# Exploring Shared Memory Protocols in FLASH

Mary Hall, Robert Lucas, Jacqueline Chame (USC / ISI)
Mark Horowitz, Robert Kunz (Stanford University)

## ABSTRACT

The goal of this project was to improve the performance of large scientific and engineering applications through collaborative hardware and software mechanisms to manage the memory hierarchy of non-uniform memory access time (NUMA) shared-memory machines, as well as their component individual processors. In spite of the programming advantages of shared-memory platforms, obtaining good performance for large scientific and engineering applications on such machines can be challenging. Because communication between processors is managed implicitly by the hardware, rather than expressed by the programmer, application performance may suffer from ***unintended communication*** – communication that the programmer did not consider when developing his/her application.

In this project, we developed and evaluated a collection of hardware, compiler, languages and performance monitoring tools to obtain high performance on scientific and engineering applications on NUMA platforms by ***managing communication*** through alternative coherence mechanisms. Alternative coherence mechanisms have often been discussed as a means for reducing unintended communication, although architecture implementations of such mechanisms are quite rare. This report describes an actual implementation of a set of coherence protocols that support coherent, non-coherent and write-update accesses for a CC-NUMA shared-memory architecture, the Stanford FLASH machine. Such an approach has the advantages of using alternative coherence only where it is beneficial, and also provides an evolutionary migration path for improving application performance. We present data on two computations, RandomAccess from the HPC Challenge benchmarks and a forward solver derived from LS-DYNA, showing the performance advantages of the alternative coherence mechanisms. For RandomAccess, the non-coherent and write-update versions can outperform the coherent version by factors of 5 and 2.5, respectively. In LS-DYNA, we obtain improvements of 18% on average using the non-coherent version. We also present data on the SpecOMP benchmarks, showing that the protocols have a modest overhead of less than 3% in applications where the alternative mechanisms are not needed.

In addition to the selective coherence studies on the FLASH machine, in the last six months of this project ISI performed research on compiler technology for the transactional memory (TM) programming model being developed at Stanford. As part of this research ISI developed a compiler that recognizes transactional memory “pragmas” and automatically generates parallel code for the TM programming model.

Final Report
Contract No. DE-FG02-03ER25564

## 1 Introduction

Achieving good performance and scalability for scientific and engineering applications on large non-uniform memory access time (NUMA) shared-memory machines has been the focus of several research projects in the past decade, including the MIT Alewife [1], Stanford's DASH [14] and FLASH [12] and the Wisconsin Tempest [19]. NUMA shared-memory machines have their processors and memory physically distributed and interconnected by a scalable communications fabric, yet they provide users with a single global address space, a programming model that is easier to use than message-passing. In spite of the programming advantages of shared memory, obtaining scalable (and machine-independent) performance for large scientific and engineering applications can be challenging. Application performance may suffer from *unintended communication* -- communication that the programmer did not consider when developing her application -- because communication between processors is managed implicitly by the hardware, rather than expressed by the programmer.

Lately there has been research in improving the performance of large scientific and engineering applications through collaborative hardware and software mechanisms. In particular, there has been a growing interest in ``selective" data coherence on cache-based shared memory machines, that is, enforcing traditional coherence on selected data structures only. Selectively enforcing coherence requires the user or compiler's knowledge about how data objects are used throughout the program. For example if the user/compiler has knowledge that data objects are private, these local temporaries do not need to be written back to main memory once ``not live". As another example, if the user knows that data with random reference patterns does not have dependences across threads, then coherence traffic is unnecessary. By default, coherence is enforced, and one can think of designating data structures as non coherent as a performance optimization.

This project explored a collection of hardware, compiler, language and performance monitoring tools to achieve high performance on NUMA platforms by managing communication. As part of this project, we conducted an experiment on reducing one form of unintended communication, coherence traffic between processors, by selectively enforcing data coherence using a new set of cache-coherence protocols that allow memory accesses to data to be seen as coherent, *non-coherent* or *write-update* accesses at different times during a computation. A non-coherent access triggers the data's home node to ignore normal coherence operations and tag the data as non-coherent. A write-update access bypasses normal coherence operations and writes data directly to memory.

The protocols developed during this project were implemented in the Stanford FLASH hardware testbed. FLASH was developed to explore the efficient integration and support of both cache-coherent shared memory and user-level message passing. Unlike prior work in cache-coherence protocols, which focus on tuning the hardware implementation in a way that is transparent to the application programmer, the programming model for selective coherence allows the application programmer to express at a high level which data structures and in which portion of the code the alternative protocols should be used. This approach permits customization of protocols only where it is expected to be beneficial, and allows an evolutionary migration path for the application programmer to achieve high performance.

This report of is organized into five sections, followed by a conclusion. Section 2 presents an overview of FLASH. Section 3 describes the programming model for supporting selective coherence mechanisms. Section 4 discusses the baseline coherence protocol and the two selective-coherence protocols developed during this project, as well as their semantics and implementation. Section 5 presents results of two computations that can benefit from selective-coherence protocols, the RandomAccess benchmark from the HPC Challenge Benchmark suite and a forward solver derived from LS-DYNA [15], and reports the performance impact of selective coherence when it is not needed, on the SpecOMP benchmarks. Finally, Section 6 discusses our work in compiling for transactional memory.

## 2 Background

The Stanford FLASH system [12] was developed to explore the efficient integration and support of both cache-coherent shared memory and user-level message passing. FLASH is a 64-processor NUMA system based on the SGI Origin 2000. The primary difference between the two systems is Stanford's desire for FLASH to support multiple memory models, not just cache-coherent shared memory as implemented by the Origin. Included in the FLASH testbed is Flashpoint, a tool for monitoring the memory behavior of specific address ranges, which provides insight into multiprocessor memory behavior.

