Memory Servers for Multicomputers*

Liviu Iftode Kai Li Karin Petersen

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

In this paper, we investigate a virtual memory management technique for multicomputers called memory servers*. The memory server model extends the memory hierarchy of multicomputers by introducing a remote memory server layer. Memory servers are multicomputer nodes whose memory is used for fast backing storage and logically lie between the local physical memory and disks. The paper presents the model, describes how the model supports sequential programs, message-passing programs and shared virtual memory systems, discusses several design issues, and shows preliminary results of a prototype implementation on an Intel iPSC/860.*

Keywords: distributed memory, memory server, multicomputer, memory management unit, virtual memory.

Introduction

Multicomputers can provide very high processor performance and tremendous total storage capacity at relatively low costs. Scalable massively parallel multicomputers are attractive architectures because they take advantage of the performance curves of both high-performance microprocessors and high-bandwidth interconnection networks. Today's multicomputers not only deliver peak performance higher than any traditional vector supercomputer, but are also an order of magnitude less expensive.

However, it is difficult to exploit the available peak performance and capacity on existing multicomputer systems [35, 16, 29, 26]. This difficulty stems primarily from the need to parallelize programs at a level of granularity that matches parameters of the target architecture such as message latencies, number of processors, and available memory on each node. In fact, many programs have to be parallelized or simply decomposed into small pieces in order for the data structures of each collaborating process to fit into the physical memory available on each node. For example, the Intel touchstone Delta multicomputer has 513 computational nodes, each of which has 16 Mbytes of memory. Although the entire machine has over 8 Gbytes of memory, it is not possible to run a parallel program that requires more than 16 Mbytes of memory for one of its processes, nor to run a sequential program with data structures larger than 16 Mbytes.

Adding more physical memory on each node may not be a viable solution to avoid memory limitations, because the size of each node is usually restricted to allow low cost, low-latency communication, and low power dissipation. Another solution to avoid memory limitations on multicomputers is to employ traditional virtual memory management on each node. Virtual memory will allow a program to run on a multicomputer even when the memory requirements on a node exceed the physical memory available on that node. Unfortunately, paging to disk is too slow for multicomputer applications. The cost of a page transfer between memory and disk is on the order of a million CPU cycles. In addition, there are usually a limited number of nodes in a multicomputer that connect to disks.

In this paper, we investigate a virtual memory management technique for multicomputers called *memory servers*. The memory server model extends the memory hierarchy of multicomputers by introducing a remote memory server layer. Memory servers lie between the local physical memory and disks. When a node runs out of its local physical memory page frames, it will use a set of memory servers as fast backing stores to pageout frames chosen for replacement. In turn, when memory servers run out physical page

*This work is supported in part by NSF Grant CCR-9020893, DARPA and ONR under contracts N00014-91-J-4039, and Intel Supercomputer Systems Division.

frames, they will use disks as backing stores. In this case the memory servers will be utilized as caches between local memory and disks.

The essential idea of the memory server approach is to take advantage of the tremendous memory resources and fast interprocessor data transmission capabilities of massively parallel multicomputers. Current multicomputer routers are capable of moving a 4 Kbyte page between two nodes three orders of magnitude faster than moving a page between memory and disk[29]. This performance gap is expected to widen in the future.

We have designed a memory server mechanism that supports sequential and message-passing programs, as well as shared virtual memory implementations. This mechanism allows programs to utilize the entire physical memory available on a multicomputer. We have implemented our first prototype on the Intel iPSC/860 on top of the NX/2 operating system, performed some preliminary experiments, and derived the performance implications on advanced multicomputers such as the Intel Touchstone Delta and Paragon.

1 The Memory Server Model

The architectural framework we assume for the memory server model is a large scale distributed memory multicomputer. Each node in the system consists of a processor, a local memory unit, and a network interface. The nodes are interconnected through a scalable, high bandwidth routing network. Only some nodes in the system contain disks and other I/O devices. The Intel iPSC/2, iPSC/860, Touchstone Delta, Paragon, the TMC CM-5, the nCUBE series, J-Machine, and Mosaic are examples of such an architecture [16, 29, 35, 26, 9, 31].

The memory hierarchy of such a multicomputer architecture consists of: local caches, local memories, and disks on the I/O nodes. Traditional virtual memory implements a mapping between local memories and disks. The **memory server** model shown in 1 extends this memory hierarchy by adding remote memory servers between the local memories and disks.

