advantage of reducing the size of exchanged messages, and thus the communication overhead. However, since our TLDS layer is built on top of a tightly-coupled hardware multiprocessor, the overhead of computing the diffs is usually greater than exchanging the entire "before" and "after" contents of a page through shared memory. Therefore, we do the equivalent of diffing[3] *only* when it is necessary to merge modifications, and not when checking for dependence violations.

There are three possible types of dependences. Each of those poses different problems and is dealt with as follows:

- **WAR:** Whenever an epoch begins execution, any modifications by succeeding epochs are undone (this may be necessary if the newly executing epoch is one which was previously violated or cancelled). Since each epoch accesses the local copy of the speculative memory, this type of dependences does not pose any problem.
- **WAW:** The support for multiple-writers with on-demand diffing resolves such dependences.
- **RAW:** The good news is that this is the only type of dependence that may cause an epoch to fail. The bad news is that these dependences are difficult to identify correctly. Since we rely solely on the virtual memory hardware, reads can only be tracked at a page granularity. Therefore, whenever a page is both written by an epoch and read by an earlier one, we have to conservatively assume that there may have been a dependence violation, even though the sets of words read and words written may be disjoint (we have no way of identifying the former). Of course, if we do not rely exclusively on the virtual memory mechanism, we can deal with this problem of false sharing. For instance, Shasta [7] uses compiler-inserted instructions to tag the precise address of each read. However, this imposes a significant overhead on each read operation (at least those to speculative memory). Our system does not currently tag reads; detailed performance evaluation with real applications should reveal whether this is an important issue.

We have implemented our own equivalents of `malloc()` and `free()` for managing the shared speculative memory. The speculative memory heap consists of a list of regions. New regions are allocated whenever necessary. A relatively simple message-passing subsystem is used for exchanging the information necessary to coordinate these operations. The implementation is based on a shared message queue and supports "broadcasting" (each message may belong to as many chains as the processors in the system), message type tags and both blocking and non-blocking send/receive operations; the design should be general enough to support future needs, while the system evolves. This message passing subsystem is currently also used during initialization to broadcast any information necessary, as well as for the implementation of the `tlds_mem_send()/tlds_mem_recv()` pair of functions. These are normally used during the initialization phase to broadcast the address of the initial block of shared memory.

3.3.2 Current status

We have implemented a prototype software-TLDS system, which runs on both IRIX and Linux multiprocessor systems. Development was done from scratch, since the use of shared data structures for communications (rather than message passing) alters the design significantly, so that reusing code from existing software-DSM implementations would offer negligible, if any, benefits.

An initial evaluation of implementation overheads using a synthetic benchmark has yielded speedups of at least a factor of three, using four processors on the SGI Origin. This is under ideal conditions (in terms of the frequency of violations), but includes the overheads of trapping memory accesses and maintaining the shared data structures and twin copies for each epoch. We are currently in the process of evaluating actual performance using real application benchmarks.

3.4 Aggressive Support Using Hardware

It is also possible to implement support for TLDS directly in hardware. Rather than augmenting a software DSM, we can add support for detecting data dependence violations and buffering speculative state by extending a standard invalidation-based cache coherence scheme. First, we give an example of how coherence can be extended to detect data dependence violations. Next, we describe our coherence scheme in detail, then we evaluate its performance on a scalable benchmark.

[3] Diffing is performed at a word granularity by default (smaller granularities impose a high overhead, mainly because of inefficient memory accesses), but any granularity can be chosen during system startup.