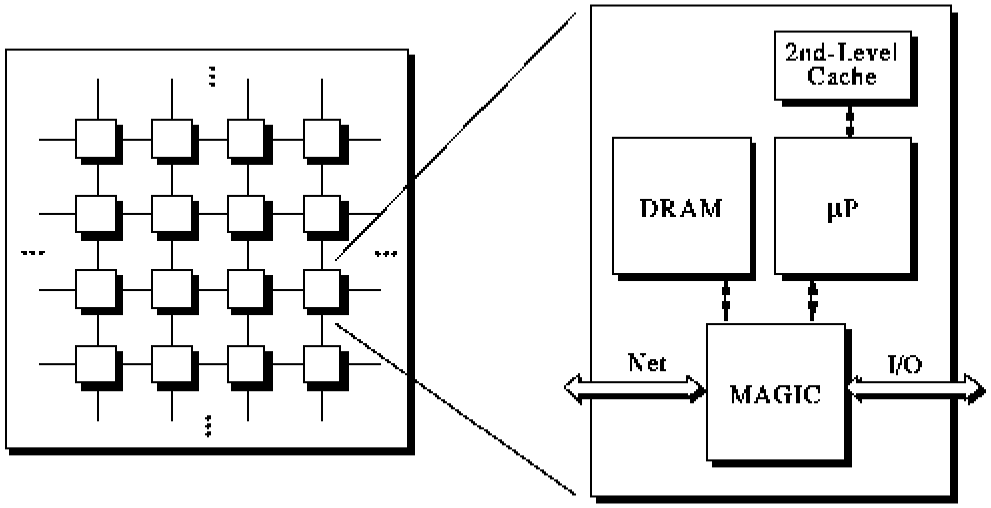

**Figure 1 FLASH block diagram**

The FLASH multiprocessor is an ideal machine to use for a study of alternative coherence protocols. The memory behavior of this machine is set by its memory controller, MAGIC, which includes an internal programmable processor. It is the code that runs on the memory controller processor that sets the operations used to provide cache coherence to the R10K main processors. Thus we can extend the types of memory consistency the machine supports by adding additional software for the embedded memory controller processor. Furthermore, the machine was built to give good visibility into the communications that result from the coherence operations. This allows us to analyze slow memory operations and specific operations that are causing problems.

The FLASH memory system presents a unified view of memory to every processor through a cache-coherence protocol. This protocol defines the set of operations necessary to satisfy memory requests for a given consistency model under various sharing conditions. The cache coherence present in modern multiprocessor systems can be significantly more complex because the memory system must handle additional race conditions that occur with additional requestors.

At times, cache coherence comes at a large cost. For example, if data is globally shared across the machine it must be invalidated if any write can occur. Getting ownership of a line to allow this write operation is called an upgrade, and Figure 2 shows the rise in remote upgrade latency as the number of remote sharers increase across three cache coherence protocols implemented on FLASH. The effect is dramatic. If data is shared across every node on FLASH, the communication latency could be 50 times larger than a normal memory access. Applications that frequently write globally shared data will not run efficiently on FLASH-like architectures. Other examples include occupancy overhead required for maintaining the sharing list and cache-to-cache transfer misses.

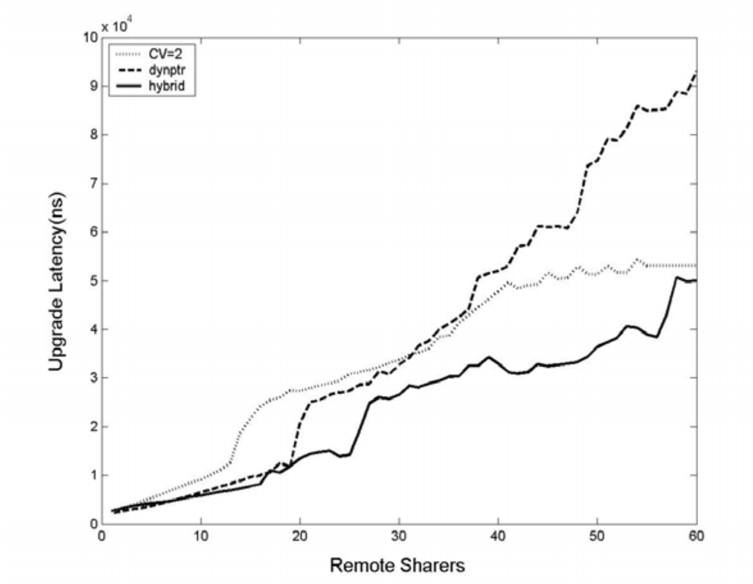

**Figure 2. Upgrade versus Total Remote Sharer.**

## 3 Programming Model

Given our goal of being able to designate accesses to certain data objects as coherent, non-coherent or write-update for specific phases of a computation, we now consider how this would be supported by the programming model. Conceptually, we want the programmer to be able to extend the type of a data structure to designate how accesses to it should be treated by the underlying hardware. Because we may want accesses to a particular object to be treated as coherent in some phases and non-coherent or write-update in others, we also want the programmer to be able to designate phases in which each coherence model can be selected for specific variables.

Similar capability is supported by Unified Parallel C, a global address space language that is becoming more available on high-end commercial parallel platforms and clusters, and is the subject of significant ongoing academic interest. UPC includes the type qualifier relaxed that permits designation of an object as having relaxed memory consistency. To express that a particular phase of a computation is to use relaxed memory consistency, the programmer can use the `#pragma upc` relaxed construct prior to that block of code[†].

Conceptually, the programming model support for the proposed selective coherence protocols would assume coherence by default. Type qualifiers such as relaxed, for the non-coherent protocol, or update, for the write-update protocol, would indicate that, for a given phase of the computation, a data structure should use an alternative coherence approach. Such type qualifiers would extend common parallel programming languages, such as dialects of C and Fortran.

As a pragmatic solution for the purposes of this experiment, we mimic this programming model support with a library-based strategy that extends OpenMP C programs. To create accesses to a same memory region with distinct coherence protocols in different phases of the computation, software calls a special malloc function that fills in a structure NCPtr_t with three pointers. The first element is a pointer to a coherent image of memory. The second element is a pointer to a non-coherent image of the same memory region, and the third is a pointer to a "write-update" image of the memory region. These three pointers are virtual address synonyms for the same physical memory location.

Figure 3 shows how to allocate and use memory under these three alternative coherence models. In this example, c, nc and wu point to the same physical memory location.

```
    // selective-coherence pointer type
    NCPtr_t p;
    malloc_non_coh(size, &p);
    // set the coherent, non-coherent and write-update pointers
    c =  p.cPtr;
    nc = p.ncPtr;
    wu = p.wuPtr;
```

**Figure 3. Malloc-based programming model for selective coherence.**

[†] In some sense, these constructs relate to the directives used as input to vectorizing compilers of 20 years ago to ignore vector dependences.