In the memory server model, multicomputer nodes are divided into two categories: *computation nodes* and *memory server* nodes. A computation node performs computation and accesses its memory space to do so, whereas a memory server node acts as fast backing stores for the computation nodes. To take advantage of locality of reference, a computation node will use its local physical memory as much as possible. However, when it exhausts its local memory, it

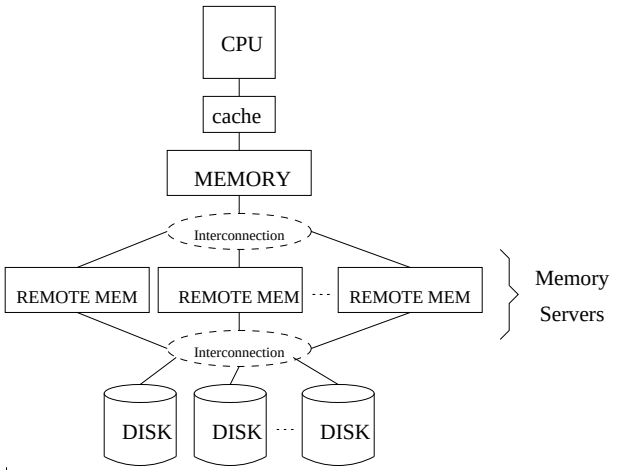

Figure 1: Multicomputer memory hierarchy.

will free page frames by using memory server nodes for their backing storage. Pages stored on memory server nodes are brought back into local memory by demand. Computation nodes that require memory servers for backing storage are called *clients*. The memory server model can support different programming paradigms: sequential programs, message-passing programs, shared virtual memory systems, and other distributed shared memory implementations.

Figure 2 shows an example of a very simple case of the memory server model: demand paging. Client node P_0 will fault on accesses to the virtual pages that are not resident in its local physical memory. In the example P_0 faults on a page that is resident in P_n' s physical memory. It will therefore request the page from memory server P_n, which then has locate the page, and transfer its contents back to P_0. At that point P_0 will load the page into a free frame, and return from the fault.

Aside from demand paging, there are several other cases of client-server interactions: (1) the client needs to write a dirty page to its backing store in order to free space for an incoming page, for this it simply sends a page-out request to the corresponding memory server; (2) a memory server has exhausted its physical memory and to make room for other pages it moves some pages to disk backing stores; (3) the server node does not have the requested page resident in memory, it then retrieves the page from the disk backing store, loads it into its local memory, and forwards it to the client node.

Integrating the memory server model with a shared virtual memory system requires page replacement

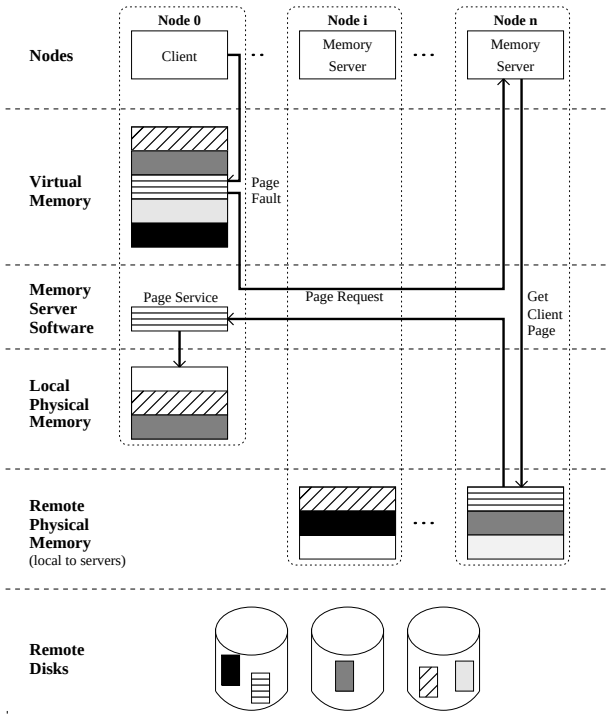

Figure 2: Client and Memory Server Interaction.

strategies different from the simple demand paging case. A page can have multiple copies and the shared virtual memory system maintains the memory consistency among those copies.

The memory server model extends the multicomputer memory hierarchy by using remote memory servers as distributed caches for disk backing stores. It provides an efficient paging mechanism that allows programs and systems to share physical memory resources. Similar to traditional virtual memory, the extended memory hierarchy allows each computation node to use address spaces larger than its local physical memory, but its interaction with the memory servers is much faster than with disk backing stores. The main difference between the memory server model for multicomputers and traditional virtual memory is that the memory server model makes the entire physical memory of a multicomputer available and accessible to any application executed on the system, regardless of its degree of parallelism. The memory server model not only supports sequential and message-passing programs with large data structures, but also allows shared virtual memory or other distributed shared memory implementations to share physical memory resources.

