[Lenoski *et al.*, 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Lenoski *et al.*, 1992] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance," In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[Mellor-Crummey and Scott, 1991] J. M. Mellor-Crummey and M. L. Scott, "Synchronization Without Contention," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 269–278, April 1991.

[MIPS Computer Systems Inc., 1991] MIPS Computer Systems Inc., *MIPS R4000 Microprocessor User's Manual*, Integrated Device Technology, Inc., 1991.

[Patel and Harrison, 1988] N. M. Patel and P. G. Harrison, "On Hot-Spot Contention in Interconnection Networks," *Performance Evaluation Review*, 16(1):114–123, May 1988, Originally published at SIGMETRICS '88.

[Pfister and Norton, 1985] G. F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Singh *et al.*, 1992] J.P. Singh, W-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1):5–44, March 1992.

[Thomas, 1986] R. H. Thomas, "Behavior of the Butterfly Parallel Processor in the Presence of Memory Hot Spots," *Proceedings of the 1986 International Conference on Parallel Processing*, pages 46–50, August 1986.

[Wittie and Maples, 1989] Larry Wittie and Creve Maples, "MERLIN: Massively Parallel Heterogeneous Computing," In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–142 – I–150, August 1989.

[Yew *et al.*, 1987] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

References

[Agarwal *et al.*, 1992] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.-H. Lim, G. Maa, and D. Nussbaum, “The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor,” In M. Dubois and S. S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1992.

[Agarwal and Gupta, 1988] A. Agarwal and A. Gupta, “Memory Reference Characteristics of Multiprocessor Applications under Mach,” *Performance Evaluation Review*, 16(1):215–225, May 1988, Originally published at SIGMETRICS ’88.

[Agarwal, 1992] Anant Agarwal, “Performance Tradeoffs in Multithreaded Processors,” *IEEE Transactions on Parallel and Distributed Processing*, 3(5):525–539, September 1992.

[Bianchini *et al.*, 1993] R. Bianchini, M. E. Crovella, L. Kontothanasis, and T. J. LeBlanc, “Alleviating Memory Contention in Matrix Computations on Large-Scale Shared-Memory Multiprocessors,” Technical Report 449, Department of Computer Science, University of Rochester, April 1993.

[Chaiken *et al.*, 1990] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, “Directory-Based Cache Coherence in Large-Scale Multiprocessors,” *IEEE Computer*, 23(6):49–58, June 1990.

[Cheriton *et al.*, 1991] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick, “Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience,” *Proceedings of the International Symposium on Shared-Memory Multiprocessing*, pages 109–118, 1991.

[Darema-Rogers *et al.*, 1987] F. Darema-Rogers, G.F. Pfister, and K. So, “Memory Access Patterns of Parallel Scientific Programs,” *Performance Evaluation Review*, 15(1):46–57, May 1987, Originally published at SIGMETRICS ’87.

[Davis *et al.*, 1991] Helen Davis, Stephen R. Goldschmidt, and John Hennessy, “Multiprocessor Simulation and Tracing Using Tango,” In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–99 – II–107, August 1991.

[Glenn *et al.*, 1991] R. R. Glenn, D. V. Pryor, J. M. Conroy, and T. Johnson, “Characterizing Memory Hot Spots in a Shared-Memory MIMD Machine,” *Proceedings of Supercomputing’91*, pages 554–566, November 1991.

[Glenn and Pryor, 1991] Raymond R. Glenn and Daniel V. Pryor, “Instrumentation for a Massively Parallel MIMD Application,” *Journal of Parallel and Distributed Computing*, 12(3):223–236, July 1991.

[Ho and Eager, 1989] Wing S. Ho and Derek L. Eager, “A Novel Strategy for Controlling Hot Spot Congestion,” In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–14 – I–18, August 1989.

[Kendall Square Research Corp., 1992] Kendall Square Research Corp., *KSR1 Principles of Operation*, Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA, 1992.

amount of cache space, but this is not so. Assuming an even distribution of servers throughout the machine, the maximum additional cache space needed per processor is *HotDataSize * NumServers / NumProcs*, where *HotDataSize* is the size of the hot data, *NumServers* is the number of servers per hot cache block, and *NumProcs* is the number of processors in the machine. This is a worst-case analysis however; in practice, a server need only store the hot data currently being referenced by clients. In addition, under the assumption that servers are also clients for hot data, then each server needs a copy of the data anyway. Furthermore, the extra cache space devoted to copies of hot data in servers is a small percentage of the cache space devoted to caching application data. In short, the space overhead of servers is not an impediment to data replication as used in eager combining.

5 Conclusions

Our simulation results clearly indicate that memory contention will have a serious impact on application performance as we scale cache-coherent multiprocessors to hundreds of nodes. We observed that for some applications the presence of hot spots completely negates any running time improvements from additional processors. For five of our applications, hot spot contention increased running time by 47%, 135%, 219%, 516%, and 3126% on machines with 200 or more nodes. Our studies of these programs suggest that hot spots are a characteristic of the way shared memory programs are written. For many programs: 1) hot spots are distributed throughout memory; 2) individual hot spots worsen with an increase in the number of processors; and 3) the number of hot spots increases with an increase in processors. Requiring that the programmer remove individual hot spots by hand is neither desirable nor particularly effective; other hot spots worsen, and it is probably not possible to eliminate enough hot spots by hand to eliminate memory contention effects.

These observations about the nature of hot spots suggest that memory contention needs to be addressed at the architecture level. We considered two options: queueing requests at the memory module, thereby increasing memory utilization during periods of contention, and automatic data replication in the coherency protocol. Our results indicate that queueing is not effective enough, and can require long queues in hardware. On the other hand, we were able to eliminate most hot spots encountered in our programs using an eager combining protocol. Based on these results, we conclude that hardware-supported replication is an attractive way to deal with memory contention, especially when used in conjunction with application restructuring techniques designed to eliminate the few extreme cases of hot spots.