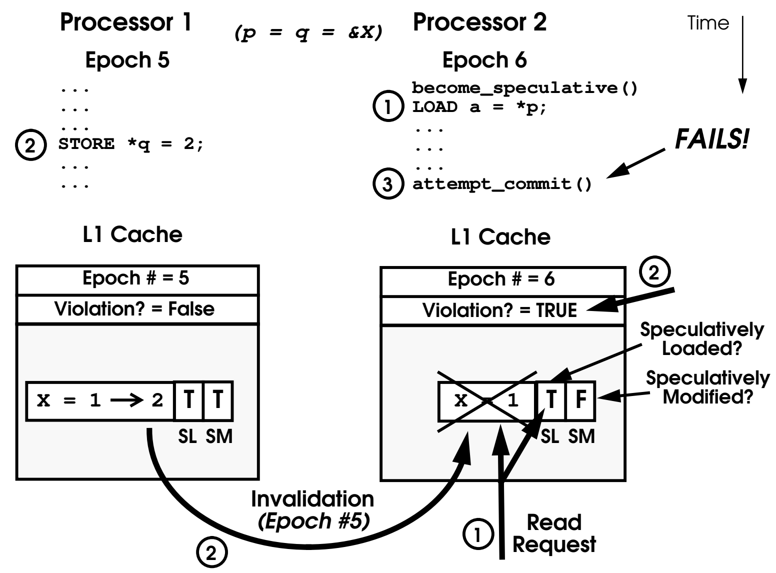


Figure 13: Using cache coherence to detect a RAW dependence violation.

3.4.1 Example

To demonstrate how invalidation-based cache coherence can be extended to track data dependences, we give an example of the detection of a read-after-write (RAW) dependence violation. Recall that a given speculative load violates a RAW dependence if its memory location is subsequently modified by another epoch such that the store should have preceded the load in the original sequential program. As shown in Figure 13, we augment the state of each cache line to indicate whether the cache line has been speculatively loaded (SL) and/or speculatively modified (SM). For each cache, we also maintain a number which indicates the sequential ordering of that epoch with respect to all other epochs (the *epoch number*), and a flag indicating whether a data dependence violation has occurred.

In the example, epoch 6 performs a speculative load, so the corresponding cache line is marked as speculatively loaded. Epoch 5 then stores to that same cache line, generating an invalidation containing its epoch number. When the invalidation is received, three things must be true for this to be a RAW dependence violation. First, the same cache line that the invalidation is for must be present in the cache. Second, it must be marked as speculatively loaded. Third, comparison of the current epoch number with the epoch number of the invalidation must tell us that the invalidation came from a logically-earlier epoch. Since all three conditions are true, a RAW dependence has been violated: epoch 6 is notified by setting the violation flag. As we will show, the full coherence scheme must handle many other cases, but the overall concept is analogous to this example.

In the sections that follow, we define the new speculative cache line states and the actual cache coherence scheme, including the actions which must occur when an epoch becomes *homefree* or is notified that a violation has occurred. We begin by describing the underlying architecture assumed by the coherence scheme.

3.4.2 Underlying Architecture

The goal of our coherence scheme is to be both general and scalable. We want the coherence mechanism to be applicable to any combination of single-threaded or multithreaded processors and shared-memory architectures, not necessarily restricted to multiprocessors on a single chip.

For simplicity, we assume some shared-memory architecture that supports an invalidation-based cache coherence scheme where all hierarchies enforce inclusion. Figure 14(a) shows a generalization of the underlying architecture. There may be a number of processors or perhaps only a single multithreaded processor, followed by an arbitrary number of levels of physically private caching. The level of interest is the first level where invalidation-based cache coherence begins, which we will call the *speculation level*. We generalize levels