2 Design Issues

There are many issues and trade-offs in designing and implementing a memory server mechanism for multicomputers. This section discusses design issues related to partitioning computation and memory server nodes, page replacement strategies for computation and server nodes, caching and prefetching of pages, and kernel and user space implementations.

Computation vs. Memory Server Nodes

As defined in the previous section, computation nodes are those on which application program processes execute. Memory server nodes serve as fast backing storage for computation node memory pages and can be viewed as distributed caches for disk backing stores.

The node types, computation or memory server, can be considered a functional attribute for a multicomputer node. The simplest node management approach is to have a dedicated functional attribute for each node. For example, a computation node is not a server node. A memory server node does not perform computation and it serves one or more computation nodes. A computation node will have no memory server if it has enough physical memory for its processes to run. Only when a computation node exhausts its local physical memory, it becomes the client of memory server nodes. In this approach, it is clear that memory server nodes should be selected among idle nodes, which have available memory and processor resources.

Another approach is to allow a node to be both a computation node and memory server. A computation node can become a memory server if it can provide enough free page frames on its local memory as well as a reasonable amount of CPU cycles to handle remote paging requests for clients. In this case, a computation node shares its physical memory with other computation nodes. On the other hand, a memory server node may become a computation node if it has enough CPU cycle resources and memory space to handle computation.

An important design issue is how to decide which nodes of the multicomputer will be doing computation or memory server work. Obviously, this can be done by the operating system, by applications, or a combination of the two. Assigning nodes through the operating system is the most transparent method. The operating system determines which nodes are used as computation nodes and which ones are memory server nodes. This method can be either static or

dynamic. A static system based partitioning scheme assigns computation and memory server nodes statically and the assignment does not change. With this approach, the system needs to allocate computation nodes and memory server nodes for each application. The main disadvantage is that it is difficult to find an optimal allocation strategy that minimizes communication overhead and matches the application process communication patterns. The static assignment can also waste processor and memory resources.

A dynamic operating system based approach decides which nodes to use for computation and memory servers when the system is allocating nodes for applications. The system views the functionality of computation and memory server as dynamically established attributes. A node can switch from one function to another depending upon the execution context.

The application based approach lets users decide about the functional attributes of nodes. Users are responsible for requesting enough nodes from the operating system for their computation and memory services, and then deciding which nodes perform computation or memory service.

The combined approach is more flexible. The idea is to let the operating system and users split the responsibility of node assignments. One way is to let users request only computation nodes from the operating system and the system supply memory server nodes from a pool of nodes as needed. Another way is to require users to inform the operating system about the application's memory requirements and to have the operating system reserve memory servers for each application.

Page Mappings and Replacement

There are two kinds of mappings the operating system needs to establish and maintain: `<local page>` → `<server node, page>`, and `<local page>` → `<I/O node, disk block>`. The latter can be maintained either on the computation or the server node. The main design goal is to use a method that is efficient and simple, and consumes a minimal amount of memory space.

The mappings for a page are established on demand upon a page fault or a page frame replacement event. The mappings can be removed only when the page is deallocated. One can apply restrictions on the mapping function domain to make implementation simple. For example, if the memory server mechanism supports only sequential and message-passing programs, the system can impose a restriction on the mapping to be one-to-one. In other words, each page only has at most one backing store copy and no page is replicated at more than one computation node.

Shared virtual memory (SVM) implementations require their memory server mechanisms to have less restricted mappings because a page may have duplicated copies. Although allowing duplicated copies requires the system to maintain memory coherence, it can make page replacement more efficient [22]. For example, page replacement of a duplicated copy requires no page transfer. Also, by enforcing duplicate copies the fault tolerance of the system can be improved.

With some modifications, page replacement techniques for traditional virtual memory systems can be used for the memory server mechanism. Similar to traditional virtual memory, the memory server mechanism also has two kinds of virtual pages: valid and invalid. In addition, a programming paradigm such as SVM has other attributes for each page, for example, the accessibility and replication status of a shared page. An SVM page can be no-access, read-copy, read-owner, or writable. A no-access page is a page whose access bits in the page table entry have been set to nil by the SVM memory coherence algorithm. Both read-copy and read-owner pages are read-only. A read-owner page is owned by the current processor and a read-copy page is not. A writable page is owned by the processor and it allows both read or write accesses.

