Measuring and Characterizing Cache Coherence Traffic

Tuan Bui, Brian Greskamp, I-Ju Liao, Mike Tucknott
CS433, Spring 2004

Abstract

Cache coherence protocols for shared-memory multiprocessors have been well-studied. Here we present a trace-driven cache simulator used to derive network traffic statistics for three simple coherence protocols and three directory formats. Results for the SPLASH benchmarks are compared with published results and found to be in agreement. A short discussion of the strengths and weaknesses of each coherence scheme ensues.

1 Introduction

Cache coherence protocols and directory schemes were a popular topic in the late 1980s. Little new material on hardware cache coherence has appeared since then, and interest has shifted from large-scale shared memory machines to message-passing clusters. However, shared-memory multiprocessors are being reborn in the form of the chip multiprocessor. As CMPs grow to include more cores, the cache coherence problem may again draw research interest. In the meantime, we have undertaken an empirical performance study in order to better understand the strengths and weaknesses of both invalidation and update protocols in snoopy and directory-based systems. Good surveys on cache coherence protocols already exist [4, 1, 2], so this paper does not aspire to novelty. Instead, it attempts to validate and reinforce the earlier work with independent simulation results.

This paper considers both snoopy (*ie.* bus-based) and directory-based systems. It covers both sparse and full-directory schemes and both write-update and write-invalidate protocols. The SPLASH/SPLASH-2 [9, 7] scientific benchmarks are used to evaluate the amount of network traffic for each system. The results are compared with those of previous work, yielding conclusions about the efficacy of each protocol for various applications.

The remainder of this paper is organized as follows. Specifications for the protocols and directory schemes appear in section 2. The tools used in the study, including a trace-driven cache simulator, a multiprocessor simulator, and a benchmark suite feature in section 3. Section 4 details the cache system simulator. Simulation results appear in section 5. Section 6 discusses the strengths and weaknesses of each configuration, and finally, section 7 concludes.

2 Protocols

This section specifies the cache coherence protocols and directory formats used throughout the paper. Two invalidation-based protocols are considered: the simple three-state invalidation or 'MSI' protocol and the Illinois [6] or 'MESI' protocol. DEC Firefly [8], an update protocol designed for small-scale snoopy systems, is also considered. All three of these protocols can operate on a shared bus in snoopy fashion or on a non-broadcast interconnect with the help of a directory. In snoopy mode, any PE has the ability to send a broadcast to all other caches in a single bus cycle. On a non-broadcast interconnect, a directory and associated controller are needed to maintain the shared state of each cache line and notify caches when a line's state changes. The following

paragraphs briefly describe the operation of each protocol in both the snoopy and directory-based cases.

Note that some details such as `ACK` and `NAK` messages are missing from our protocol models. In a real system, every coherence transaction has a request and a response phase. For simplicity, we omit the response phase. This should not be a problem since we do this consistently for all protocols and since the number of messages comprising the response phase is equal to the number sent in the request phase, so it does not affect the *proportion* of the messages.

Three State Invalidation A cache line may be in one of three states: `dirty`, `shared`, or `invalid`. Before making a line `dirty` by writing to it, a cache must first broadcast an invalidation for that line to all other caches in the system. In a snoopy system, this is accomplished by sending a single `snoopy_invalidate` message onto the bus. In directory mode, the protocol must send a `make_dirty` message to the directory. The directory is then responsible for sending `invalidate` messages to any caches that might have the line. The snoopy system handles the dirty read-miss case easily. In the directory system, the directory must check to see if any processor has the line dirty before returning data from memory on a read request. The directory will send a `writeback_request` to the dirty cache. The dirty cache obeys the message by writing the line back and supplying it to the requesting processor.

Illinois Also known as "MESI", Illinois builds on three-state invalidation by adding an `exclusive` state. A line is `exclusive` when there is only one sharer and that sharer has not yet written to the line. No coherence traffic is required to transition from `exclusive` to `dirty`. It works well on snoopy systems which have hardware support to allow a cache to determine if it is the only sharer. In a directory-based system, the directory must regard `exclusive` lines as `dirty` since the cache may write the line without notifying the directory. This *may* cause extraneous traffic (in the form of `writeback_requests` for lines which are not actually dirty.