Application	Running Time Increase	
	64 processors	256 processors
`mp3d`	4%	20%
`mp3d2`	0%	1%
`barnes-hut`	0%	0%
	100 processors	200 processors
`bmatmult`	0%	0%
`sgauss`	1%	1%
`mgauss`	0%	0%
`matinv`	6%	18%
`tclosure`	0%	0%
`all-pairs`	83%	303%

Table 10: Running Time Increase Due to Contention Under Eager Combining

under eager combining, improving on the speedup of 74 under queueing. `Matinv`, which was only able to achieve a speedup of 43 on 200 processors under queueing, is able to achieve a speedup of 99 under eager combining. `Tclosure`, which exhibited the best speedup under queueing, was able to improve speedup from 124 to 154 when using eager combining in place of queueing.

Most applications perform well under eager combining; for `mp3d2`, `barnes-hut`, `sgauss`, `mgauss` and `tclosure`, the increase in running time due to contention is less than 2%. Although three applications (`mp3d`, `matinv` and `all-pairs`) still exhibit noticeable contention, eager combining dramatically reduces the overhead due to contention, especially in comparison to the model in which busy memory modules reject arriving requests.

Two approaches are possible to further alleviate contention in these programs. We can eliminate selected hot spots in software, keeping the number of servers at 10; alternatively, we can increase the number of servers participating in the eager combining protocol. We evaluated these two approaches through two simple experiments with `mp3d` on 256 processors. The first approach is applicable because, as discussed in Section 3.4, `mp3d` has five pages that have a very high degree of contention, while the other pages exhibit more moderate amounts of contention. Thus, it may be appropriate to eliminate those five hot spots by restructuring the program, while allowing eager combining to handle the more widespread, lower-contention hot spots. In fact, assuming the five hottest pages are contention free and using eager combining for the other pages, we were able to reduce the runtime overhead due to contention from 20% to 13%. This result, while not conclusive, suggests that the use of application-level restructuring may be an effective complement to hardware-supported data replication in removing hot spots from certain applications.

In our second experiment, we used 20 servers for each hot cache block, again on a 256-processor machine. In this case, the run time overhead due to contention in `mp3d` was reduced from 20% to 15%. This result demonstrates that it is profitable to replicate hot data on a larger number of servers in order to further reduce contention. As another example, we were able to reduce the contention overhead of `all-pairs` from 303% to 127% by using 20 servers per hot block on a 256-processor machine.

It may seem that replicating cache lines in a large number of servers consumes an excessive

Application	Latency	
	64 processors	256 processors
`mp3d`	85.6	99.4
`mp3d2`	83.4	101.4
`barnes-hut`	83.3	86.4
	100 processors	200 processors
`bmatmult`	82.0	82.2
`sgauss`	82.3	83.0
`mgauss`	82.6	86.1
`matinv`	91.4	124.9
`tclosure`	82.2	84.3
`all-pairs`	550.5	1232.8

Table 9: Remote Memory Latency Under Eager Combining

For example, on 200 or more processors, memory queues must be able to hold at least 76 outstanding requests for `mp3d2`, 116 outstanding requests for `matinv`, and 175 outstanding requests for `all-pairs`. It is obvious from these results that small queues will not significantly reduce the effects of contention for these programs.

Table 8 presents the increase in running time attributable to contention under the queueing memory model. While queueing at the memory helps reduce the performance impact of contention, it does not solve the problem entirely, since several applications still suffer from a high degree of contention. In particular, when running on 200 or more processors, contention increases the running time of `mp3d` by 26%, `mgauss` by 41%, `matinv` by 173%, and `all-pairs` by 417%. This increase in running time due to contention has a serious effect on speedup; `mgauss` has a speedup of only 74 on 200 processors, while `matinv` is even worse, with a speedup of 49 on 100 processors, and 43 on 200 processors.

Eager Combining

The previous section showed that, when memory modules are kept constantly loaded during periods of contention, long queues develop, and contention still keeps some programs from performing well. A method of radically increasing the effective memory bandwidth is needed. This section presents an evaluation of our proposed method, eager combining, showing its effectiveness at alleviating memory contention.

Tables 9 and 10 present the performance of our application suite under eager combining. We used 10 servers in our implementation of eager combining; this particular value represents a compromise between the additional memory used and the performance improvement of the application. As seen in the tables, the average memory latency and the percentage increase in running time due to memory contention are lower for eager combining than they are for queueing. In four cases (`mgauss`, `matinv`, `tclosure`, and `all-pairs`) the differences between eager combining and queueing are significant, and in three cases (`mgauss`, `matinv` and `tclosure`) these differences translate to a significant improvement in speedup. On 200 processors, `mgauss` attained a speedup of 102

Application	Latency	Queue Len.	Latency	Queue Len.
	64 processors		256 processors	
`mp3d`	86.6	5	112.5	8
`mp3d2`	84.0	11	109.9	76
`barnes-hut`	NA	NA	NA	NA
	100 processors		200 processors	
`bmatmult`	82.2	3	82.4	3
`sgauss`	85.1	7	97.5	10
`mgauss`	114.0	18	232.2	43
`matinv`	443.6	58	822.5	116
`tclosure`	152.1	19	411.7	44
`all-pairs`	820.6	77	1797.0	175

Table 7: Remote Memory Latency and Queue Lengths Under Queueing

Application	Running Time Increase	
	64 processors	256 processors
`mp3d`	5%	26%
`mp3d2`	0%	2%
`barnes-hut`	NA	NA
	100 processors	200 processors
`bmatmult`	0%	0%
`sgauss`	2%	4%
`mgauss`	5%	41%
`matinv`	52%	173%
`tclosure`	3%	23%
`all-pairs`	111%	417%

Table 8: Running Time Increase Due to Contention Under Queueing

Consistency Model	Protocol	Total Number of Messages	Number of Hops
Sequential	EC DASH-like	(2 * S + 2 * (C - SC)) + 2 (2 * C) + 2	(2 * S + 2 * (C - SC)) + 2 (2 * C) + 2
Release	EC DASH	(2 * S + 2 * (C - SC)) + 2 (2 * C) + 2	2 2

Table 5: Messages transferred in coherency actions (read-shared to modified)