To minimize the system overhead for page replacement, physical page frames of different types of SVM pages should be treated differently. Physical page frames whose virtual pages are no-access should be reclaimed with higher priority than any other type of shared page, because a processor that issues a memory reference to a no-access page will page fault regardless of the page being mapped in physical memory or not. Physical page frames whose SVM pages are read-copy should be reclaimed next, because there is at least another copy of the page (namely, read-owner) in the system. Physical page frames whose SVM pages are read-owner should be chosen over those SVM pages that are writable because it is less expensive to relinquish the ownership of a page than to transfer a page back to disk.

Two parameters are important for page frame reclamation: page type and last-reference time. It is a bad strategy to reclaim all page frames of one type before starting to reclaim pages of another type, since recently referenced page frames may be reaccessed. Using only the last-reference time as the replacement criterion is not a good strategy either, because page frames whose pages are no-access may have been referenced recently. Since parallel programs exhibit a

high degree of spatial locality of reference [12], a better strategy is to combine both parameters to decide the priorities in which page frames are reclaimed. A practical approach is to use a list data structure to represent an ordered set for each type of page frames. Each set is ordered by its last reference time, and, each set has thresholds to control when its page frames should be reclaimed.

Caching and Prefetching

The memory server model provides a convenient platform for caching and prefetching pages between the remote memory servers and disks. In fact, memory servers are distributed caches between the local memories and disk backing stores.

To implement a memory server mechanism, one has to make a design decision on when pages should be written to disks. The simplest way is to postpone this operation as long as possible. Pages will be written to disks only when the number of available page frames on a memory server is below a defined threshold. The main advantages of this approach are its simplicity and that it imposes little disk paging traffic and requires minimal disk space for backing stores.

Another alternative is to move pages from memory servers to disks as early as possible. Every page that is transferred from a computation node to a memory server is replicated on a disk I/O node immediately. The disk I/O node writes replicated pages to its disks in batches. If the target multicomputer has adequate network bandwidth for I/O nodes and an efficient multicast mechanism, this approach can be very effective. This method can eliminate the latency imposed by "demand paging" on memory server nodes, and it also provides fault tolerance for memory server node failures. The disadvantage of this approach is that it needs more disk space and generates more disk I/O traffic than the delayed approach.

Memory server nodes can prefetch pages for computation nodes using different strategies. Unlike in traditional virtual memory systems for uniprocessors, prefetching on memory servers is practically "free" of cost since the server's CPU cycles and disk I/O operations can overlap with the program executions on the computation nodes. Because there is a large performance gap between routing network bandwidth and disk bandwidth, prefetching to memory servers can be effective in a system for which the prefetching strategies match the memory access patterns on the computation nodes.

3 Implementation

We have designed and implemented a memory server mechanism prototype on the Intel iPSC/860 multicomputer. The main goal of the prototype is to validate our ideas, investigate design tradeoffs, perform measurements of applications, and predict the performance for systems with different architectural parameters. The reason to choose the iPSC/860 as our target architecture is its availability and its similarity to the Intel Touchstone Delta and Paragon machines, which will be available to us in the near future. Although the network bandwidth of the iPSC/860 is one order of magnitude lower than that of Delta and two orders of magnitude lower than that of Paragon, our implementation can be used to predict the performance implications of the memory server mechanism.

The iPSC/860 hypercube we have available for our implementation has a total of eight nodes, four i860 and four i386 nodes, each with 8 Mbytes of physical memory. Each of the i386 nodes is configured as an I/O node and is connected to an 1 Gbyte SCSI disk. The disks on the I/O nodes are accessed through the concurrent file system (CFS) [28]. CFS reserves about 5 Mbytes of physical memory on the i386 nodes for file caching. The bandwidth of the routing network on the iPSC/860 is 22 Mbits/second.

Our memory server prototype is implemented in the context of the NX/2 operating system, a small kernel mainly supporting protected message passing primitives for parallel programming. For simplicity, the client implementation resides completely within the kernel. The server is implemented in user space. We also implemented a traditional virtual memory system using the concurrent files system (CFS) as backing storage. We use the host node of the iPSC/860 to register the information about memory server node assignments.

The client implementation is composed of modules for physical memory management, page fault handling, and page replacement. The physical memory management implements memory allocation and deallocation. The page fault handler is triggered by a memory reference to an invalid page and may invoke page replacement. Paging between client and server nodes is implemented using a sequence of short control messages for the protocol and a long 4K byte message for page transfer.