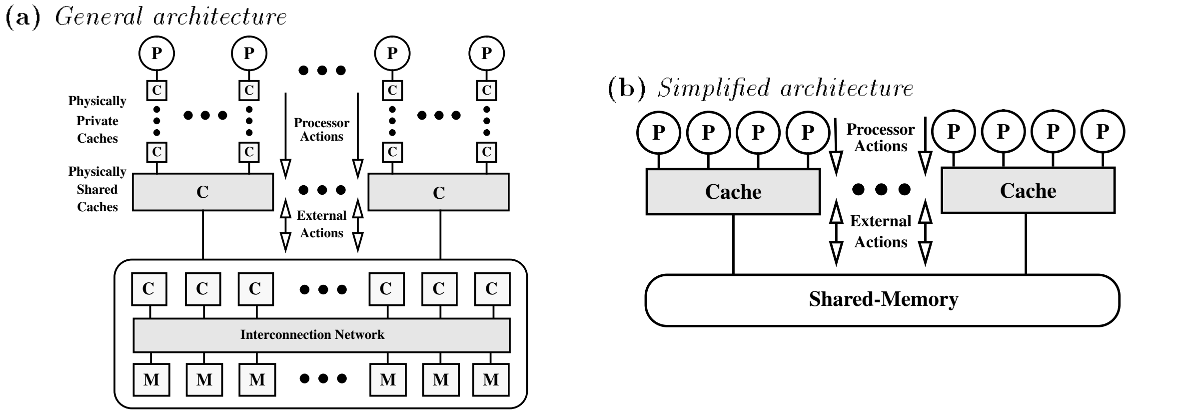


Figure 14: Base architecture for the TLDS coherence scheme.

of the system below the speculation level, as an interconnection network providing access to main memory with some arbitrary number of levels of caching.

The amount of detail shown in Figure 14(a) is not necessary for the purposes of describing our cache coherence scheme. Instead, Figure 14(b) shows a simplified model of the underlying architecture. The speculation level described above happens to be a physically shared cache and is simply referred to from now on as "the cache". Above the caches, we have some number of processors, and below the caches we have an implementation of cache-coherent shared memory.

Although coherence can be recursive, speculation only occurs at the speculation level. Above the speculation level (i.e. closer to the processors), we maintain speculative state and buffer speculative modifications. Below the speculation level (i.e. further from the processors), we simply propagate speculative coherence actions and enforce inclusion.

3.4.3 Overview of our Scheme

The remainder of this section describes the important details of our coherence scheme, which requires the following fundamental elements:

- a notion of whether a cache line has been speculatively loaded and/or speculatively modified;
- a guarantee that a speculative cache line will not be propagated to regular memory, and that speculation will fail if a speculative cache line is replaced;
- an ordering of all speculative memory references (provided by epoch numbers and the *homefree* token).

Following the description of our baseline scheme, we will show how we can improve performance by adding support for suspending violations and multiple writers. We also present a way to avoid scanning the cache for speculative cache lines, as well as other performance optimizations.

3.4.4 Line State in the Cache

A standard invalidation-based cache coherence scheme can be in one of the following states: *invalid* (I), *exclusive* (E), *shared* (S), or *dirty* (D). The *invalid* state indicates that the cache line is no longer valid and should not be used. The *shared* state denotes that the cache line is potentially cached in some other cache, while the *exclusive* state indicates that this is the only cached copy. The *dirty* state denotes that the cache line has been modified and must be written back to external memory. When a processor attempts to write to a cache line, exclusive access must first be obtained—if the line is not already in the *exclusive* state, invalidations must be sent to all other caches which contain a copy of the line, thereby invalidating these copies.

To detect data dependences and to buffer speculative memory modifications, we extend the standard set of cache line states as shown in Table 9. For each cache line, we need to track whether it has been

Table 9: Cache Line States

State	Description
I	Invalid
E	Exclusive
S	Shared
D	Dirty
SpE	Speculative (SM and/or SL) and exclusive
SpS	Speculative (SM and/or SL) and shared