Firefly An update protocol, Firefly was designed for small bus-based systems. Its states are: `exclusive`, `shared`, `dirty`, and `invalid`. On a write hit to a `shared` line, the cache must broadcast an update to all other sharers. On a read miss, the cache tests a special line on the bus to determine whether it should have the line in `shared` or `exclusive` mode. As with the Illinois protocol, when operating in directory mode, an `exclusive` line may transition to `dirty` without notifying the directory.

2.1 Directory Formats

In directory-based systems, a single protocol transaction such as an invalidation may require many messages to be sent. The number of messages that will be sent depends on the number of sharers and the granularity with which state data is kept in the directory. We consider three directory formats: Dir$N$, Dir$I$, and DirI_{CV}. Dir$N$ maintains a complete bit vector for each line of memory, recording the state of that line for every processor in the system. With the full bit vector, the directory knows exactly which caches need to receive invalidations and updates, so network traffic is at a minimum. On the other hand, the bit vector consumes a significant amount of memory for large systems. Dir$I$ saves memory by keeping a limited number of pointers i to sharers for each line. When the number of sharers exceeds i, a broadcast bit is set and the directory conservatively assumes that all caches have a copy of the line. A variant of Dir$I$ is DirI_{CV}. The directory enters "coarse vector" mode when the number of sharers exceeds i. In coarse vector mode, each bit in the directory entry is used to store the state for a group of processors. With n processors, the size of a group is $\lceil n/\lceil lg(n) \times i \rceil \rceil$.

3 Tools

A trace-based simulator was used to count protocol coherence messages. The simulator is discussed in detail in section 4. Traces for the simulator were generated with the RSIM [] multiprocessor simulator running SPLASH/SPLASH-2 [] benchmarks. The following sections discuss the simulation procedure in detail.

3.1 Memory Trace Generation

We use RSIM [5] to generate our memory traces. RSIM is a shared-memory multiprocessor simulator that models superscalar processors connected to a CC-NUMA memory system. We modified RSIM to generate a trace file which contains a line for each memory access a processor makes. The line includes the processor ID, the memory address, and whether the access was a read or a write. We use select programs from the SPLASH/SPLASH-2 application suite running on RSIM to generate our traces.

3.1.1 Simulator Overview

RSIM is an execution-driven simulator designed to study multiprocessor system build from superscalar PEs. Its processor model is more representative of current and near-future processors than other currently available multiprocessor simulators. Since RSIM models an out-of-order processor with parallel issue, memory access patterns correspond more closely to those of real-world advanced processors than if we had used a simulator with a more simplistic in-order or single-issue-width processor model.

3.1.2 Modifications

The only modification made to RSIM is a call to print each memory access (on the processor side) to our memory trace file (henceforth, `memfile`). `Memfile` sizes range from the high hundreds of megabytes to the low gigabytes for each application run.

3.2 Simulated Applications Overview

The Stanford ParalleL Applications for SHared memory (SPLASH) and The improved SPLASH (SPLASH-2) are suites of parallel programs written for cache-coherent shared memory machines. For this project, three programs were chosen from the SPLASH-2 suite to run on RSIM. The three programs from SPLASH-2 are FFT, LU, and Radix. The program chosen from SPLASH was MP3D. The following sections give a brief description of each of the programs used.

MP3D In this N-body simulation program, each processor computes the movement of a set of molecules. Since the partitioning of molecules is not related to its position is space, each processor must access the space array to complete its computation. This causes processor locality to be low. Contention for the same data occurs when two processors compute on molecules that occupy the same space cell.

FFT The FFT Kernel implements Bailey's six-step fast fourier transform. Bailey's six-step fft is described in [3] This in an improved FFT for hierarchical memory systems. Every processor is assigned a contiguous set of rows. Communication is mainly through 3 all to all matrix transpose steps. The transposes are blocked to exploit reuse of the cache lines.