We implemented an approximate LRU algorithm for page replacement. The mapping between a swapped page and its server node is established only after a memory server has accepted a page service request. Since page replacement is modularized, it is

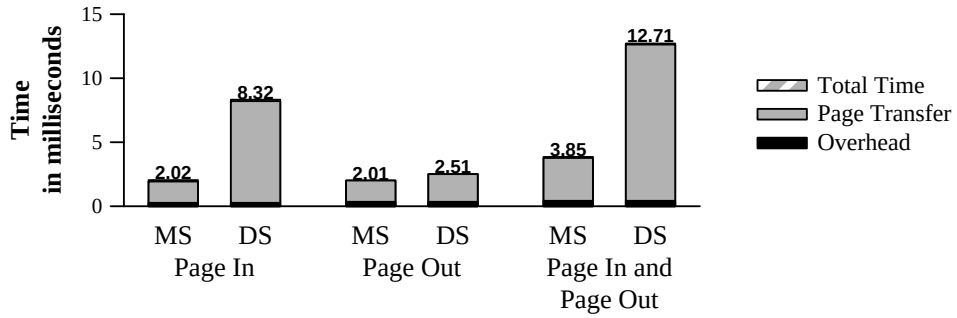

Figure 3: Page Fault times.

	Time in ms	
	Memory Server	**I/O Node**
Context Switch	.024	.024
Control Message	.109	.109
Page Transfer	1.746	2.23-9.27 (*)

* Depending on File Caching at the I/O nodes

Table 1: Timings for Page Fault Operations.

easy to implement other replacement policies. We are in the progress of incorporating the memory server model into our shared virtual memory implementation.

The memory server implementation is straightforward. We used simple data structures, such as hash tables, to maintain page mappings. Pages are moved to disk only when the amount of free physical page frames on a memory server is below its threshold.

4 Preliminary Performance Evaluation

We measured the cost of handling a page fault, and performed preliminary measurements with our memory server mechanism on the iPSC/860.

Figure 3 shows the cost breakdown for handling a page fault served by a memory server node or by an I/O server using CFS. In our experiments the memory servers always run on i860 nodes. The results in Figure 3 were generated by sequentially touching each page in a 16 Mbytes address space and calculating the ratio of total execution times and number of generated page faults. For both cases, memory server (MS) and disk I/O server (DS), most time is spent locating the page at the server and transferring it to the client. The remaining operations, selecting a page for replacement, cache flushing and context switching, actually have a negligible effect on the page fault time. On iPSC/860, serving a page fault from a memory server is three to four times faster than serving it from an I/O node. This difference is much less if the server only receives page-out requests, since the I/O node can cache write requests in its local memory before utilizing the disk. Table 1 shows the cost of individual operations executed during a page fault.

We used three sequential applications to get a preliminary evaluation of the benefits of paging to memory servers rather than using the traditional virtual memory mechanism, which pages to disk I/O nodes. MMUL is a blocked matrix multiply program [20]. The input to MMUL are two $800x800$ floating point matrices and it generates another $800x800$ result matrix. The computation is blocked, for locality purposes, to consecutively generate each of the four $400x400$ result submatrices. GE generates the equivalent upper triangular matrix for an $1440x1440$ floating point input matrix. In order to increase locality of reference, GE's computation is blocked in three $480x1440$ submatrices. The last application is SORT, which generates the sorted version of 1 million randomly generated floating point numbers. The application is a variant of merge sort. In a first pass SORT generates three sorted sublocks of the initial set of numbers by further partitioning the sublocks in 100 smaller blocks, sorting the small blocks and then merging them. During a second pass, SORT merges the three sorted sublocks to generate the sorted set of

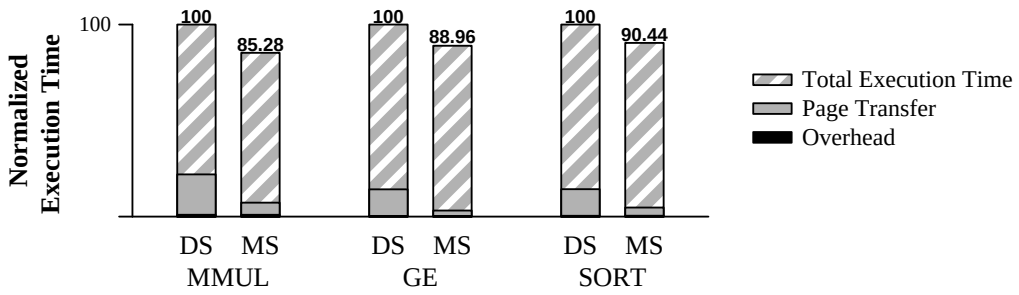

Figure 4: Memory Server vs. Disk Paging.