Protocol	Total Number of Messages	Number of Hops
EC	S + 5	4
DASH	4	3

Table 6: Messages transferred in sharing actions (modified to read-shared)

in the tables, eager combining may employ more messages than the DASH protocol when making a transition from *read-shared* to *modified* or *modified* to *read-shared*, because hot data blocks are sent to servers whether or not the servers use the data, and both the servers and clients must be kept consistent. These extra messages are unlikely to be a serious problem however, since we expect hot data blocks to be accessed by most processors (including the servers), and the extra messages required to replicate data to servers can be overlapped under any consistency model. Also, it is not necessary to wait for invalidation acknowledgements if sequential consistency is not required. Therefore, under a relaxed consistency model, eager combining is not likely to impose significantly greater communication latency than the DASH protocol. As shown by the number of hops in the critical path in tables 5 and 6, eager combining and DASH require roughly the same number of hops for each state transition, under the assumption that each server actually requires the data it provides to its clients. Our simulations assume a relaxed consistency model, and therefore do not include any waiting time in the coherency protocol.

4.3 Evaluation

Queueing at the Memory Module

In the simulation results presented in Section 3.2, a memory module simply rejects any requests that arrive when it is busy. Rejected requests suffer a round-trip penalty, since the original request must be re-issued. Moreover, the memory module can go under-utilized while waiting for re-issued requests to arrive, especially if the network latency is large. By queueing requests at the memory module rather than rejecting them, we can guarantee that the memory module is fully utilized while it is hot.

Tables 7 and 8 present the performance of our set of applications under the queueing memory model. Table 7 presents the average latency for a remote request, and the average length of the queue when a request arrives at a module under contention. The average queue length can be considered a lower bound on the queue capacity needed to achieve the average latency. Table 7 shows that queueing is reasonably effective at reducing latency, but that very long queues develop.

the server doesn't have a copy of the block, or if the server has modified its copy of the block, the server forwards the client's request to the home node. Subsequent requests from other clients for the same data block are queued at the server until it receives the block from the home node.

Upon receiving a forwarded read request, the home node proceeds according to the state of the data block. If the block is not in the *modified* state, the home node sends the block to the client directly. Otherwise, the home node forwards the request to the current owner of the block. As in the DASH protocol, the owner transmits the data block to both the requester and the home node. On receiving the updated contents of the block, the home node sends a copy to each of the servers for the block, and sets the state of the block to *read-shared*. This multicast from the home node to the servers can be overlapped with computation on all nodes.

It is important to note that the home node does not multicast a data block to its servers each time the block is written. The multicast takes place only on the transition from *modified* to *read-shared*. Thus, we avoid eager sharing of partially modified data blocks. Nonetheless, eager combining could exacerbate any adverse performance effects caused by fine-grain sharing and false sharing.

When a client issues a write to a hot data block, a request for ownership is sent directly to the home node, bypassing the server. On receiving this request, the home node proceeds according to the current state of the data block. If the block is in the *modified* state, the protocol proceeds exactly as in DASH. That is, the request is forwarded to the current owner of the block, which transfers ownership to the requesting node, and requests that the home node update the ownership of the block. If the block is in the *read-shared* state, the home node must invalidate all copies, both in the servers and clients.

To implement these invalidations, the home node sends invalidation messages to servers, who then pass on invalidations to their clients. In this scheme, the directory information in the home node is consistent with respect to the state of a data block and the number of servers, but not the number of clients. When a write request reaches the home node, the home sends the data to the new owner, and tells the new owner the number of servers containing copies of the data block. The home node sends invalidation messages to each sever, which then send invalidation messages to each client. When the clients have all acknowledged the invalidation to their server, the server sends an acknowledgement to the new owner.

In comparison to the DASH protocol, this invalidation scheme reduces the total amount of work required of the home node to service hot data blocks (under the assumption that servers for a data block usually access the data block). Instead of distributing copies of a hot data block to all processors (as in DASH), the home node need only send copies to the servers for the block. Most read requests are satisfied by servers, and therefore never reach the home node. If a server must forward a read request to the home node on behalf of one client, that request will generate a single response that will satisfy requests from the other clients of the server. In general, this scheme reduces the number of messages the home node must send, since the number of servers for a hot block is expected to be much smaller than the number of processors using the block.

Eager combining is not without costs, however. Tables 5 and 6 present a comparison between the number of messages involved in the DASH protocol and in eager combining. In these tables, S stands for the number of servers per hot data block, C is the total number of clients of the hot block, and SC is the number of servers that are also clients of the hot block. The number of hops referred to in the tables is the number of messages in the critical path of the protocol. As observed

4.2 A Coherency Protocol Incorporating Eager Combining

In this section we describe a coherency protocol that implements data replication for hot spots. Our purpose is to show that the coherency protocol can be used to alleviate contention in direct-connected, distributed-shared-memory multiprocessors.

We assume certain physical address ranges are marked *hot*, and these addresses are treated specially by the coherence protocol. Two ways to accomplish this are:

- The choice of protocol could be selected on a per-page basis; an analogous feature is already present in the MIPS R4000 cache coherence controller [MIPS Computer Systems Inc., 1991].
- A portion of the physical address space of the machine could be permanently set aside for hot data.

Which of these approaches is chosen depends on tradeoffs involving cache memory cost, the benefits of dynamic protocol selection, and the ability of the compiler or programmer to precisely identify hot data ranges. Our simulation results assume that all shared data is marked as *hot.*

Our basic approach is to designate a fixed number of "server nodes" for each hot physical page, assigning to each server some subset of the remaining nodes as clients. The protocol uses eager sharing to distribute data to servers, which then satisfy requests from multiple client nodes. Multiple requests that cannot be satisfied immediately by a server are combined to reduce the traffic directed to a hot spot. Since our approach incorporates the properties of both eager sharing and combining trees, we call it *eager combining.*

We use the DASH cache coherence protocol [Lenoski *et al.*, 1990] as a starting point for our eager combining protocol. Each data block in DASH is assigned to a memory module, and that module's node is referred to as the data block's *home node*. In the eager combining version of the protocol, we designate a fixed number of *server nodes* for each hot data block, which are determined statically from the physical page number. As with regular data blocks, hot data blocks can be in one of three states: *uncached, read-shared,* and *modified.* We make three modifications to the DASH protocol in order to handle hot data blocks:

- Reads to a hot data block are directed to a server rather than to the home node;
- When a hot data block makes a transition from *modified* to *read-shared,* the block's home node multicasts the data to the block's servers;
- When a hot data block makes a transition from *read-shared* to *modified,* the clients and the servers must have their copies of the block invalidated.