## 4 Protocols

Under a cache-coherence protocol, the memory system presents a unified view of memory to every processor. A cache-coherence protocol defines the set of operations necessary to satisfy memory requests for a given consistency model under various sharing conditions. In this project, in addition to an existing baseline coherence protocol, we developed and implemented two alternative coherence protocols (non coherent and write update) that are triggered by the special malloc libraries described Section 3. From an implementation perspective, accesses to non-coherent or write-update memory are distinguished from accesses to coherent memory by special address bits that tell FLASH's memory controller, MAGIC, to treat non-coherent and write-update accesses using a different set of protocols.

This section describes the baseline, non-coherent and write-update protocols. All protocols are implemented using a directory-based system. The directory-based system uses a reserved portion of main memory to store coherence state for each cache line. The initial read of the directory header represents one potential latency cost of the directory-based cache coherence on FLASH, but this cost remains constant when we modify the coherence protocol to implement relaxed coherence because the size of directory accesses does not change.

The cache-coherence protocol handles all bookkeeping operations to maintain sharing lists and directory state *after* replying to the original requester or forwarding the request to a remote processor. This approach minimizes the impact of directory bookkeeping on point-to-point latency of cache coherence. However, the memory controller remains occupied or busy while managing the entire handler and this occupancy can significantly impact latency through memory contention.

**Baseline coherent protocol**

The baseline coherence protocol uses two separate state machines to handle coherence transactions. These states are held in the directory in a reserved portion of main memory. The memory controller reads both states out of the 64-bit header field and stores updated header information back into the directory at the end of each handler if the directory state has changed.

The first state machine determines the sharing status of the cache line. Valid states include clean, exclusively in another cache, held in the I/O subsystem, or pending another transaction. In every state except clean, the transactions required by the protocol to satisfy the request are relatively simple because the protocol needs to communicate only with the original requester, the local directory, and potentially a remote dirty copy of the cache line.

If the cache line is read-only, the protocol reads a second state machine to track the list of read-only copies, or *sharing list*. In programs written well for shared-memory, most cache lines are shared only between a few processors. The coherence protocol, however, must still handle cases where cache lines are globally shared in almost every cache. Our 64-bit header does not include enough bits to maintain one bit for each processor, so the two-state protocol maintains the sharing list in a separate format depending on the size of the sharing list and the distribution of the sharers in the system.

The simplest sharing list formats cover the common cases of *non-shared* and *one-sharer*. These cases are represented as separate states to optimize common checks in the coherence protocol. For example, the memory controller can send an exclusive copy of a cache line immediately if the data is clean and non-shared. Similarly, the memory controller can immediately acknowledge an upgrade if the requester is the only sharer on the sharing lists. Both cases are checked quickly in our protocol.

The *local-bitvector* sharing list state uses 16 bits of the header, 1 bit for each node, to track “topologically close” sharers. This set of sharers is defined as the neighboring caches with the shortest point-to-point latency across the interconnection network. In FLASH, these neighbors are nodes on the local 16-processor meta-cube, but this approach could easily be extended to more complicated network topologies.

If the sharing list is distributed among caches that are not all within the local meta-cube, the protocol provides a *limited pointer* sharing-list state that uses 30 bits in the header to track up to five remote sharers. Each sharer requires 6 bits because the maximum node number is 63.

If the sharing list does not fall among 16 local neighbors or among 5 more dispersed neighbors, the sharing list is represented as a full 64-bit *bitvector*. Recall that the header includes only 64-bits in total and some bits are needed to represent both the coherence state and the sharing-list state. Therefore, the protocol allocates reserved protocol memory to store the bitvector and uses the header bits as a pointer to the full bitvector. Updates to the sharing list now require a pointer indirection, but this sharing-list state occurs infrequently in most shared-memory programs.

**Non-Coherent Protocol**

The non-coherent protocol adds a “non-coherent'” state to the coherence state machine and does not modify the sharing-list state machine. If a coherent memory location is accessed as non-coherent, the memory will transform the whole cache line to non-coherent state. This transformation can be expensive because it requires invalidations and interventions to all processors that hold clean or dirty copies.

When the memory controller receives a request for a non-coherent copy, it invalidates all outstanding coherent copies and then sends the cache line to the requester. Once all flush acknowledgements arrive at the requester, MAGIC sends an acknowledgment to the initial requester and sets the cache line to non-coherent. Subsequent non-coherent requests to the same cache line will be satisfied by the home node with no coherence actions to any cached copies. Additional non-coherent requests update the sharing-list state to track potential outstanding non-coherent copies in the system.

The protocol also uses the sharing list to recover coherence if the memory controller receives a coherent request for the cache line by invalidating the cache line from all potential cache lines that hold a non-coherent copy. Thus the sharing list reduces the number of invalidations because the memory controller does not send invalidations to processors that did not request a non-coherent copy.

**Write-Update Protocol**
Write-update accesses bypass normal cache coherence operations and write data directly to memory. A write-update access is similar to an uncached access because the request bypasses the secondary cache entirely. The memory controller sends invalidation messages to shared copies or flush operations to exclusive copies and immediately writes the update data to memory. This write operation is not sequentially consistent because the memory update is not held until all read copies are removed from remote caches. However, this violation only occurs when recovering cache coherence from a non-coherent state, which by definition violates sequential consistency.

The processor issues write updates to the memory controller whenever software writes to the update pointer from the previous section. The current interface does not support reads to the update pointer memory. However, the memory is still read-able through the coherent and non-coherent pointers.

## 5 Performance Studies

In this section we present an evaluation of the selective coherence protocols using RandomAccess, a benchmark from the DARPA HPCS benchmark suite. We also present experimental data for a sparse solver from LS-DYNA. Finally we present an experimental evaluation of the overhead of selective coherence protocols for applications running on a traditional coherent protocol.

### 5.1 Experiments using the Random Access benchmark

Random Access is part of the DARPA HPCS benchmark suite, and it is designed to stress the memory system by performing updates to randomly selected entries of a large table. The application can be parallelized such that each processor performs a chunk of the updates (a small number of errors due to race conditions between updates is acceptable). We chose Random Access for our experiment with relaxed coherence protocols versus non-coherent memory protocols because of its characteristics: a large, shared data structure that does not fit in the memory of a single node; intrinsic parallelism; potentially high memory coherence traffic and false sharing due to random updates to a same table entry or cache line. In addition, even though RandomAccess is a synthetic benchmark, its data access patterns are found in many scientific applications and benchmarks, as for example in LS-DYNA and NAS CG.