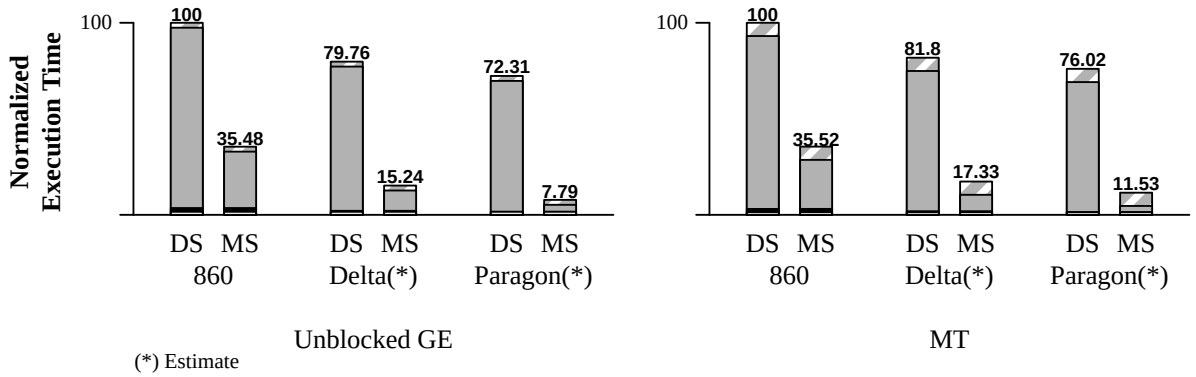

Figure 5: Memory Server performance.

all numbers.

In all examples, we used one i860 node for computation and three i860 nodes as memory servers. Each memory server provides 7.1 Mbytes of physical memory to be used as fast backing storage. The execution time for each application with the traditional virtual memory system implementation was 235 milliseconds, 735 milliseconds, and 455 milliseconds for MMUL, GE, and SORT respectively. The measured execution time does not include data initialization.

In order to compare the memory server approach with the traditional virtual memory approach, we normalized the execution times. In each graph, the execution time for the disk I/O server based traditional virtual memory implementation represents 100%, and the execution time of the memory server based approach is normalized to it.

Figure 4 shows the difference in performance of the applications using memory servers or I/O node based virtual memory. All applications perform better when using memory servers for fast backing storage. The performance gap differs according to the relative impact of paging activity on the applications execution time. For example, MMUL, spends 21% of its time handling faults when paging to the disk I/O nodes, and only 8% when paging to memory servers. SORT on the other hand only spends 14% paging to the I/O nodes, and therefore, using memory servers as fast backing storage improves performance by only 10%.

In our multicomputer, the number of computation nodes and I/O nodes is the same, making the file caches on the I/O nodes effective for moderate file sizes. For large scale multicomputers, however, the ratio between I/O nodes and computation nodes will decrease significantly which will decrease the effectiveness of the I/O node disk caches. When the disk caches become ineffective, the difference in performance of the memory server approach with respect to a traditional virtual memory implementation will be more significant than the results shown in Figure 4.

In the previous examples, we have used blocked algorithms for all applications. In practice, automatic blocking transformation is quite limited and manual transformation is is difficult. In order to study the effect of the memory server approach on applications without blocking, we ran an unblocked version of GE on the same data size. Figure 5 shows the results of this experiment. In this case, GE spends 93% of its execution time on page fault handling using the I/O nodes as backing stores. The memory server approach improves the performance of the unblocked version of GE by a factor of three.

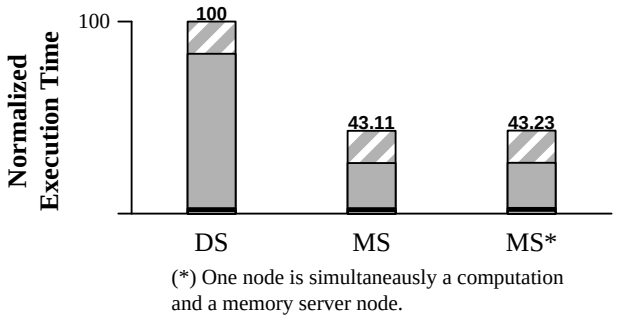

Figure 6: Computation nodes used also as Memory Servers.