LU The LU Kernel performs the LU matrix factorization. A dense square matrix A is divided into an array of $B \times B$ blocks and distributed among the processors. The elements within the blocks are allocated to improve spatial locality.

Radix In this radix sort algorithm, each processor generates a set of local histograms. These histograms are then combined into one large, globally shared histogram. The global histogram is then used to generate keys for the next iteration. Communication is all to all mainly through writes.

4 Simulated Cache System

A custom-written cache simulator processes the memory access traces from RSIM. The simulator is designed to be extensible, capable of implementing a wide range of directory-based and snoopy cache coherence protocols. For simplicity, the intricacies of the interconnect network are ignored, and reliable equal-cost message delivery is assumed between all nodes. The following sub-sections discuss the simulator abstractions and their implementations. The meanings of the various statistics reported by the simulator are also enumerated.

4.1 Implementation Overview

An important observation is that the directory format (Dir$N$, Dir$I$, DirI_{CV}) is logically separate from the coherence protocol. Therefore, the two are implemented as separate modules in the simulator. With this separation, p cache protocols and v directory formats can simulate a total of $p \times v$ distinct cache systems with only O($p+v$) implementation effort. Each coherence protocol and each directory format is implemented as a dynamically loadable library that exports a common interface. At runtime, the simulator customizes itself by loading the protocol and directory format specified on the command line. New protocols and directory formats can thus be added by simply compiling a new shared object (`.so`) file. Consequently, the use dynamic rather than static linking simplifies the task of adding new protocols to the distribution. On the other hand, it is detrimental to portability.

In order to serve a wide range of protocols, the interface to the directory schemes and protocol state machines was carefully considered. Among the compromises that were made in the name of simplicity are that the directory does not support dirty sharing; no line can simultaneously be dirty in two or more processors' caches. Also, instead of a distributed directory, a central directory equidistant from all processors is assumed.

4.2 Per-Processor Cache Implementation

The simulator contains an array of caches (one per processor) wrapped in the `cache_sys` module. All of the caches in the system have identical parameters (*ie.* size, associativity, line length, and replacement policy). The `cache_sys` API allows the caller to perform the following operations on a particular cache, addressed by processor number:

`touch(address)` If the given address is not in the cache, allocate a line for it and mark it valid, evicting a resident line if necessary. Move the line to the tail of the LRU replacement queue.

`set_state(address, state)` Set the state of the line containing the specified address. State can be any protocol-defined integer value. If `state=-1`, invalidate the line.

`get_state(address)` Return the state set by the most recent call to `set_state` or -1 if the line is not cached.

4.3 Directory Implementation

In addition to the per-processor states kept in the cache line, the simulator uses a centralized directory to store information about which processors have copies of a given line. The directory is implemented as a sparse data structure, containing lines only for data that the directory believes currently exists in some processor's local cache.

Each line in the directory has a 'share state' and a list of sharers[1]. The share state is the global state of the line. All caches that have the line are in the same share state. The list of sharers is to be interpreted as a conservative listing of processors that *might* have a copy of the line. Each directory format implements functions to extract the share state and sharer list from a directory entry as follows:

[1] not necessarily implemented as a list, depending on the directory format

`get_share_state(entry)` Return the share state for the given entry.

`set_share_state(entry, state)` Set the share state for the specified entry.

`clear_sharers(entry)` Clear the sharers list for the given entry.

`get_is_sharing(entry, proc)` Return true if the specified processor is sharing the line associated with this entry.

`set_is_sharing(entry, proc, is_sharing)` If `is_sharing` is true, add the specified processor to the list of sharers. If `is_sharing` is false, remove it.

The directory provides an API for retrieving a directory entry for each cache line as follows:

`touch(address)` If the given address is not in the directory, allocate an entry for it and set the share state to `INVALID`.

`get_entry(address)` Return the directory entry for the given address.

4.4 Protocol Implementation

Each protocol implements four functions, each corresponding to a different event that the protocol must process. At the simulator top level, the trace feeds read and write events to the protocol. As the protocol manipulates the directories and the local caches, those objects also generate events to the protocol via two callback functions. The interface is as follows:

`read(address, processor)` Invoked by the top level whenever the trace specifies a read event.