When a node makes a read request on a hot data block, the request goes directly to the proper server, which is selected based on the requester's node number and the physical page number.[2] If the server has an unmodified copy of the block, it immediately sends the block to the requester. If

[2]The mapping between clients and servers can be static or dynamic. Static mapping is simplest; dynamic mapping can help to eliminate contention for server memory when a static mapping happens to map contending processors to the same server. Our results are based on static server selection; however, we have examined dynamic server selection experimentally and found that it does not significantly improve on static selection for our applications.

memory module. Distributing data across more memory modules (pages) requires smaller pages or discontiguous allocation, neither of which is desirable.

We can also reduce the effect of memory contention by queueing requests at the memory, rather than rejecting them when the memory module is busy. Queueing requests allows a memory module to be continuously utilized during periods of contention. We evaluate this approach in Section 4.3, and show that queueing reduces, but does not eliminate, the effects of contention. In addition, we show that queueing requires hardware support for very long queues.

Another conceivable strategy for reducing contention is to use some kind of back-off strategy, in which a processor that is rejected by a busy memory module waits for a small random period of time before reissuing the request. However, random back-off is likely to leave memory modules underutilized during periods of contention, and in the best case approaches the performance of queueing at the memory module. For this reason, we do not consider this option any further.

Since queueing at the module does not fully alleviate contention, we concentrate on still another way of dealing with contention: data replication. Data replication can be either producer-driven or consumer-driven. Producer-driven approaches are referred to as *eager sharing*. In eager sharing, the producer of a data block actively transmits it to the nodes that will subsequently need it. An efficient implementation of eager sharing requires that we 1) identify the recipients of the data to be multicast and 2) broadcast the data block only when the producer is finished modifying the entire block.

Consumer-driven data replication approaches are referred to as *combining trees* [Yew *et al.*, 1987]. In this approach, processors are organized into a tree structure, in which each processor requests the data block from its parent node, and the producer is at the root. Each parent node combines the requests of its children into a single request to its parent.

The relative feasibility of these two styles of data replication is dependent on the communication and addressing features of the underlying architecture. Scalable distributed-shared-memory architectures can be separated into two categories, based on whether a data block has a home. Architectures in which data has no fixed home are referred to as COMA (Cache Only Memory Architecture), e.g., the KSR1 [Kendall Square Research Corp., 1992]. Architectures in which data blocks have a home are referred to as CC-NUMA (Cache-Coherent Non Uniform Memory Access), e.g., DASH [Lenoski *et al.*, 1990].

COMA machines are usually based on broadcasting media, since processors must be able to easily locate any data blocks they access. For example, the KSR1 uses a hierarchy of rings; a processor broadcasts requests for data blocks on the local ring, which are forwarded up the hierarchy of rings if necessary to satisfy the request. Once data is brought into a local ring, subsequent requests for the data are satisfied by the local ring. In effect, the KSR1 implements a dynamic combining tree solution to memory contention, limiting it to at most 32 processors (the number of processors on the local ring). In addition, the KSR1 supports a form of programmer-specified eager sharing using the *post-store* instruction, which updates all cached copies of a replicated data block.

CC-NUMA machines, on the other hand, usually have no such broadcast medium, making it more difficult to either locate replicated data (in a combining tree approach) or to perform dynamic multicast (in an eager sharing approach). In Section 4.2 we propose a modification to the DASH coherency protocol that implements data replication in a CC-NUMA machine. We show how our protocol solves the problems associated with the lack of a broadcast medium, and then show in Section 4.3 that the protocol greatly alleviates memory contention in the applications we consider.

Application	Number of Hot Spots	
	64 processors	256 processors
mp3d	6	24
mp3d2	1	1
barnes-hut	NA	NA
	100 processors	200 processors
bmatmult	2	3
sgauss	87	326
mgauss	288	495
matinv	448	442
tclosure	401	512
all-pairs	400	400

Table 4: Number of Hot Spots

became even hotter, with latencies of 2764 and 2634 cycles; one new page became hot; and one more hot spot reached an average latency higher than 350.

Repeating this process by removing the hot spots with latency higher than 2000 cycles resulted in a similar effect. The application's performance does not improve much, still suffering a 67% degradation. The most important effect of removing these two hot spots was that another page became extremely hot with a latency of 1930 cycles.

After this step there were still 20 hot spots. We cannot expect to continue this process indefinitely, because in practice each hot spot removal would require the laborious identification of a hot physical page, determination of its virtual address, identification of the data structure residing at that address, and the restructuring of the computation on the data structure. In addition, this approach to contention elimination may not be always possible in the presence of arbitrary data structures.

These results, in combination with the results presented in the previous section, indicate that individual elimination of hot spots through software modification cannot be expected to completely alleviate the effects of memory contention in future machines. Hot spots must be addressed in a more systematic way, at the hardware level. The next section evaluates two such hardware approaches.