Applications with large data structures and low reference locality benefit greatly from the memory server mechanism. Such an example is matrix transpose (MT) which is used in graphics applications very frequently. Figure 5 also shows the results for MT, a program executing the transposition of an $1440x1440$ floating point matrix. To increase locality of reference, the computation is blocked by transposing one pair of $480x480$ submatrices at the time. MT with traditional virtual memory spends most of its execution time (95%) paging to the I/O servers. By using memory servers as fast backing stores the execution time is reduced by a factor of 3 on the iPSC/860.

We also estimated the impact of architectures with faster interconnection networks, such as the Intel Touchstone Delta and Paragon. Our estimates were based on the following assumptions: the processor speed remains the same, the network bandwidth of Delta and Paragon is 9.6 MB/sec and 200 MB/sec compared with the iPSC/860's 2.8 MB/sec, and all architectures spend 200 microseconds setting up a page transfer. The performance gap between routing networks and disk I/O is critical for applications without blocked algorithms or without much computation. In our preliminary experiments, the high bandwidth network on Paragon improve the performance of unblocked GE and blocked MT by an order of magnitude.

For parallel applications whose threads have unbalanced memory requirements, some computation nodes can also serve as memory server nodes. Figure 6 shows the results of running GE on two processors. One node executes GE on $\frac{3}{4}$ parts of the matrix, and the other one the rest. The graph shows the results for running this experiment using disk as backing storage, then using a set of memory servers nodes disjoint from the set of nodes used for computation, and finally using the node with less computation and memory load to be one of the memory servers for the other computation node. The memory server approach clearly outperforms using the traditional disk server virtual memory technique. Furthermore, the performance difference of both memory server executions is insignificant, showing that lightly loaded computation nodes can also be memory servers efficiently.

5 Related Work

Memory hierarchies and virtual memory systems have been studied in great detail. Earlier work focused on uniprocessor architectures, with emphasis on multilevel memory hierarchies [2, 25, 32, 23], memory access patterns [1], virtual memory systems [10, 11], memory and disk caches [33, 34].

Most related work on memory management for parallel architectures includes page placement strategies for distributed shared memory architectures, and shared virtual memory systems. Non uniform memory access (NUMA) architectures allow processors to access data in either local or remote memory directly. The cost of a memory access to local memory is significantly lower than accessing remote memory. In order to use a NUMA architecture efficiently data should be located close to the threads that use it. Research in this area has shown that dynamic page placement is an effective solution to the problem [4, 5, 15, 6, 18, 19]. However, on message-based multicomputers these techniques do not apply because there is no hardware support for direct access to remote memory.

The shared virtual memory (SVM) approach provides various parallel programming paradigms on distributed memory architectures that do not provide physically shared memory [21, 24, 7]. A recent effort in the area of shared virtual memory and distributed shared memory research is to relax the memory consistency models used by these systems [17, 3]. As mentioned in previous sections, the memory server model

can be incorporated into an SVM system to exploit the utilization of available physical memory.

Researchers in the local area network community haven taken a similar approach to the memory server model [8, 13, 30], using the memory available on idle workstations as backing stores. Although conceptually their approach is similar to that in this paper, there are several differences. Mainly, the memory server mechanism for multicomputers presented in this paper supports multiple programming models: sequential, message-passing and shared virtual memory. Supporting a shared virtual memory system requires very different page replacement and caching strategies. In addition, the architectural differences between a massively parallel multicomputer and a network of workstations impose different design tradeoffs.

Several research efforts have pointed out that memory bandwidth is crucial for performance of both uniprocessor and parallel architectures [27, 14]. The memory server mechanism attacks the memory latency problem by utilizing available physical memory to reduce accesses to slow disks.

6 Conclusions

The memory server model is a simple virtual memory management technique for multicomputers, yet it is a practical technique for utilizing the tremendous memory resources available on multicomputers. This paper shows that the memory server mechanism not only supports large sequential and message-passing programs, but can also be integrated easily with shared virtual memory systems and other distributed shared memory implementations.

The memory server model takes advantage of both fast routing networks and available memory resources in multicomputers. A page transfer on the state-of-the-art multicomputer is three orders of magnitude faster than a page transfer between memory and disk. The performance advantage of the memory server mechanism over the traditional virtual memory management will become even more significant as the performance gap between routing networks and secondary storage widens.