`write(address, processor)` Invoked by the top level whenever the trace specifies a write event.

`evict(address, processor, state)` Invoked by a processor's local cache when it must evict a line on a write miss. Depending on the state of the line being evicted, the protocol may need to write it back.

`invalidate(address, processor)` Invoked by the directory when using a directory entry format with a limited number of sharer pointers and an old pointer must be deallocated to make room for a new sharer (would be needed to implement DirI_{NB}).

4.5 Output

The protocol module keeps track of the number of messages sent. These statistics will form the basis for the protocol comparison. The following counters are maintained and dumped to `stdout` at the end of the simulation:

`bus_reads` Number of times data is read into a cache.

`bus_writes` Number of write-backs to memory.

`invalidations` Number of invalidation messages sent. An invalidation to n nodes counts as n. Unused in snoopy mode.

`snoopy_invalidations` Number of invalidation broadcasts in snoopy mode. Unused in directory mode.

`updates` The number of point-to-point update messages in directory mode. Unused in snoopy mode.

`snoopy_updates` Number of update broadcasts in snoopy mode. Unused in directory mode.

`writeback_reqs` Number of point-to-point messages from the directory to caches requesting the caches to write back a specified line.

`make_dirty_msgs` Number of point-to-point messages from the caches to the directory requesting that the directory mark the line as dirty.

`eviction notification` How many lines have been evicted from the caches due to conflicts (not bus traffic).

5 Results

This section uses the cache simulator of section 4 to measure network traffic. For all of the simulations, the cache size is 128KB with 4-way set asscoiativity and 64-byte lines unless otherwise noted. The working sets for all benchmarks except Radix fit comfortably within the caches, so conflict and capacity misses should be at a minimum.

5.1 Sparse Directories

To test the cache simulator and evaluate the three directory formats, a memory trace was generated stochastically. 64 processors access memory addresses with a gaussian random distribution. The center address μ is the same for each processor and $\sigma = s/2$ where s is the cache size in bytes. Figure 1 shows the total amount of network traffic on a trace with ten million accesses, 30% of which are writes. The full directory Dir$N$ scheme caused 27,033,133 coherence messages. Since a sparse directory for a 64 processor machine requires $\lceil lg(64) \rceil \times i$ bits, it is only profitable to build the sparse directory with fewer than eleven pointers. Within the feasible range for i, DirI_{CV} performs significantly better than Dir$I$, approaching optimality at $i = 6$.

In order to understand behavior of sparse directories on the SPLASH/SPLASH-2 benchmarks, each benchmark was executed on a 64-processor system using the `inval3` protocol for all three directory formats. Figure 2 shows the effect of varying the number of sparse directory pointers from 1 to 8 for the mp3d and FFT benchmarks. Figure 3 varies the same parameter for the LU and FFT benchmarks. LU and Radix benefit from the coarse vector optimization much more than mp3d and FFT. With three pointers, the coarse vector optimization reduces network traffic over Dir$I$ by 36% for

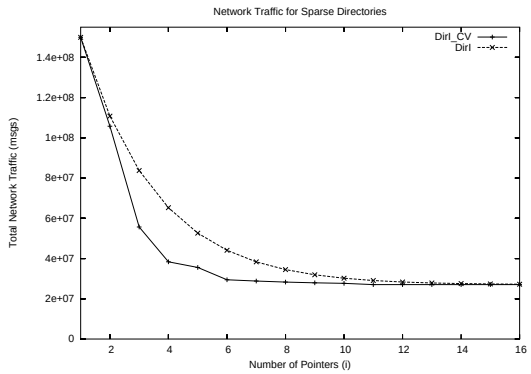

Figure 1. Network traffic for `inval3` protocol with gaussian random memory accesses on a 64-processor system with sparse directories.

LU, 40% for Radix, 16% for FFT, and 14% for mp3d.

Having characterized the directory performance and seeing that the Dir$I$ scheme performs much more poorly than DirI_{CV}, the remainder of this paper focuses on determining the scalability of the various coherence schemes given a full directory. For the four benchmarks examined here, a DirI_{CV} directory with six pointers would have performance almost equivalent to the full directory assuming a 64-processor system.