We concentrate on the parallel implementation of RandomAccess, where each process performs a fraction of the updates and a small number of errors due to race conditions between updates is acceptable. For our experiments in FLASH, we developed three OpenMP versions of RandomAccess, where the shared table is partitioned so that each node keeps a fraction of the table in its local memory, and each thread performs a set of random updates to the table. The sequential code and these three parallel versions are shown in Figure 4 and Figure 5.

```
RandomAccess
{
      uint64 *Table;
      malloc (tableSz, Table);
      // initialize table
      for (i = 0; i < tableSz; i ++)
            Table[i] = i;
      // perform updates
      ran = Ran(seed);
      for (i = 0; i < nUpdates; i ++) {
            ran = (ran << 1) ^ (ran < 0 ? POLY: 0);
            Table[ran & (tableSz-1)] ^= ran;
      }
}
```

(a) Sequential RandomAccess

```
CohRandomAcess
{
      uint64 *Table;
      malloc (tableSz, Table);
      // initialize table
      for (i = 0; i < tableSz; i ++)
            Table[i] = i;
      // parallel threads perform updates in coherent memory
      omp_set_num_threads(nThreads);
      tUpdates = nUpdates/nThreads;
#pragma omp parallel private (threadId, ran)
{
      threadId = omp_get_thread_num();
      ran = Ran(threadId);
      for (i = 0; i < tUpdates; i ++) {
            ran = (ran << 1) ^ (ran < 0 ? POLY: 0);
            Table[ran & (tableSz-1)] ^= ran;
      }
} // end parallel section
}
```

(b) Coherent RandomAccess

**Figure 4. Sequential and baseline RandomAccess code.**

```
NonCohRandomAccess
{
      NCPtr_t Table;
      malloc_non_coh (tableSz, Table);
      cohTable = Table.cPtr;
      nonCohTable = Table.ncPtr;
      // initialize table in coherent memory
      for (i = 0; i < tableSz; i ++)
            cohTable[i] = i;
      // perform updates in non-coherent memory
      omp_set_num_threads(nThreads);
      tUpdates = nUpdates/nThreads;
#pragma omp parallel private (threadId, ran)
{
      threadId = omp_get_thread_num();
      ran = Ran(threadId);
      for (i = 0; i < tUpdates; i ++) {
            ran = (ran << 1) ^ (ran < 0 ? POLY: 0);
            nonCohTable[ran & (tableSz-1]] ^= ran;
      }
} // end parallel section
}
```

(a) Non-Coherent RandomAccess

```
UpdateRandomAcess
{
      NCPtr_t Table;
      malloc_non_coh (tableSz, Table);
      cohTable = Table.cPtr;
      updateTable = Table.wuPtr;
      // initialize table using write-update accesses
      for (i = 0; i < tableSz; i ++)
            updateTable[i] = i;
      // perform updates to table using write update accesses
      omp_set_num_threads(nThreads);
      tUpdates = nUpdates/nThreads;
#pragma omp parallel private (threadId, ran)
{
      threadId = omp_get_thread_num();
      ran = Ran(threadId);
      for (i = 0; i < tUpdates; i ++) {
            ran = (ran << 1) ^ (ran < 0 ? POLY: 0);
            tmp = cohTable[ran & (tableSz-1)];
            updateTable[ran & (tableSz-1]] = tmp ^ ran;
      }
} // end parallel section
}
```

(b) Write-update RandomAccess

**Figure 5. RandomAccess code for non-coherent and write-update protocols.**

`RandomAccess` in Figure 4(a) is the sequential version of the benchmark. The first parallel version, `CohRandomAccess` shown in Figure 4(b), is designed for a cache-coherent shared memory model. It relies on the cache coherence protocol to ensure that updates to a given entry are performed to a coherent memory location. The order in which updates are performed is not relevant for the correctness of the computation because the operation performed by the updates is associative (an exclusive-or operation). In `NonCohRandomAccess`, shown in Figure 5(a), the parallel threads perform updates in a non-coherent way, since the benchmark tolerates a small amount of errors, and this should improve performance associated with coherence traffic due to false sharing. The third parallel version, `UpdateRandomAccess`, uses the write-update protocol described in Section 4. In `UpdateRandomAccess` the table is initialized using write-update, which avoids caching un-initialized data. During the main computation the read operations use the coherent memory pointer, and the writes use the write-update pointer, as shown in Figure 5(b).

Figure 6(a), (b) and (c) show the speedups of RandomAccess using the baseline coherent protocol, the non-coherent and write-update protocols. We present results for table sizes of 8, 16 and 32 MBytes. For each table size the graph shows speedups for 2, 4, 8 and 16 threads with respect to the single-thread baseline coherent version. The number of updates to the global table is configured to be four times the table size, which is the default number of updates recommended by the HPC Challenge benchmarks. When using multiple threads, each thread runs on a distinct FLASH node and a fraction of the table is allocated on each node's local memory.

The coherent version of RandomAccess suffers a slowdown for 2 and 4 threads for table sizes of 8 and 16M Bytes, before achieving modest speedups for 8 and 16 threads. As the table size grows to 32Mbytes, the slowdowns are eliminated, as the likelihood of local accesses increases. Larger table sizes beyond 32MBytes were not included in the experiment due to limitations in this experimental implementation in how the operating system maps the memory address space, but it is clear that the trend is similar across table sizes. With the 32-MByte table the coherent version achieves speedups of up to 4.1 for 16 threads -- at best a modest improvement from parallelizing the computation.