4 Reducing Memory Contention

4.1 Alternatives for Reducing Memory Contention

The memory contention we observed in our programs could conceivably be reduced by increasing total memory bandwidth. There are two ways of achieving such an increase: increasing the bandwidth of each memory module (through a wider and/or faster memory) or effectively utilizing more memories. Our simulations are already very aggressive in terms of the bandwidth of each

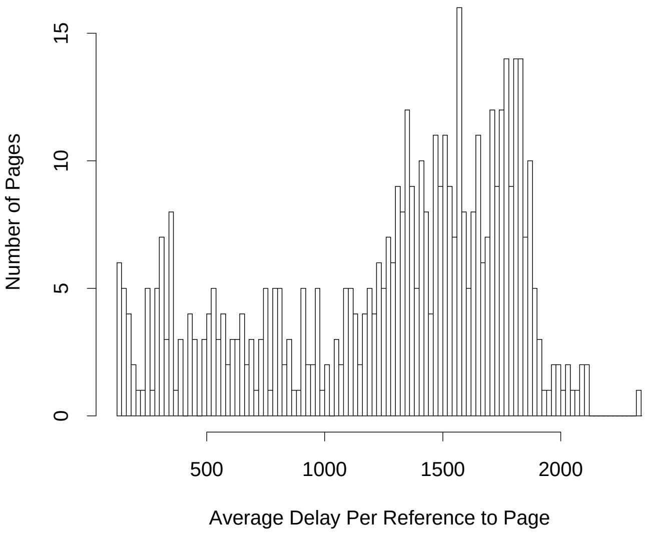

Figure 3: Distribution of Hot Spots in `mgauss`, 200 processors

Table 4 illustrates the growth in the number of hot spots in our applications that occurs when using larger numbers of processors. For some applications (`mp3d`, `sgauss`, `mgauss` and `tclosure`), the number of hot spots increases markedly as we move the application to a larger machine. For other applications (`mp3d2`, `bmatmult`, `matinv`, and `all-pairs`), the number of hot spots increases slightly or remains roughly constant. (For `all-pairs` the number of hot spots does not increase because all shared pages are hot, even on 100 processors.) Even in those cases where the number of hot spots does not increase significantly, pages become hotter as the number of processors is increased.

3.4 Persistent Aspects of Memory Contention

One response to the presence of hot spots might be to eliminate them on an individual basis, using software techniques. In this section we show that removing hot spots on an individual basis has the effect of causing other hot spots to worsen.

We performed selective hot-spot removal on `mp3d` because it has a few pages that are much hotter than the others. The average remote memory latency for `mp3d` on 256 processors is 302 cycles. One of its pages is extremely hot with an average latency of 4411 cycles. Four other pages are also very hot with average latencies of 2522, 2096, 1257 and 1222 cycles. No other page has average latency higher than 350 cycles.

We modified the trace analyzer so that references to the three hottest pages would always be serviced without contention costs. The result was that the average remote memory latency declined from 302 to 212, indicating a reasonably successful improvement to the application. However, the application still suffers a 76% degradation in running time due to the remaining contention. We observed three interesting effects after this modification: The two pages that were already very hot

Processors	Input Size	Runtime Increase
64	320×320	71%
128	400×400	333%
256	512×512	1004%

Table 3: Scaled Speedup Experiments With `mgauss`

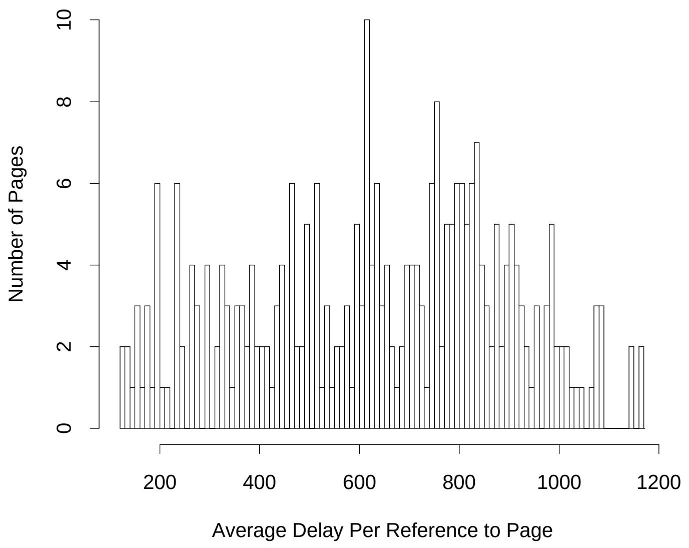

Figure 2: Distribution of Hot Spots in `mgauss`, 100 processors

- hot spots are spread widely throughout memory;
- the degree of contention for individual hot spots increases drastically with an increase in the number of processors; and
- the number of hot spots increases with an increasing number of processors.

Figure 2 shows the distribution of access times to memory for `mgauss` running on 100 processors. If we define a hot spot as a page with an average access time of more than 123 cycles (that is, a page whose average access time is 50% larger due to contention), then we see that hot-spot contention in this program is spread across a number of pages. Over 130 pages exhibit an average access time between 400 and 800 cycles, while 18 pages exhibit an average access time in excess of 1000 cycles.

Figure 3 presents the distribution of hot spots for `mgauss` on 200 processors. By comparing figures 2 and 3, we can see that the number of hot spots and the average access time per hot spot increases as we increase the number of processors. The number of hot pages is much larger in figure 3; more than 490 pages are hot on 200 processors. The average access times of the pages are also very high, reaching above 2000 cycles in ten cases. We observed this same effect for most of the application programs in our study.

reference the first element of a row in the matrix. However, serial access to the first element in a row tends to skew the requests for subsequent elements in that row, thereby avoiding contention for individual elements. Thus, factor (2) is primarily responsible for the contention seen in these programs.

We confirmed this conclusion using a simple experiment in which we simulated Gaussian elimination on 50 processors, using a matrix that was allocated so that elements within the same row were placed in different pages. This allocation strategy reduced the percentage of delayed references from 20% to 1.5%, and the average remote access latency from 164 cycles to 83 cycles. This experiment confirms that the memory contention seen in our kernels is due primarily to simultaneous access to the elements of a row, all of which reside in one memory module.

Note that `bmatmult` does not exhibit any memory contention, since the relatively small number of remote memory accesses in the program are not tightly synchronized.