5.2 Directory-Based Protocols

Figures 4, 5, and 6 show the total number of coherence messages for a directory-based system running `inval3`, `illinois`, and `firefly` respectively. The results for `illinois` and `inval3` are practically indistinguishable, but `firefly` produces much more traffic than the invalidation protocols in most cases. For invalidation protocols, LU traffic is expectes to scale as $O(\sqrt{N})$ where N is the number of processors [4]. FFT and Radix are predicted to be $O(1)$. Both predictions appear to hold[2].

Another important metric is the number of

[2]Note that the plots use a log-log scale

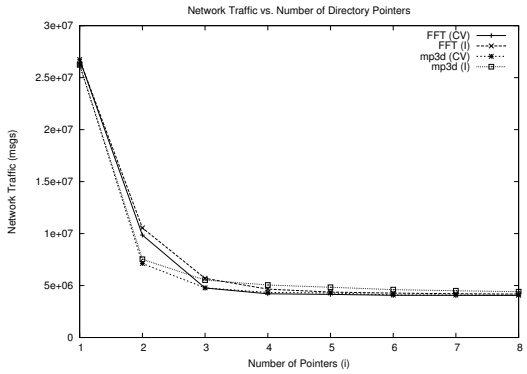

Figure 2. Network traffic under `inval3` protocol for mp3d and FFT benchmarks on a 64-processor system with sparse directories.

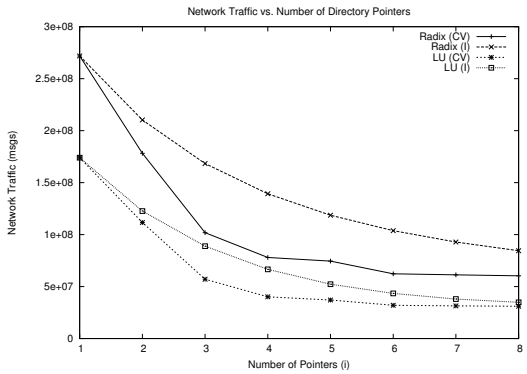

Figure 3. Network traffic under `inval3` protocol for LU and Radix benchmarks on a 64-processor system with sparse directories.

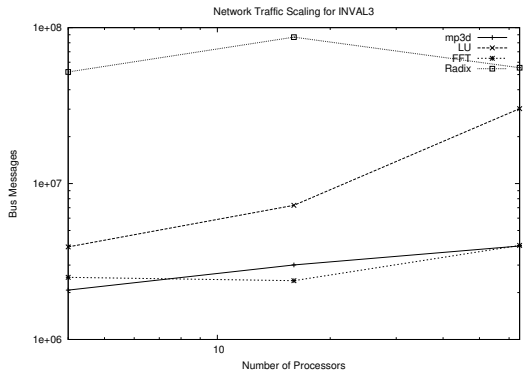

Figure 4. Network traffic for `inval3` as N scales.

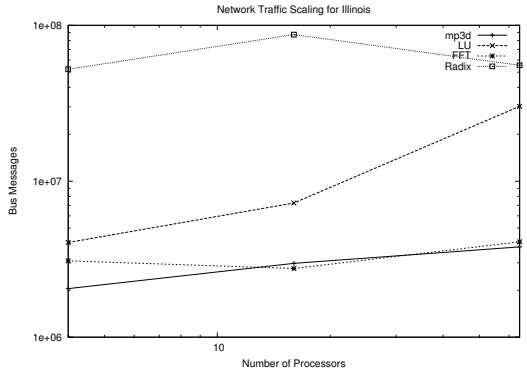

Figure 5. Network traffic for `illinois` as N scales.