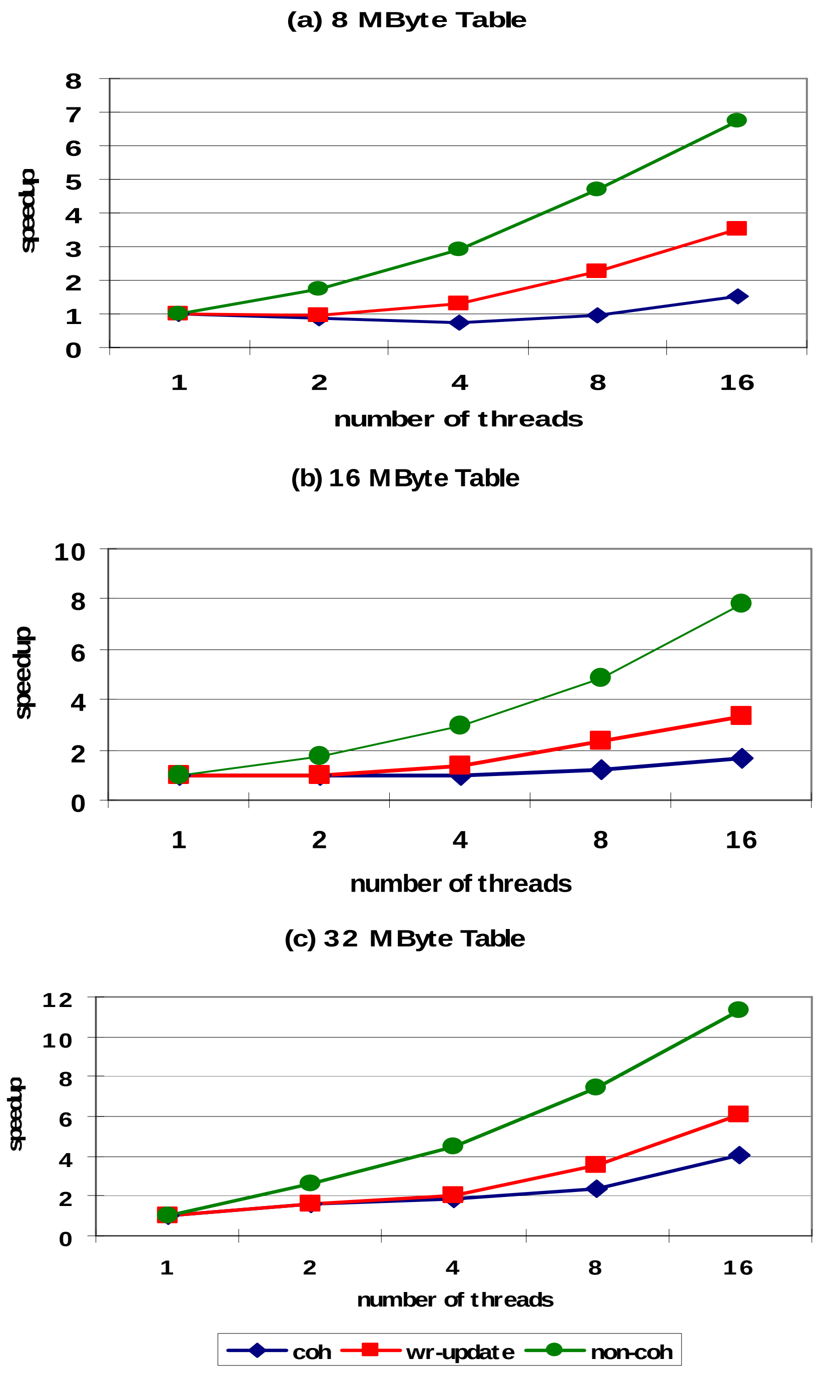

**Figure 6. Speedups of Non-Coherent and Write-Update RandomAccess with respect to Coherent RandomAccess baseline.**

RandomAccess with non-coherent accesses to the global table achieves much better speedups, up to 11.35 for 16 threads with the 32 MByte table. Overall, performance is always improved, and is as much as 5 times better than the baseline coherence protocol for 16 threads. This result demonstrates the value of the non-coherent protocol for improving performance by eliminating coherence traffic on codes with unpredictable data access patterns. The reason why the speedup is 11.35 for 16 threads, rather than something closer to 16, is as follows. Although coherence is not being enforced, a large fraction of the memory accesses of each thread are not local and suffer large remote memory latencies. In addition, the costs of transitioning from coherent to non-coherent memory (the table is initialized as coherent memory) also limit the speedup.

RandomAccess with write-update to the global table also performs consistently better than its coherent version, up to 2.5 times better, but not as well as the non-coherent version. Write-update accesses result in more coherence traffic than non-coherent accesses because any coherent copies must be invalidated/flushed from cache(s) on each write. Subsequently, this data may be read through the coherent pointer, since write-update reads are not supported.

As an extension to the write-update protocol requiring additional hardware support in the memory controller, in place of the write-update protocol we could use a “fire-and-forget” protocol that performs the operation on the data within the memory controller of the home node, eliminating the need for the read.

## 5.2 Experiments using the Sparse Solver from LS-DYNA

LS-DYNA [15] is a commercial derivative of DYNA, originally developed at Lawrence Livermore National Laboratory. The LS-DYNA solver is a general-purpose, nonlinear finite element program capable of solving a vast array of engineering and design problems ranging from bioprosthetic heart valve operation to automotive crash and earthquake engineering.

The dataset used for LS-DYNA was taken from an automotive metal forming problem, the stamping of sheet metal into a hood as shown in Figure 7. Simulation of spring back after stamping is handled implicitly, and a symmetric-indefinite direct sparse solver is generally required. The computational bottleneck of this transient, non-linear problem is the triangular solver, of which the forward solver is the first half.