In order to further study the contention behavior of programs, we simulated some programs on a larger range of machine sizes. Figure 1 shows the increase in running time attributable to contention for the `mgauss` application. This figure illustrates how rapidly contention-induced slowdown rises as an application is run on machines of increasing size. These simulation results were modeled in a more detailed analysis presented in [Bianchini *et al.*, 1993]. In that paper we showed that, beyond a certain number of processors, each additional processor adds the cost of an entire matrix's memory service time to the execution time of the program. Thus, an estimate of the overhead due to contention in `mgauss` (using lock synchronization), on a large number of processors (P) is:

$$OVH(P) = \frac{M}{E} C(P - \theta)$$

where C is the memory's service time (10 cycles), M is the number of elements in the entire matrix, E is the number of elements per cache line, and θ is the threshold number of processors beyond which the system shows memory saturation. This analysis shows that the effect of memory contention can be expected to grow linearly with increasing machine size.

In addition, it is important to note that some programs exhibit large amounts of contention, even when the input size is scaled up with the number of processors. Table 3 presents performance results of `mgauss` when we kept the amount of work per processor constant and thus kept the communication to computation ratio the same. Since communication is the source of contention, we would expect that maintaining the amount of work per processor constant should reduce the amount of contention seen. However, as can be seen in the table, increasing the problem size with the number of processors helps, but contention is still a serious problem. While contention increases the total running time by 71% on 64 processors, it increases running time by as much as 1004% when moving to 256 processors, even though the problem size has scaled with the number of processors.

3.3 Widespread Aspects of Memory Contention

In this section we show that the large amount of memory contention we find seems to be intrinsic to the way programs for shared-memory architectures are written. In particular, we show that for many applications:

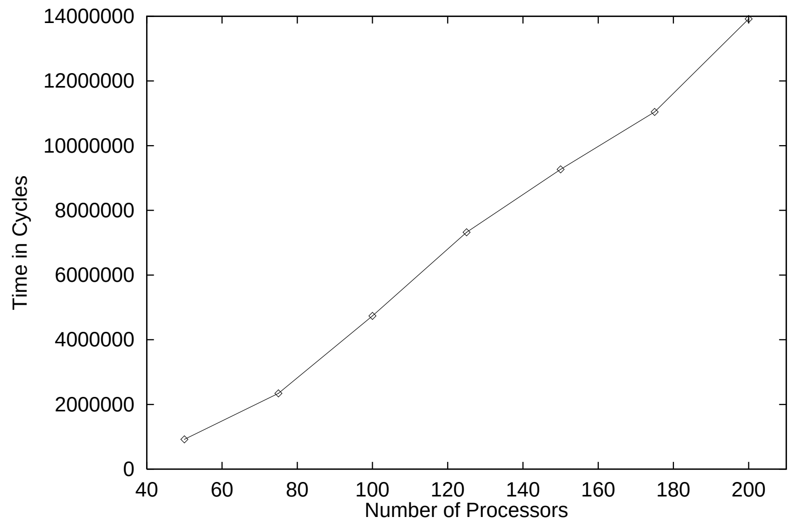

Figure 1: Increased Running Time Due to Contention in `mgauss`

The results in these two tables are disconcerting, considering that our simulation parameters are optimistic. The effects of contention are even worse if we relax some of our optimistic simulation assumptions. For example, if we double the memory latency to 20 processor cycles, the effect of contention is even more pronounced: on 200 processors, 92% of the misses in `mgauss` suffer contention (up from 84%), and the average remote reference latency increases to 2910 cycles (up from 1546). Similarly, if we keep memory latency at 10 cycles and reduce the number of bytes we fetch on a miss to 32 bytes, then 90% of the misses in `mgauss` suffer contention, and the average remote latency increases slightly to 1571 cycles. If we both double the memory latency and reduce the number of bytes fetched to 32 bytes, then the average remote latency increases to 2904 cycles.

The very poor performance results we observed are due to a number of application characteristics. `Mp3d`, for instance, is known to exhibit poor locality of reference, and frequent synchronization. Contention in `mp3d` is caused by very tightly synchronized processes accessing a data structure containing counters, and the reservoir of particles, which acts as a central pool of particle velocities. `Mp3d2` improves the locality characteristics of `mp3d`, and contention decreases markedly.

Our simulations show that `mgauss`, `matinv`, `tclosure` and `all-pairs` have good locality of reference and good speedup in the absence of contention; speedups of 114, 124, 154, and 144 (respectively) on 200 processors. However, Table 2 shows that these applications forfeit all their speedup to contention effects. The excessive memory contention seen in `sgauss`, `mgauss`, `matinv`, `tclosure` and `all-pairs` could have been caused by any of three factors: (1) simultaneous access to a single element of the main matrix, (2) simultaneous access to a single row of the main matrix (which resides in a single page, and therefore results in memory contention), and (3) simultaneous access to multiple rows that happen to reside in the same page. In all of our kernel examples we padded the rows of the matrix to fill a page, and therefore eliminated any contention caused by simultaneous access to multiple rows within a single page. Simultaneous access to a single element of the matrix *can* occur in our programs since, upon creation, all processes immediately try to

Application	Average Latency	
	64 processors	256 processors
`mp3d`	132.3	302.6
`mp3d2`	98.7	286.6
`barnes-hut`	97.3	180.6
	100 processors	200 processors
`bmatmult`	82.7	83.0
`sgauss`	100.1	232.8
`mgauss`	571.5	1546.7
`matinv`	535.6	990.5
`tclosure`	228.4	619.3
`all-pairs`	5924.1	12335.5

Table 1: Average Remote Memory Latency

Application	Running Time Increase	
	64 processors	256 processors
`mp3d`	40%	135%
`mp3d2`	2%	6%
`barnes-hut`	4%	15%
	100 processors	200 processors
`bmatmult`	0%	0%
`sgauss`	7%	30%
`mgauss`	112%	516%
`matinv`	68%	219%
`tclosure`	8%	47%
`all-pairs`	927%	3126%

Table 2: Running Time Increase Due to Memory Contention

from network contention; adding network contention to our simulations would assign some of the contention we observe to the network rather than the memory, but would not be likely to affect the trends and conclusions we present in this paper.

The principal outputs of the analyzer are the average remote memory latency, and the running time of the program in the presence of memory contention. In this paper, average remote memory latency means the average duration from the time the memory request is issued to the time the data is received. This average duration is an important metric that can be used to guide analytical studies of large-scale multiprocessors, as in [Agarwal, 1992]. Additional statistics output from the trace analyzer indicate the average remote memory latencies on a per-page basis (which allows us to determine the distribution of memory latencies across pages, and the identities of hot pages) and the average queue lengths (when memory requests are allowed to queue at the memory modules).