Our limited expperiments show that for sequential applications with large data structures, the memory server mechanism can outperform the traditional virtual memory management by a large factor. On the other hand, the experiments reported in this paper are from a small iPSC/860, with four computational nodes and four disk I/O nodes. As the ratio of computational nodes to disk I/O nodes increases, the disk I/O can become a bottleneck for paging. In this case, we expect that the advantage of the memory server mechanism over the traditional virtual memory management will be more pronounced.

Our prototype implementation currently runs only on the iPSC/860 which has a slow routing network, two orders of magnitude slower than the current generation routers in the Intel Paragon. The current implementation does not include the paging strategies for shared virtual memory. We are planning to integrate the memory server mechanism into our shared virtual memory system and port our implementation to the Touchstone Delta and Paragon in the future.

Acknowledgements

We would like to thank Brian Marsh and Brain Bershad for their helpful comments on the early draft of this paper, and James Plank for his `jgraph` tool which was used to produce the bar graphs in this paper.

References

[1] B.Brawn and F.G. Gustavson. Program Behavior in A Paging Environment. In *1968 AFIPS Proceeding of the Fall Joint Computing Conference*, pages 1019–1032, 1968.

[2] L.A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the IEEE COMPCON'93 Conference*, February 1993.

[4] D. Black, A. Gupta, and W-D Weber. Competitive Management of Distributed Shared Memory. In *Sprint COMPCON 89 Digest of Papers*, pages 184–190, February 1989.

[5] W. Bolosky, M. Scott, and R. Fitzgerald. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 19–31, December 1989.

[6] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, and A. Cox. NUMA Policies and their Relationship to Memory Architecture. In *Proceedings of the 4th International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[7] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium*

on Operating System Principles, pages 152–163, October 1991.

[8] Douglas Comer and James Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the 1990 USENIX Summer Conference*, pages 127–135, June 1990.

[9] William J. Dally. Performance Analysis of k-ary $n-$-cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[10] Peter J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[11] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[12] S.J. Eggers and R.H. Katz. A Characterization of Sharing in Parallel Programs and Its Applications to Coherence Protocol Evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 373–383, June 1988.

[13] E. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical Report 91-03-09, University of Washington, March 1991.

[14] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254–263, May 1991.

[15] M. Holliday. Reference History, Page Size, and Migration Deamons in Local/Remote Architectures. In *Proceedings of the 3rd International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 104–112, April 1989.

[16] The iPSC/860. Technical Reference Manual, 1991.

[17] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[18] R. P. LaRowe Jr., J. T. Wilkes, and C. S. Ellis. Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor. In *Proceedings of the 1991 Symposium on the Principles and Practice of Parallel Programming*, pages 122–132, April 1991.

[19] Richard P. LaRowe Jr., Carla Schlatter Elllis, and Laurence S. Kaplan. The Robustness of NUMA Memory Management. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 137–151, October 1991.

[20] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[21] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transaction on Computer Systems*, 7(4):321–359, November 1989.

[22] Kai Li. Scalability Issues on of Shared Virtual Memory for Multicomputers. In M. Dubois and S.S. Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 263–280. Kluwer Academic Publishers, May 1991.

[23] Kai Li and Karin Petersen. Evaluation of Memory System Extensions. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 84–93, May 1991.

[24] Kai Li and Richard Schaefer. A Hypercube Shared Virtual Memory System. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume I, pages 125–132, August 1989.

[25] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[26] The nCUBE/2. Technical Reference Manual, 1988.

[27] John K. Ousterhout. Why aren't Operating Systems getting faster as fast as Hardware? In *Proceedings of the 1990 USENIX Summer Conference*, pages 247–256, June 1990.

[28] Paul Pierce. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of the 4th. Conference on Hypercubes, Concurrent Computers and Applications*, volume I, pages 155–160, March 1989.

[29] Justin Rattner. Paragon System. In *Presentation in the DARPA High Performance Software Conference*, January 1992.

[30] Bill N. Schilit and Dan Duchamp. Adaptive Remote Paging for Mobile Computers. Technical Report TR CUCS-004-91, Department of Computer Science, Columbia Univeristy, February 1991.

[31] Charles L. Seitz. Mosaic C: An Experimental Fine-Grain Multicomputer. In *Presentation in the DARPA High Performance Software Conference*, January 1992.

[32] Gabriel M. Silberman. Determining Fault Ratios in Multilevel Delayed Staging Storage Hierarchies. *IEEE Transactions on Computers*, C-31(4), April 1982.

[33] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[34] Alan J. Smith. Disk Cache—Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[35] The Connection Machine CM-5. Technical Summary, 1991.