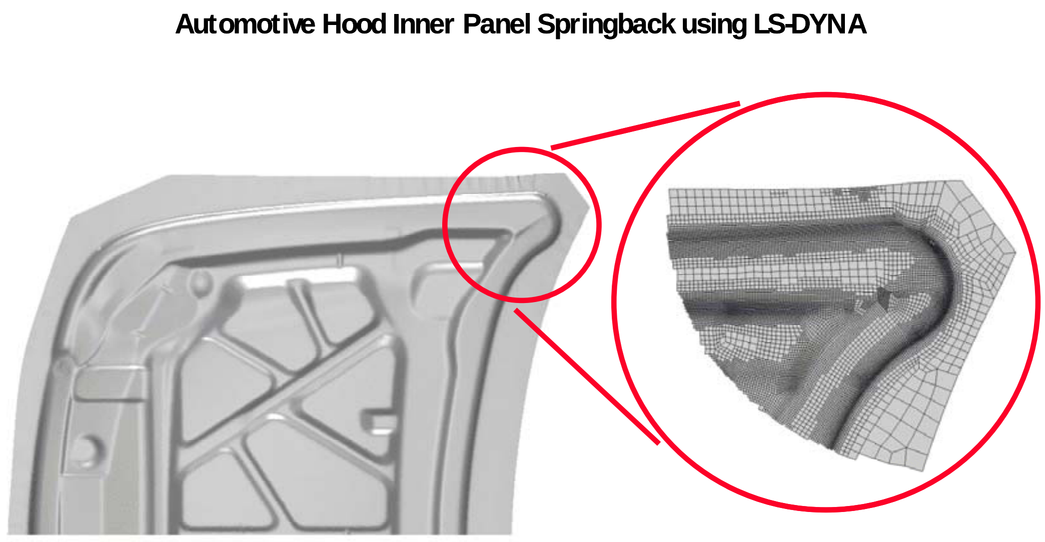

**Figure 7. Hood dataset**

Figure 8 illustrates the core computation of the forward solver from LS-DYNA, in which accesses to array `x` exhibit a very similar behavior as accesses to the main data structure in RandomAccess. The better-known NAS NPB Conjugate Gradient benchmark [18] has a similar construct wherein a large matrix is addressed with unit stride and is used to update randomly chosen entries in a smaller, right-hand-side vector.

```
   // Forward solve
   p = 1
   do 10 j = 1, nrhs
     do 20 k = 1, size
       p = p + 1
       reg11 = matrix(p-1) * x(indices(k),j)
       do 30 i = k+1, ld
         x(indices(i),j) = x(indices(i),j) – matrix(p) * reg11
         p = p + 1
30 continue
20 continue
10 continue
```

**Figure 8. LS-DYNA forward solver.**

We extracted just the code and associated data for the forward solver in LS-DYNA using a tool called a code isolator, which was developed for the purpose of facilitating performance experiments on large applications [13]. We investigated the performance of the forward solver step of LS-DYNA shown in Figure 10 when allocating array x as non-coherent memory. In the OpenMP parallel version, the code from **Error! Reference source not found.** is nested within an additional loop that is parallelized, with a sequential outermost loop. We omit the details in the figure. While accesses to arrays matrix and indices are regular and have spatial locality in the innermost loop, accesses to x are irregular. Furthermore, in the parallel version of the loop nest, arrays matrix and indices are partitioned so that accesses are local to each thread (FLASH node). Hence the irregular accesses to x are responsible for most remote memory accesses, false sharing and unnecessary coherence traffic.

Figure 9 shows the speedups of the forward solver step, comparing the baseline coherent and non-coherent protocol on the irregular array x. The graph presents speedups for 1, 2, 4, 8, and 16 threads with respect to the single-threaded execution under the baseline coherent protocol. The non-coherent version speedups are 18% better on average than those of the coherent version. In both versions the speedups are limited by load imbalance and OpenMP parallelization overhead. Still, the improvements observed when using non-coherent accesses to array x indicate that this loop nest benefits from selective coherence protocols.

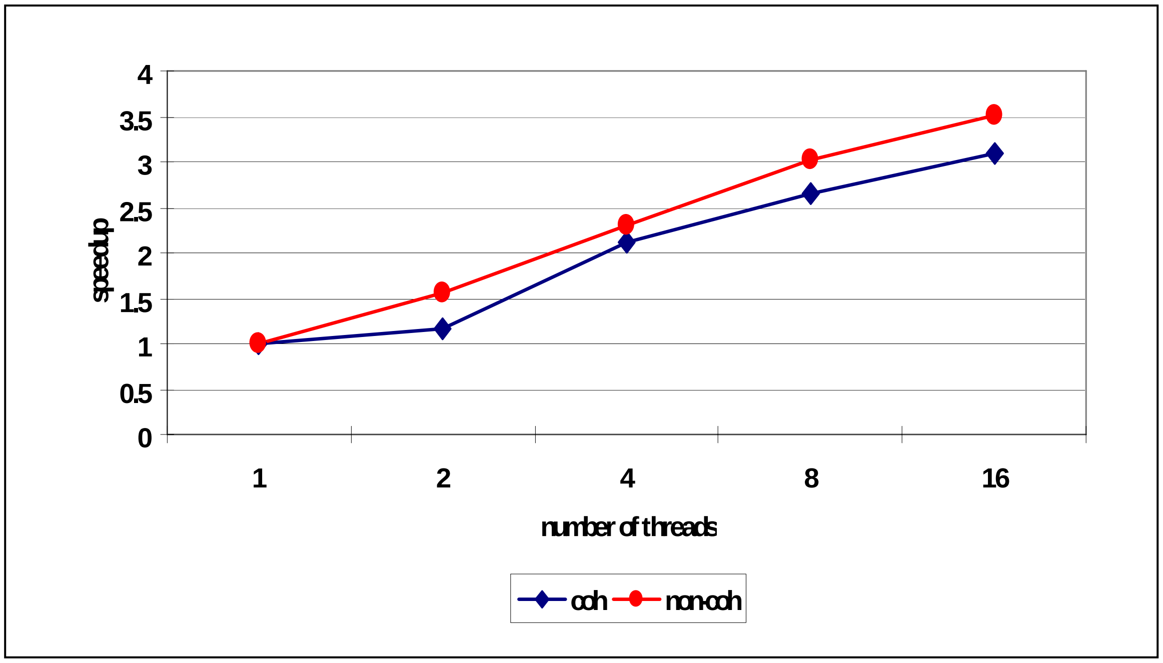

**Figure 9. Speedups of forward solver step with respect to sequential execution.**

We envision that write-update accesses to x would prevent caching of data that has no inherent locality. Therefore, we expect the benefits of the write-update protocol would be the elimination of false sharing and unnecessary coherence traffic. Unfortunately the FLASH machine is no longer available for our experiments (it was decommissioned in February 2006), but given the performance of RandomAccess with write-update and the similar access patterns exhibited by these two kernels, we expect that the forward solver could also benefit from the write-update protocol.

## 5.3 SpecOMP Performance Measurements

Relaxing the shared-memory model augments--but does not supplant--the cache-coherence model. We took care to implement our non-coherent protocol extensions to minimize the impact on shared-memory transactions. Nevertheless, adding support for non-coherent memory requires the memory controller to check the coherency status of the cache-line before processing coherent memory requests. This overhead should be small because the coherence protocol must check the directory header anyway to check if the cache-lines are dirty or pending.