We studied the programs shown in the left-hand column of Table 1. The first three are well-known applications used in many previous studies. `Mp3d` and `barnes-hut` are programs from the SPLASH suite [Singh *et al.*, 1992]. `Mp3d` is a wind-tunnel airflow simulation. `Barnes-hut` is an N-body application that simulates the evolution of a system under the influence of gravitational forces. `Mp3d2` is a version of `mp3d` restructured for better cache behavior, as described in [Cheriton *et al.*, 1991]. In our experiments, both versions of mp3d were run with 30000 particles for 100 time steps. `Barnes-hut` was run with a system of 4K bodies.

The last six programs in Table 1 represent computational kernels similar to those present in many programs. `Bmatmult` is blocked matrix multiplication. `Mgauss` is medium-grained Gaussian elimination, in which each process eliminates one row of the matrix. `Sgauss` is fine-grained Gaussian elimination, in which each process eliminates a single element of the matrix. `Matinv` is matrix inversion. `Tclosure` is transitive closure, in which each process operates on a set of rows of an adjacency matrix. `All-pairs` computes the all-pairs shortest paths of a graph, using a parallelization of Warshall-Floyd's algorithm. Our example kernels, except for `bmatmult`, all require that all processors simultaneously access data (a row of the main memory) that was recently modified by a single processor. This form of producer/consumers relationship can lead to memory contention if the data that must be accessed by all processors resides in a single memory module, as is usually the case. All these kernels operate on 512×512 matrices, except for `bmatmult` and `all-pairs`, which operate on 400×400 matrices.

3.2 Observed Contention

We evaluated a set of applications with sufficiently different behavior to establish that contention can be a common problem in large-scale machines. Table 1 presents the average memory latency for references that missed in the cache for different numbers of processors. The table shows that all the applications studied exhibited some degree of contention for memory, and that in the vast majority of cases, average latency increases significantly with the number of processors.[1]

The most important metric provided by our simulations is the increase in execution time due to contention. Table 2 shows that the amount of time spent contending for memory, especially on large machines, can significantly increase the running time for most applications.

[1]Recall that remote memory latency in the absence of contention is 82 cycles.

3 Memory Contention in Real Programs

3.1 Experimental Method

Since we are interested in studying memory contention in the truly large-scale shared-memory multiprocessors currently under development, direct experimentation is not an available option. Therefore, we simulate a large-scale direct-connected multiprocessor (up to 256 processors) executing our example applications. Our simulations consist of two distinct steps: a trace collection process, and a trace analysis process. The trace-collection step uses Tango [Davis *et al.*, 1991] to simulate a multiprocessor with (infinite) coherent caches. The traces generated by Tango contain the data references that missed in the local cache of each processor, and all synchronization events.

Our analyzer process takes as input an address trace produced by Tango, and simulates execution of the references in the trace on a distributed-shared-memory multiprocessor. The analyzer assigns each reference to the appropriate processor at the appropriate time by tracking the delay induced by previous references, combined with the time spent executing instructions on the processor. The analyzer respects the synchronization behavior of an application as represented by the synchronization events contained in the trace. Synchronization events are not allowed to cause contention in our model, although they are critical in maintaining the relative timing of events during trace analysis.

In our machine model, a memory module can process only one request at a time. Requests arriving when the module is busy are rejected and must be reissued. Our analyzer measures contention for memory at the page level; thus each 4KB page is treated as a separate memory module to which requests may be directed. We treat each page as a separate memory module so as to simulate an ideal page placement policy in which contention caused by simultaneous accesses to multiple pages does not occur. One consequence of this assumption is that the number of memory modules in the system is dependent on the size of the problem and not on the number of processors in use. As a result, our estimates of memory contention are optimistic, in that we measure the contention inherent in an application, independent of the placement of pages in memory modules.

Our simulations assume a cache line size of 16 bytes with four lines fetched on a miss, a fixed network latency of 36 processor cycles, and local memory latency of 10 processor cycles. In the absence of contention, a remote memory request requires a request message, a reply message, and memory service time, or 82 cycles total. Each request rejected due to contention suffers a 72 cycle penalty, corresponding to an immediate re-issue of the request.

Our simulation assumptions are optimistic, in that we chose values for the simulation parameters that are likely to result in less contention than would exist in the machines of the near future. For example, we chose to use 64-byte fetches on misses, which is larger than the fetching units used in both DASH and Alewife (but smaller than the fetching units used in the Kendall Square KSR1). In addition, we chose a memory latency of 10 processor cycles per fetching unit, and a network latency of 36 processor cycles, both of which are quite optimistic. We would expect the combination of fewer misses (due to larger fetching units, and assuming no fine-grain sharing in our applications) and faster memory service time to result in less memory contention than would otherwise occur in DASH and Alewife. Our infinite cache assumption means that we only measure the effect of invalidation-related misses, and ignore capacity misses. Our assumption that network latency is fixed (i.e., there is no network contention) allows us to isolate the effects of memory contention

1988; Ho and Eager, 1989]. Those studies focused on eliminating tree saturation, and used synthetic applications for experiments. Our experiments show significant performance degradation without including network congestion effects, and are based on real applications.

Glenn *et al.* [1991] studied hot-spot effects in synthetic applications in the absence of network congestion and processor caches. They divided hot spots into three categories: 1) a read-only memory location with a large number of readers; 2) synchronized access to memory modules due to related strides; and 3) hot spots caused by synchronization references. This classification does not apply directly to machines with processor caches, which do not exhibit type 1 hot spots, and may not exhibit type 3 hot spots given an appropriate implementation of synchronization [Mellor-Crummey and Scott, 1991]. In addition, type 2 hot spots are primarily an issue on machines that use low-order interleaving of addresses. In this paper we focus on a fourth type of hot spot, synchronized access to read-mostly data. We show that this type of contention can be a major source of overhead in shared-memory multiprocessors, and that it is often caused by synchronized access to large data structures, as opposed to single addresses.