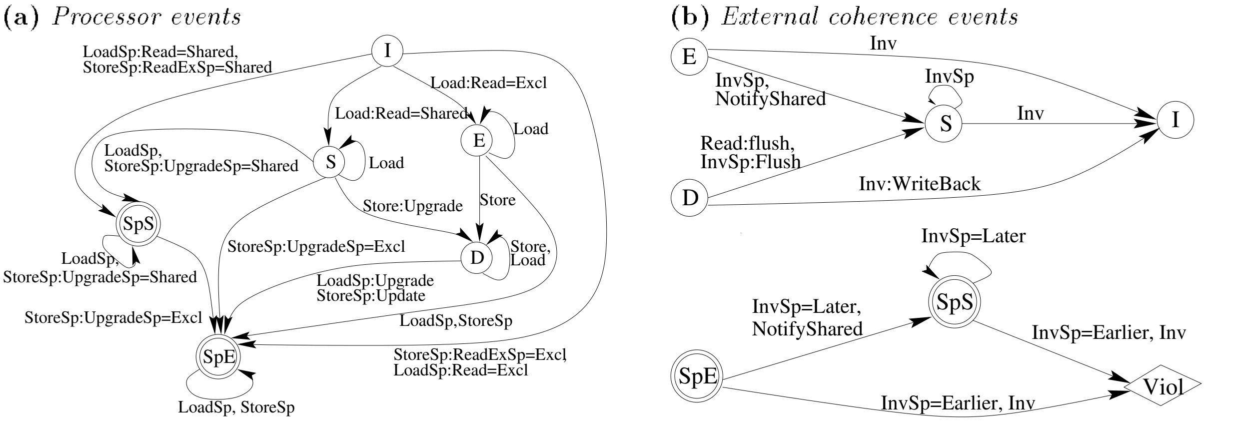


Figure 15: Basline coherence scheme for speculation

speculatively loaded (SL) and/or speculatively modified (SM), in addition to exclusiveness. Rather than enumerating all possible permutations of SL, SM, and exclusiveness, we instead summarize by having two speculative states: *speculative-exclusive* (*SpE*) and *speculative-shared* (*SpS*).

For speculation to succeed, any cache line with a speculative state must remain in the cache until the corresponding epoch becomes *homefree*. Speculative modifications may not be propagated to the rest of the memory hierarchy, and cache lines that have been speculatively loaded must be tracked in order to detect whether data dependence violations have occurred. If a speculative cache line must be replaced, then this is treated as a violation causing speculation to fail and the epoch is re-executed—note that this will affect performance but not correctness or forward progress. Previous work has shown that a 16KB, 2-way set-associative cache along with a four-entry victim cache is sufficient to avoid nearly all failed speculation due to replacement [10].

3.4.5 Coherence Messages

To support speculation, we extend standard invalidation-based cache coherence, as shown in Table 10, by adding three new speculative coherence messages: *read-exclusive-speculative*, *upgrade-request-speculative*, and *invalidation-speculative*. These new speculative messages behave similarly to their non-speculative counterparts except for two important distinctions. First, the epoch number of the requester is piggybacked along with the messages, so that the receiver can compare the logical ordering of the requesting epoch and the current epoch. Second, the speculative messages are only hints and do not compel the cache to relinquish ownership.

3.4.6 Baseline Coherence Scheme

We describe the coherence scheme for supporting TLDS using two state transition diagrams (given in Figure 15). The first shows transitions for processor-initiated events such as speculative and non-speculative loads and stores. The second shows transitions in response to coherence messages from the external memory system.

Table 10: Coherence Messages and Conditions

Message	Description
Read	Read a line.
ReadEx	Read-exclusive: return a copy of the line with exclusive access.
Upgrade	Upgrade-request: gain exclusive access to a line that is already present
Inv	Invalidation.
Writeback	Supply a copy of the cache line and relinquish ownership
Flush	Supply a copy of the cache line but maintain ownership
NotifyShared	Notify that the line is now shared
ReadExSp	Read-exclusive-speculative: return a copy of the line, possibly with exclusive access
UpgradeSp	Upgrade-request-speculative: try to gain exclusive access to a line that is already present
InvSp	Invalidation-speculative. Only invalidate the line if this is from a logically earlier epoch.
Condition	**Description**
=Shared	The request has returned shared access
=Excl	The request has returned exclusive access
=Later	The request is from a logically later epoch
=Earlier	The request is from a logically earlier epoch

First, let us summarize a standard invalidation-based cache coherence scheme. If a processor load misses we issue a read to the memory system, and if a store misses we issue a *read-exclusive*. If a store hits and the cache line is in the *shared* (*S*) state, we must issue an *upgrade-request* to obtain exclusive access. Exclusiveness is enforced by sending invalidations to all caches owning a copy of the cache line.