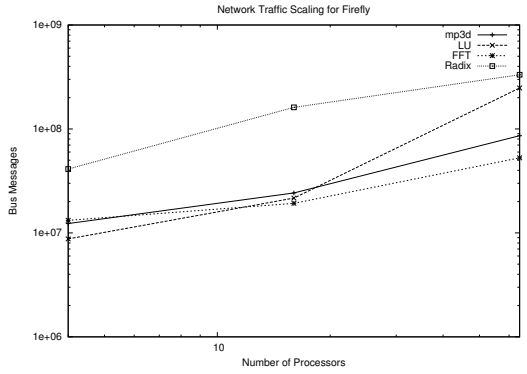

Figure 6. Network traffic for `firefly` as N scales.

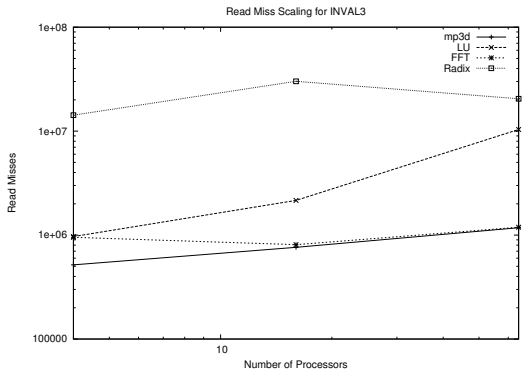

Figure 7. Bus reads under `inval3` as N scales.

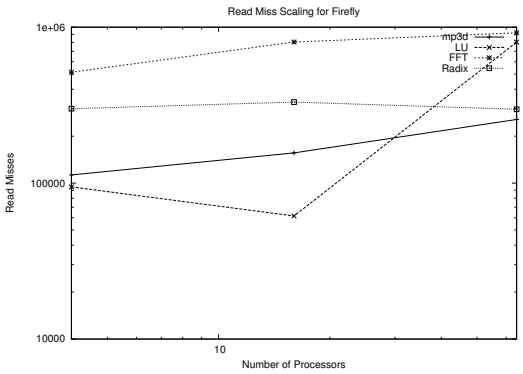

Figure 9. Bus reads for `firefly` as N scales.

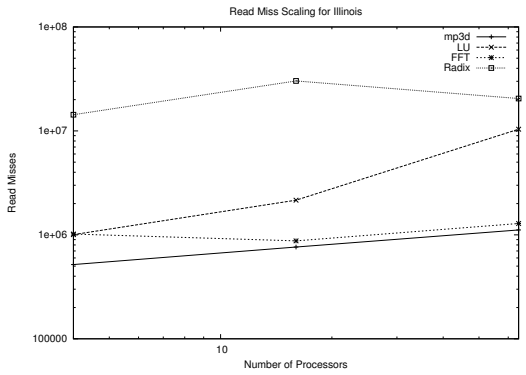

Figure 8. Bus reads under `illinois` as N scales.

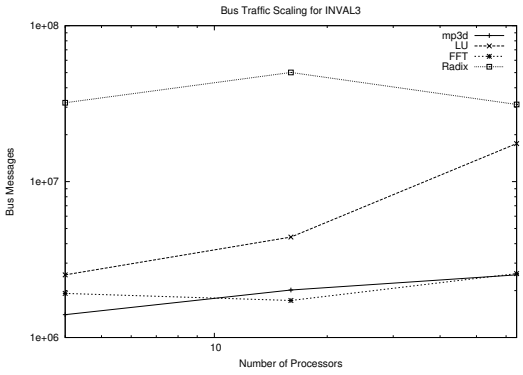

Figure 10. Bus traffic for snoopy `inval3` as N scales.

bus reads incurred by each protocol. Assuming a network of sufficient capacity to handle all of the protocol messages, bus reads will be the main cause of processor stalls. Figures 7, 8, and 9 show how the number of bus reads scales with system size for `inval3`, `illinois`, and `firefly` respectively. Here, Firefly is the clear winner with an order of magnitude fewer bus reads in many cases.

5.3 Snoopy Protocols

If the coherence protocol remains unchanged, the number of bus reads remains constant when switching from a Dir$N$ directory-based system to a snoopy system. However, the number of protocol messages changes since the snoopy bus allows invalidation and update messages to be broadcast. Figures 10, 11, and 12 show the message counts for the benchmark traces, taking advantage of the broadcast capability. In these plots, invalidating or updating all processors in the system costs only 1 message.