In their study of a single application, Glenn and Pryor [1991] found hot spots to be a significant problem. They noted that each time their application was moved to an increasing number of processors, new hot spots developed. We show that this effect is common, occurring consistently in the applications we studied.

A number of other studies have considered memory reference patterns of applications running on multiprocessors. However, none to date has measured the amount of hot spot contention in typical applications. Agarwal and Gupta [1988] did not record exact time in their traces, so precise quantification of memory contention was not possible. Darema-Rogers *et al.* [1987] found significant burstiness in the memory access rates of parallel programs, which indicates the likelihood of memory contention, but they did not measure or predict contention effects.

In general, memory contention results from accesses to data by multiple processors in a coordinated way; this effect is not a simple function of temporal, spatial, or processor locality, which are the metrics emphasized by previous trace-based studies. In [Bianchini *et al.*, 1993], we address this problem at the application (compiler) level through a technique called block-column allocation. Although successful at alleviating contention, block-column allocation is only effective for certain types of matrix computations. This paper explores more widely applicable solutions that do not depend on sophisticated compilers or programmers.

We propose a general solution to hot-spot contention consisting of simple hardware support for data replication. Our strategy combines ideas from software combining trees [Yew *et al.*, 1987] and eager sharing (e.g., [Wittie and Maples, 1989]). We describe and evaluate the implementation of our strategy for use in distributed directory-based cache-coherency schemes [Chaiken *et al.*, 1990; Lenoski *et al.*, 1990].

The quantification of memory contention we present in this paper has implications for analytical models of multiprocessor performance, such as [Agarwal, 1992]. In particular, the round-trip latency of a non-local memory reference is an important parameter when predicting the number of processor contexts that will be most profitable in multi-threaded multiprocessors. We show that non-local memory references are often considerably more expensive when memory contention effects are included.

1 Introduction

Effective use of large-scale multiprocessors requires the elimination of all bottlenecks that reduce processor utilization. A common assumption in analytical models used to guide multiprocessor design is that applications generate a uniform memory referencing pattern, resulting in a uniform utilization of the memory modules in the system. This assumption is overly optimistic, because nonuniformity in reference patterns will lower throughput as processors contend for the bandwidth of individual memory modules. This effect has been termed hot-spot contention [Pfister and Norton, 1985].

In this paper we show that significant memory contention is common in typical parallel applications, when those programs are run on large-scale machines. In addition, we show that hot spot contention increases in degree and extent as programs are moved from smaller to larger machines. Whereas previous hot-spot studies have focused primarily on their effects on multistage interconnection networks, we show that hot spots will have serious performance implications in direct-connected, distributed-shared-memory machines such as the Stanford DASH [Lenoski *et al.*, 1992] and the MIT Alewife [Agarwal *et al.*, 1992]. We relate our hot spot studies to previous analytical and simulation studies of hot spots, and previous work on memory reference patterns, in Section 2.

Our approach to hot spot evaluation differs from previous approaches, which did not use reference traces from real programs. We generate reference traces from a number of typical parallel programs using the Tango simulator [Davis *et al.*, 1991], and then simulate the contention effects of those traces when accessing a distributed memory. This allows us to explicitly measure the delay experienced by requests to each memory module, and provides insight into the distribution of hot spots in memory.

Our simulator, and the quantification of hot-spot contention in the programs we studied, is presented in Section 3. We show that memory contention can drastically increase running time, even for programs with good locality of reference and under optimistic simulation parameters. We show that hot spots are often spread throughout memory; in some applications over 90% of the shared data pages exhibit contention. We also show that the selective removal of individual hot spots can cause other hot spots to worsen. The widespread and persistent nature of memory contention in our applications suggests that it should be addressed at the architectural level.

In section 4 we evaluate two mechanisms for hot spot removal: queueing at the memory module, and hardware-supported data replication. We show that queueing at the memory module is reasonably effective, but that very long queues are needed. As an alternative, we propose a combination of eager sharing and combining trees, applied to distributed directory-based coherence protocols. We evaluate the performance of our protocol, and show that it virtually eliminates the effects of hot spot contention. We conclude, in section 5, that hardware-supported replication is an attractive way to deal with memory contention, especially when used in conjunction with application restructuring techniques designed to eliminate the few extreme cases of hot spots.

2 Related Work

Previous evaluations of the effects of hot spots have focused primarily on their impact in multistage interconnection networks (MINs) [Pfister and Norton, 1985; Thomas, 1986; Patel and Harrison,

Memory Contention in Scalable Cache-Coherent Multiprocessors

Ricardo Bianchini, Mark E. Crovella,
Leonidas Kontothanassis and Thomas J. LeBlanc

{ricardo,crovella,kthanasi,leblanc}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 448

April 1993

Abstract

Effective use of large-scale multiprocessors requires the elimination of all bottlenecks that reduce processor utilization. One such bottleneck is memory contention. In this paper we show that memory contention occurs in many parallel applications, when those applications are run on large-scale shared-memory multiprocessors. In our simulations of several parallel applications on a large-scale machine, we observed that some applications exhibit near-perfect speedup on hundreds of processors when the effect of memory contention is ignored, and exhibit no speedup at all when memory contention is considered. As the number of processors is increased, many applications exhibit an increase in both the number of hot spots and in the degree of contention for each hot spot. In addition, we observed that hot spots are spread throughout memory for some applications, and that eliminating hot spots on an individual basis can cause other hot spots to worsen. These observations suggest that modern multiprocessors require some mechanism to alleviate hot-spot contention.

We evaluate the effectiveness of two different mechanisms for dealing with hot-spot contention in direct-connected, distributed-shared-memory multiprocessors: queueing requests at the memory module, which allows a memory module to be more highly utilized during periods of contention, and increasing the effective bandwidth to memory by having the coherency protocol distribute the hot data to multiple memory modules. We show that queueing requires long queues at each memory module, and does not perform as well as our proposed coherency protocol, which essentially eliminates memory contention in the applications we consider.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships. Mark Crovella is supported by a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.