Important features of our speculative coherence scheme:

- When a speculative memory reference is issued, we transition to the *speculative-exclusive* (*SpE*) or *speculative-shared* (*SpS*) state as appropriate. For a speculative load we set the SL flag, and for a speculative store we set the SM flag.
- When a speculative load misses, we issue a normal read to the memory system. In contrast, when a speculative store misses, we issue a *read-exclusive-speculative* containing the current epoch number. When a speculative store hits and the cache line is in the *shared* (*S*) state, we issue an *upgrade-request-speculative* which also contains the current epoch number.
- When a cache line is in the state *speculative-shared* (*SpS*) and a *invalidation-speculative* is received, it is possible that a read-after-write dependence has been violated. This is the case if the *invalidate-speculative* is from a logically earlier epoch, and the processor is notified of the violation.
- When a cache line is *dirty*, the cache owns the only up-to-date copy of the cache line and must preserve it. When a speculative store accesses a *dirty* cache line we generate a flush, ensuring that the only up-to-date copy of the cache line is not corrupted with speculative modifications. For simplicity, we also generate a flush when a speculative load accesses a *dirty* cache line (we describe later how this case can be optimized).
- The goal this version of the coherence scheme is to slow down a non-speculative thread as little as possible. For this reason, a cache line in a non-speculative state is not invalidated when a speculative *upgrade-request* arrives from the external memory system. As an example, a cache line in the state *shared* (*S*) stays in the state *shared* (*S*) when an *invalidation-speculative* is received. Alternatively, the cache line could be relinquished in order to give exclusiveness to the speculative thread, possibly eliminating the need for that thread to obtain ownership when it becomes *homefree*. The experimental analysis of the performance benefits of these two options is beyond the scope of this paper.

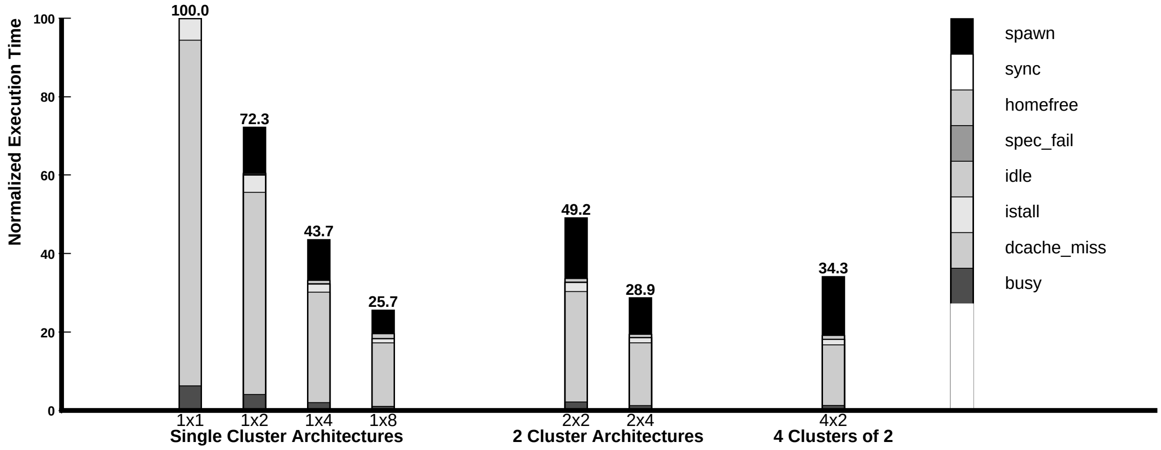


Figure 16: Region performance of `buk` on a variety of cluster architectures (*NxM* means N clusters of M processors).

When the *homefree* token arrives: Our scheme depends on ensuring that epochs commit their speculative modifications to memory in logical order. We implement this ordering by waiting for and passing the *homefree* token at the end of each epoch. When an epoch receives the *homefree* token, we know that all logically earlier epochs have completely performed all speculative memory operations, and that any pending incoming coherence messages have been processed—hence memory is consistent with the rest of the system. At this point, the epoch is guaranteed not to have violated any data dependences with logically earlier epochs and can therefore commit all of its speculative modifications. For each cache line that is in the *speculative-shared* state (*SpS*) and has been speculatively modified, we must issue an *upgrade-request* to acquire exclusive access to the cache line. Once it is exclusively owned, the cache line may transition to the *dirty* (*D*) state—effectively committing the speculative modifications to regular memory. Maintaining the notion of exclusiveness is therefore important since a speculatively modified cache line that is exclusive (*SpE*) does not require an *upgrade-request* to be sent out when modifications are committed to memory. On receipt of the *homefree* token, any cache line which has only been speculatively loaded immediately makes one of the following state transitions: from *speculative-exclusive* (*SpS*) to *exclusive* (*E*), or from *speculative-shared* (*SpS*) to *shared* (*S*). We will describe in the next section how this operation can be implemented efficiently.