6 Analysis

This section contains a discussion of each protocol's strengthss and weaknesses, based on the data from the previous section. Topics include the apparent poor performance of Firefly,

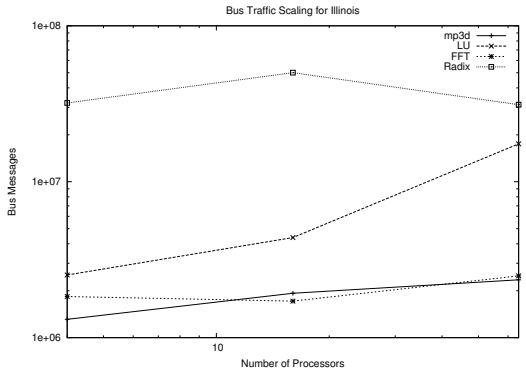

Figure 11. Bus traffic for snoopy `illinois` as N scales.

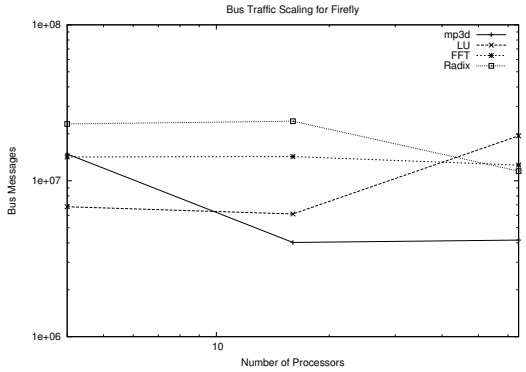

Figure 12. Bus traffic for snoopy `firefly` as N scales.

an evaluation of sparse directory performance, and an explanation of the apparent equivalence between Illinois and three-state invalidation protocols.

6.1 Update vs. Invalidation

In general, it is expected that write-update protocols will perform well when shared data exhibits the producer-consumer pattern. In the presence of other sharing patterns, they produce large amounts of bus traffic but still deliver the lowest-possible cache miss rate. The write-invalidate protocols have modest bus traffic for all sharing patterns but suffer high miss rates in the producer-consumer pattern. Unsurprisingly, for the scientific SPLASH-2 workload, update protocols scaled poorly. One exception was Radix, where each processor writes to a shared buffer after performing its local sort. Each processor then reads the data from the shared buffer.

The Firefly protocol's high message rates on te SPLASH benchmarks are justified by the fact that it was originally designed for a small-scale SMP workstation; it was never intended to be directory-based or to be scaled to a large number of processors. Workstation-class UNIX workloads *would* benefit from a write-update protocol, since the traditional parallel programming model on UNIX calls for multiple processes communicating through sockets or pipes. The processes share no memory except for the kernel buffers used for communication. Pipes implemented on top of shared memory exhibit exactly the kind of producer-consumer behavior at which Firefly excels. Firefly also has the advantage of guaranteeing a lower read-miss rate than the invalidation protocols at the expense of greater bus traffic.

With two or four processors, the extra traffic is probably manageable, but problems can arise with process migration on even a small system. Each time a process is scheduled onto a different CPU than the one on which it previously ran, the coherence protocol will continue to update stale lines in the old proces-

sor's cache. Consequently, in spite of its excellent handling of sockets and pipes, Firefly has fallen out of favor and most current SMP systems use invalidation protocols. Invalidation protocols efficiently support a wider variety of communications patterns.

In an invalidation protocol, a processor must fetch the line from another cache on a dirty miss. A processor that writes to a shared line needs only send one broadcast message to invalidate all other shared copies. Therefore, if a line is written by one processor before it is read by another processor, then the invalidation protocols will generate exactly two bus cycles: an invalidation and a read miss serviced by the dirty cache. The number of writes before the read does not matter. However, in an update protocol, the number of bus transactions is exactly equal to the number of writes before the second processor reads. The two protocols are even in terms of network traffic when a cache line is written twice before it is read. If the number of writes is greater on average, then invalidation protocols win. In general, for applications that are parallelized by pipelining, Firefly will be superior. In most other cases, like the SPLASH benchmarks, invalidation protocols will generate less network traffic.