We expect that the overheads to be even smaller in a hard-wired memory controller. In practice, overheads reflect limitations of the MAGIC protocol processor's limited 16-KB direct-mapped instruction cache. Large coherence protocols introduce larger occupancy penalties because the overall memory footprint of the coherence protocol is larger to support non-coherent handlers.

We measure the impact of this architectural change by comparing our non-coherent protocols with a base bit-vector coherence protocol [2] that does not support non-coherence protocol extensions. Each bit in the bitvector represents 2 processors and our implementation includes a local bit to differentiate between local and remote sharing. This base protocol experiences shorter handler occupancies because it does not need to check if the cache-line is non-coherent before handling memory requests. To compare, we execute a subset of the SpecOMP2001 benchmarks using both protocols on a 64-processor FLASH machine. We use only 56 of the available processors to limit the potential impact of parasitic operating system effects. Table 1 illustrates the overall execution time, maximum per-thread occupancies, and average per-thread occupancies for both protocols. `Base` refers to a standard coherence protocol, while `Sel` refers to the non-coherence protocol implementation of Section 4.

Our benchmark selection focuses attention on applications with significant local memory traffic because most applications written well for shared-memory experience a degree of memory locality. The `SWIM` benchmark generates the highest frequency of cache misses. 95% of these requests access local memory. However, any remote request introduces contention because local and remote requests compete for access to the memory controller. The `WUPWISE` and `MGRID` benchmarks have lower rates of incoming memory requests and a higher percentage of remote requests. About 75% of `FMA3D`'s cache misses access local memory and remote memory requests access 2 remote nodes. We include `APSI` as an example of an application with poor memory locality. Only 3% of its cache misses access local memory and remote requests are randomly distributed around the machine. More information on these benchmarks can be found in [11].

The non-coherent protocol introduces an average slowdown of 3% and increases the maximum occupancy by 2%. In high-occupancy benchmarks like `SWIM`, the overheads are more pronounced (4.5% slowdown with a 10% increase in maximum occupancy). Larger occupancies introduce contention at communication hot-spots. This contention eventually impacts latency in the form of queuing delay.

Overall, the impact of non-coherent support on coherent memory transactions is small. Although there is no need to use the non-coherent protocol for the SpecOMP benchmarks, these results serve to support the point that the alternative protocol implementations will not adversely impact applications that use mixed coherent and non-coherent protocols.

| | Time(s) | | Max Occup (%) | | Ave Occup (%) | |
|---|---|---|---|---|---|---|
| | Base | Sel | Base | Sel | Base | Sel |
| SWIM | 439 | 459 | 64 | 71 | 56 | 64 |
| WUPWISE | 521 | 559 | 40 | 37 | 35 | 30 |
| MGRID | 836 | 831 | 71 | 63 | 62 | 51 |
| FMA3D | 791 | 816 | 36 | 42 | 26 | 22 |
| APSI | 579 | 586 | 46 | 48 | 25 | 18 |
| Average | 614 | 633 | 50 | 51 | 38 | 33 |

**Table 1 SpecOMP2001 overheads of non-coherent versus a base bit-vector coherence protocol.**

## 6 Compiling for Transactional Memory

The study of transactional memory behavior was performed on the Stanford Transactional Coherence and Consistency (TCC) simulation environment, Tassel. Tassel is an execution-driven simulator that models the TCC architecture and runs on workstations with PowerPC G5 processors. Tassel supports fast forwarding (by direct execution on the G5) to speed up the simulation of the initialization code of benchmarks.

USC/ISI provided expertise in applications and compiler technology in the SUIF compiler. SUIF targets shared-memory multiprocessor architectures, including the SGI Origin upon which FLASH is based. USC/ISI's compiler research program revolves around the SUIF infrastructure. This collaboration with Stanford has resulted in an initial prototype compiler for TCC, based on SUIF, which translates from a high-level programming model to a lower level representation suitable for interaction with simulation and hardware. Initial optimizations include reduction and privatization for exposing parallelism. In addition the compiler applies loop transformations such as strip-mining for generating efficient code for parallelized loops with the desired transaction granularity and iteration/data distribution (cyclic, block or block-cyclic).

### 6.1 Transactional Coherence and Consistency (TCC)

Transactional Memory (TM) is a recent paradigm shift that makes shared-memory multiprocessors approachable to the average programmer. Using transactions as the central abstraction for designing and programming parallel systems leads to a shared memory programming and memory consistency model called Transaction Coherence and Consistency (TCC). Transactions simplify parallel programming by providing a way of writing correct and efficient shared-memory programs without locks, semaphores, or condition variables. Stanford has produced significant research results on this topic [6][7][8][9], and an ongoing collaboration between Stanford and USC/ISI has produced a prototype optimizing compiler for high-level TCC constructs.

The compiler developed by USC/ISI supports optimizations for exposing parallelism and decreasing the number of violations in reduction computations. Data that is local to a transaction can be privatized, and therefore not be part of the state of the transaction that is buffered and communicated across transactions (for example, large temporary arrays used to store temporary data values). Further, operations that reduce values to a loop carried `sum` variable are frequently used within loops that are otherwise good targets for transactional parallelization. These operations can cause frequent violations on the `sum` variable. However, when such operations are associative, such as addition, multiplication, and minimum/maximum, and can therefore be reordered to maximize parallelism, we can privatize the `sum` variable within each processor and only combine these variables to a sequential sum after the end of the loop.

In addition, the compiler uses loop transformations such as strip-mining and tiling for generating efficient code for parallelized loops with the desired transaction granularity and iteration distribution (cyclic, block or block-cyclic). Parallelization at different levels of loop nests will have different performance properties that need to be evaluated. Outer loops provide large granularity, but can sometimes get too large, because realistic TCC systems have finite amounts of per-transaction buffering. Inner loops can often be too small to effectively use without combining iterations, due to transaction startup/commit overheads. Either may have loop-carried dependencies that prevent it from being a good target. In many cases, standard loop transformations can improve performance such as tiling (to adjust transaction "chunk" size), and loop fission and fusion. Tuning of parallelization granularity involves navigating a complex tradeoff between managing run-time overheads, avoiding buffer overflows, avoiding violations, and managing the relationship between parallelism and data locality.

Future work on the TCC compiler would include *data specifications*, where high-level data specifications or properties proven by the compiler can be used to both reduce buffering and/or communication among processors, and support for *speculative parallelism*, where the compiler speculates on the safety of parallelizing a particular construct.