On becoming *homefree*, it would take far too long to scan the entire cache for all speculatively modified and shared cache lines—ultimately this would delay passing the *homefree* token and diminish the performance of our scheme. We propose instead that the addresses of cache lines requiring ownership be stored in an *ownership required buffer* (ORB). This requires that we simply enter a cache line address into the ORB whenever a cache line becomes both speculatively modified and shared. When the *homefree* token is received we generate an *upgrade-request* for every entry in the ORB, and then pass the *homefree* token to the next epoch when all of the *upgrade-requests* have completed.

Recovering from failed speculation: If speculation fails for an epoch, all cache lines which have been speculatively modified by that epoch must be invalidated (changed to the invalid state). Any cache lines which has only been speculatively modified makes one of the following state transitions: from *speculative-exclusive* (*SpS*) to *exclusive* (*E*), or from *speculative-shared* (*SpS*) to *shared* (*S*). In the next section, we will describe how this operation can also be implemented efficiently.

3.4.7 Evaluation of Scalability

To show that our coherence scheme does in fact scale to a clustered multiprocessor, we evaluate a speculative version of the `buk` benchmark from the NAS-Parallel suite. `Buk` is an implementation of the bucket sort algorithm that has been reduced to its kernel, removing the dataset generation and verification code. Two important loops in `buk` both randomly access arrays through indirect references, so traditional parallelization techniques would be ineffective. We transformed the loops to execute speculatively in parallel using the TLDS primitives described in Section 3.1.1.

Figure 16 shows the performance of `buk` on a range of cluster architectures. Each bar is broken down into eight categories explaining what happened during all potential graduation slots.[4] The top four sections represent non-graduating slots attributed to the following TLDS-related reasons: waiting to begin a new epoch (*spawn*); waiting for synchronization for a forwarded location (*sync*); waiting to become homefree (*homefree*); and graduating speculative instructions for an epoch that is violated (*spec_fail*). The remaining sections represent regular execution: the *busy* section is the number of slots when instructions actually graduate and commit; the *dcache_miss* section is the number of non-graduating slots attributed to data cache misses; and the *istall* section is all other slots where instructions do not graduate. Finally, the *idle* section represents slots when a processor has nothing to execute.

First, we show execution on a single cluster with a varying number of processors (1x1), which shows that `buk` is mostly limited by data cache misses. This is because of the randomness of the key hashing in the main loops of the bucket sort routine. As we increase the number of processors in a cluster to eight (1xN), we see that the performance of `buk` continues to improve. Note that we spend an increasing amount of time waiting to start new epochs, but that we spend far less time stalled for data cache misses. This is because the larger architectures utilize more cache and more miss handlers.

We now focus on the multi-cluster architectures. In our cluster model, inter-cluster communication latency is twenty-times that of intra-cluster communication latency (i.e. 200 cycles rather than 10 cycles). We see that two clusters of two processors (2x2) give comparable speedup to one cluster of four (1x4), and that one cluster of eight (1x8), two clusters of four (2x4), and four clusters of two (4x2) also give comparable performance. These results are important because they demonstrate that TLDS mechanisms and overheads can scale, and that they will not limit the scope of our coherence scheme.

References

[1] A. K. Demke. Automatic I/O Prefetching for Out-of-Core Applications. Master's thesis, University of Toronto, Department of Computer Science, January 1997.

[2] IEEE. Threads Extension for Portable Operating Systems (Draft 7), February 1992.

[3] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the winter usenix conference*, January 1994.

[4] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[5] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

[6] Jeffery Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, February 1997.

[7] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[8] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of ISCA 22*, pages 414–425, June 1995.

[9] 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.

[4] The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles multiplied by the number of processors.

[10] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallellization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[11] M. Swanson, L. Stoller, and J. Carter. Making distributed shared memory simple, yet efficient. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, March 1998.