6.1.1 Illinois vs. Three-state Invalidation

Overall, the `illinois` and `inval3` protocols perform approximately the same on the SPLASH benchmarks we examined. The difference in the protocols is the `exclusive` state present only in Illinois. This extra state allows Illinois to perform better when there is data that exists in only one cache at a time. When compared to three-state invalidation, Illinois will produce one fewer protocol transaction when un-shared data is read into a cache and modified. This is because no invalidation signal is needed to go from `exclusive` to `dirty` whereas `inval3` must send an invalidation to transition from `shared` to `dirty`.

While this works in favor of the Illinois protocol in some cases, it is detrimental in the event of multiprocessor reads. If a directory scheme is used with Illinois, the directory controller must send a writeback request to the `exclusive` processor if a second processor attempts to read the line. This is because the directory cannot distinquish between the `dirty` and `exclusive` states (the cache does not notify it on the `exclusive` → `dirty` transition). In the SPLASH benchmarks, the beneficial and detrimental features approximately cancel.

6.2 Snoopy vs. Directory

At first glance, the snoopy protocols appears to work much better than directory schemes because the total bus traffic (measured by number of messages) is far lower. Although a snoopy hardware scheme requires significantly less total bandwidth than a directory scheme, it requires a bus. This single bus grows more congested as the system size increases. The difficulties of constructing a snoopy bus system that can both accomodate the high amount of traffic that would pass across it, as well as maintain a high clock rate over potentially large distances as found in a multi-board system prevent the snoopy protocol from finding use in large systems. The SGI Challenge computers of the mid-90's scaled to only 12 processors using a snoopy protocol. Directory schemes surpass snoopy ones in terms of scalability, and are more suited to large shared memory systems.

With the advent of CMPs, the snoopy protocols may become a viable alternative to directory schemes. The physical proximity of processing elements would mean that a snoopy bus could theoretically maintain a high clock rate. However, contention might still be high. In addition, each cache must perform more tag checks in a snoopy system than in a directory system; each PE must listen to every request on the bus and query its cache to find out if the line in question needs to be updated, invalidated, or sent out to another PE. Thus, even if overhead on the snoopy bus line itself were manageable, hardware complexity of the caches

might be increased.

6.3 Directory Protocol Scaling

As shown in Figure 1 and Figure 3, the DirI_{CV} scheme scales much better than Dir$I$. Using less than half the space of the full directory, the coarse vector sparse directory generates about the same amount of network traffic on the SPLASH benchmarks. Also of interest is Figure 2, where Dir$I$ performs quite well. This is due to the 'all-or-nothing' communications patterns in the LU and Radix applications. The buffers processors use to do their local sorting are not shared, and the global histogram is shared by all. The broadcast bit in the Dir$I$ scheme captres both states. In general, though, there is no reason to build Dir$I$ directories without the coarse vector optimization, since the optimization should never hurt performance.

7 Conclusions

We have presented a trace-driven cache simulator, which we used to obtain statistics on coherence protocol network traffic. We have studied the behavior of three coherence protocols with three different directory formats and found our results to be in accordance with existing literature. Finally, we have provided a discussion of the specific conditions under which a particular coherence protocol or directory format should be selected. Non-scientific workloads were not featured in our empirical study, and would be an interesting topic for future research.

References

[1] A. Agarwal, A. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *ISCA*, pages 280–289, May 1988.

[2] J. Archibald and J. L. Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[3] D. H. Bailey. FFT's in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, March 1990.

[4] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, 1993.

[5] V. Pai. Rsim: An execution-driven simulator for ilp-based shared-memory multiprocessors and uniprocessors, 1997.

[6] Papamarcos and J. Patel. A low overhead coherence solution for multiprocessors with private cache. *ISCA*, 1984.

[7] J. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. *Technical Report CSLTR-91-469, Stanford University*, 1991.

[8] C. Thacker, L. Stewart, and E. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[9] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *ISCA*, June 1995.