## 7 Conclusion

In this project we addressed the problem of unintended communication, that is, coherence traffic between processors when hardware cache coherence is supported. Our approach for managing unintended communication is based on relaxing data coherence using new cache-coherence protocols. To investigate the potential benefits of relaxing data coherence, we conducted a series of experiments based on an actual implementation of a pair of selective coherence protocols for a CC-NUMA shared-memory architecture, the Stanford FLASH hardware testbed.

Our experiments demonstrate the advantages of extending the basic shared-memory model to improve performance of applications that do not require traditional cache coherence for all the data, without harming the performance of more common shared-memory applications. For RandomAccess, the non-coherent and write-update versions outperform the coherent version up to 5 and 2.5 times, respectively. In LS-DYNA, we obtain improvements of up to 35% and on average 18% using the non-coherent version. In SpecOMP benchmarks, the protocols have a cost of less than 3% in applications where the alternative mechanisms are not needed.

The methods described in this report point the way to greater improvements in programming shared memory machines by making focused but orthogonal changes to the memory system and cache hierarchy. Traditional approaches focus on system-wide changes to the memory model, whereas we target a specific bottleneck. Under our approach the user is not required to use our techniques to write correct programs (as in message passing or simple NUMA) which improves the portability of applications from other cc-NUMA or uniprocessor architectures and eliminates the need to translate the entire program to our new non-coherent memory models.

Future architectures that provide orthogonal extensions of the shared memory model coupled with software tools to identify opportunities to take advantage of these extensions will further improve the attractiveness of the shared-memory model at large scale.

## References

[1] Agarwal A, Bianchini R, Chaiken D, Johnson K, Kranz D, Kubiatowicz J, Lim B-H, Mackenzie K, Yeung D. et al. *The MIT Alewife Machine: Architecture and Performance*. Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 2--13, 1995.

[2] Censier L, Feautrier P. *A New Solution to Coherence Problems in Multicache Systems*. IEEE Transactions on Computers C-27, pages 1112--1118, Dec. 1978.

[3] Chafi H, Minh C, McDonald A, Carlstrom B, Chung J, Hammond L, Kozyrakis C, Olukotun K. *TAPE: A Transactional Application Profiling Environment*. Proceedings of the 19th Annual International Conference on Supercomputing, pages 199-208. June 2005.

[4] Chaudhuri M, Heinrich M, Holt C, Singh J-P, Rothberg E, Hennessy J. *Latency, Occupancy, and Bandwidth in DSM Multiprocessors: A Performance Evaluation*. IEEE Transactions on Computers, volume 52(7), pages 862--880, July 2003.

[5] Chaundhuri M, Heinrich M. *The Impact of Negative Acknowledgements in Shared Memory Scientific Applications*. IEEE Transactions on Parallel and Distributed Systems, volume 15, pages 134--150, February 2004.

[6] Chung J, Chafi H, Minh C, McDonald A, Carlstrom B, Kozyrakis C, Olukotun K. *The Common Case Transactional Behavior of Multithreaded Programs*. In Proceedings of the 12th Intl. Conference on High Performance Computer Architecture. Feb. 2006.

[7] Hammond L, Carlstrom B, Wong V, Hertzberg B, Chen M, Kozyrakis C, Olukotun K. *Programming with transactional coherence and consistency (tcc)*. Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 1-13, New York, NY, USA, October 2004. ACM Press.

[8] Hammond L, Wong V, Chen M, Carlstrom B, Davis J, Hertzberg B, Prabhu M, Wijaya H, Kozyrakis C, Olukotun K. *Transactional memory coherence and consistency*. Proceedings of the 31st International Symposium on Computer Architecture, pages 102Ð113, June 2004.

[9] Heinrich M, Kuskin J, Ofelt D, Heinlein J, Baxter D, Singh J-P, Simoni R, Gharachorloo K, Nakahira D, Horowitz M, Gupta A, Rosenblum M, Hennessy J *The Performance Impact of*

*Flexibility in the Stanford FLASH Multiprocessor*. Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 274--285, Oct. 1994.

[10] Gibson J. *Memory Profiling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 2002.

[11] Kunz R. *Performance Bottlenecks in Large-Scale Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 2005.

[12] Kustin J, Ofelt D, Heinrich M, Heinlein J, Simoni R, Gharachorloo K, Chapin J, Nakahira D, Baxter J, Horowitz M, Gupta A, Rosenblum M, Hennessy J. *The FLASH Multiprocessor*. Proceedings of the 21st International Symposium on Computer Architecture, pages 302--313, April 1994.

[13] Lee Y-J, Hall M. *A Code Isolator: Isolating Code Fragments from Large Programs*. Proceedings of the 17th Workshop on Languages and Compilers for Parallel Computing, 2004.

[14] Lenoski, D, Laudon J, Gharachorloo K, Weber W-D, Gupta A, Hennessy J, Horowitz M, Lam M. *The Stanford DASH Multiprocessor*. IEEE Computer, volume 25(3), pages 63--79, March 1992.

[15] *LS-DYNA User's Manual V. 960*. Livermore Software Technology Corporation, http://ww.lstc.com, March 2001.

[16] Martin M, Hill M, Wood D. *Token Coherence: Decoupling Performance and Correctness*. Proceedings of the 30th International Symposium on Computer Architecture, pages 182--193, June 2003.

[17] McDonald A, Chung J, Chafi H, Minh C, Carlstrom B, Hammond L, Kozyrakis C, Olukotun K. *Characterization of TCC on Chip-Multiprocessors*. Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05), pages 63-74, Washington, DC, USA, September 2005. IEEE Computer Society.

[18] *NAS Parallel Benchmarks*. http://www.nas.nasa.gov/Software/NPB.

[19] Reinhardt S, Larus J, Wood D. *Tempest and Typhoon: User-Level Shared Memory*. Proceedings of the 21st Annual International Symposium on Computer Architecture, pages 325--337, April 1994.

[20] Saulsbury A, Wilkinson T, Carter J, Landin A. *An Argument for Simple COMA*. Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture, pages 276--285, January 1995.

[21] Soundararajan V, Heinrich M, Verguese B, Gharachorloo K, Gupta A, Hennessy J. *Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors*. Proceedings of the 25th International Symposium on Computer Architecture, pages 342--355, June-July 1998.