EVALUATION OF MEMORY SYSTEM EXTENSIONS

Kai Li
Karin Petersen

CS-TR-307-91

March 1991

Evaluation of Memory System Extensions*

Kai Li and Karin Petersen
Department of Computer Science
Princeton University

Abstract

A traditional memory system for a uniprocessor consists of one or two levels of cache, a main memory and a backing store. One can extend such a memory system by adding inexpensive but slower memories into the memory hierarchy. This paper uses an experimental approach to evaluate two methods of extending a memory system: direct and caching. The direct method adds the slower memory into the memory hierarchy by putting it at the same level as the main memory, allowing the CPU to access the slower memories directly; whereas the caching method puts the slower memory between the main memory and the backing store, using the main memory as a cache for the slower memory. We have implemented both approaches and our experiments indicate that applications with very large data structures can benefit significantly using an extended memory system, and that the direct approach outperforms the caching approach in memory-bound applications.

Introduction

The demand for large memory continues to increase. Large memory not only improves the performance of programs with large data structures such as scientific programs, artificial intelligence applications, and VLSI design tools, but also simplifies the design for systems such as memory-resident database and transaction processing systems [Sto84,LN88,SGM90]. As the gap between processor speed and disk paging has widened dramatically in the past few years, programs with large data structures have become very sensitive to virtual memory disk paging.

Recent work by Ousterhout [Ous90] shows that programs run much slower on fast machines than the raw speed of the the processor would indicate. He believes that low memory bandwidth is one of the reasons for these results: "Memory-intensive applications are not likely to scale well on these RISC machines. In fact, the relative performance of memory copying drops almost monotonically with faster processors, both for RISC and CISC machines."

On the other hand, the density of current DRAM chips, physical space and other engineering factors limit the size of main memory. When reaching the engineering limits, architects need to consider how to add additional memory devices to the memory hierarchy. We can envision several possible extensions: memory devices connected to the machine over a bus or a network, multiple memory modules in a shared-memory multiprocessor, and remote memory paging on a network of processors or on a multicomputer.

How much can a uniprocessor system benefit by having an extended memory and what is the best way to extend a memory system? In an attempt to answer these questions, we took an experimental approach to study two design alternatives for extending memory systems for a uniprocessor system: direct and caching. The direct approach adds a large number of inexpensive DRAM memory modules on a secondary memory bus that can be directly accessed by the processor's cache. The caching approach puts extended memory modules between the processor's main memory and its backing store, using the main memory as a cache.

We chose actual measurement over trace-driven simulation for evaluating different extended memory design alternatives. Although trace-driven simulation is an effective method to evaluate the behavior of a memory hierarchy and has been used effectively in the past [Bel66,MGST70,PACS73,Gec74,Smi82], it is difficult to collect meaningful traces for extended memory designs because programs requiring very large data structures usually run for more than an hour and their trace sizes are extremely large. It would take weeks to run a simulation, even if we were able to collect and store such traces. The actual measurement approach is lack of the flexibility of varying hardware parameters, but

*This research was supported in part by the National Science Foundation agreement CCR-8814265 and DCR-8420948, by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts N00014-85-C-0456 and N00014-85-K-0465.

it provides genuine, workload-based performance measurements.

We implemented the cached and direct approaches on an extended memory system prototype built under the MMM (massive memory machine) project at Princeton University. We selected and ran several application programs that require large data structures. These programs include a main memory database, Gaussian elimination with partial pivoting, quicksort, a recursive memory reference string generator and matrix transposition. Our experiments and analysis indicate that applications with very large data structures can benefit significantly using an extended memory system, and that the direct approach outperforms the caching approach for these memory-bound applications.

Extended memory architectures

The simplest and most trivial memory extension is to increment the number of memory boards on the machine. This approach is limited by the physical constraints, such as the density of memory chips, the slots on the machine backplane and the memory bus length.

The next obvious approach is to add inexpensive, slow memory modules into the memory hierarchy such that the extended memory modules are used as a large cache between the main memory and the disk devices. When a virtual page fault occurs, rather than going to the disk backing store, it sends a request to the slow memory modules instead. We call this approach the *caching* method (Figure 1).

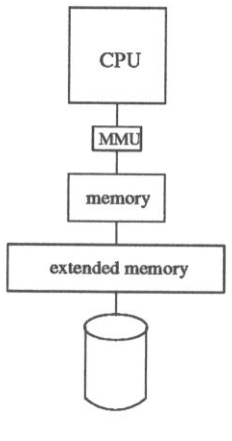

Figure 1: Caching method

Some researchers recently proposed using the main memories of other processors as the cache between the local main memory and the local or remote disk backing stores [LS89,Fel90,CG90]. We call this the *remote memory* method (Figure 2). The idea is based on the observation that modern network interconnections provide higher data transfer bandwidth than disks. This is particularly true for multicomputers.

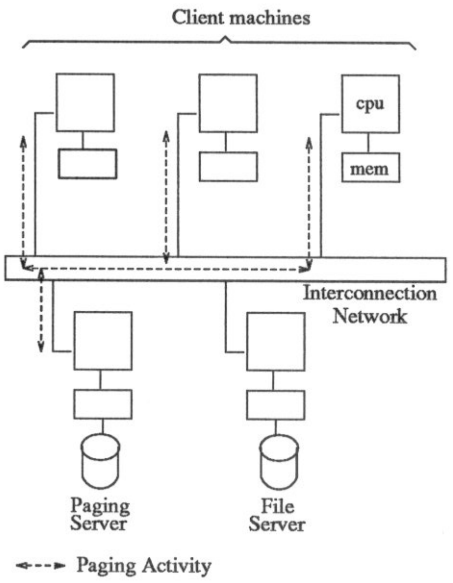

Figure 2: Remote memory

Another approach is to use a secondary memory bus to connect the processor with extended memory modules, shown in Figure 3. The processor can directly access memory cells in the extended memory modules as if it accesses its main memory. We call such an approach the *direct* method. Although the memory access time of the extended memory modules via the secondary memory bus is usually slower than the main memory, the simplicity of the design and the low cost of slow memory chips make this approach attractive. The secondary memory bus approach can be extended to a network of memory buses.

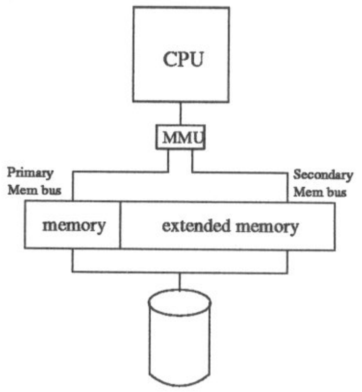

Figure 3: Direct method using a secondary memory bus

One can use the caching method to simulate the remote memory approach; therefore we concentrate on evaluating the caching approach and the direct approach.

GigaSUN prototype

We conducted our study on the GigaSUN prototype built at Princeton under the massive memory machine project. The GigaSUN is an enhanced Sun 3/180 with 32 Mbytes of primary physical memory and a large extended memory system as shown in Figure 4. A VME bus [VME85] is used as a secondary memory bus to connect the main memory bus with eight extended memory modules.

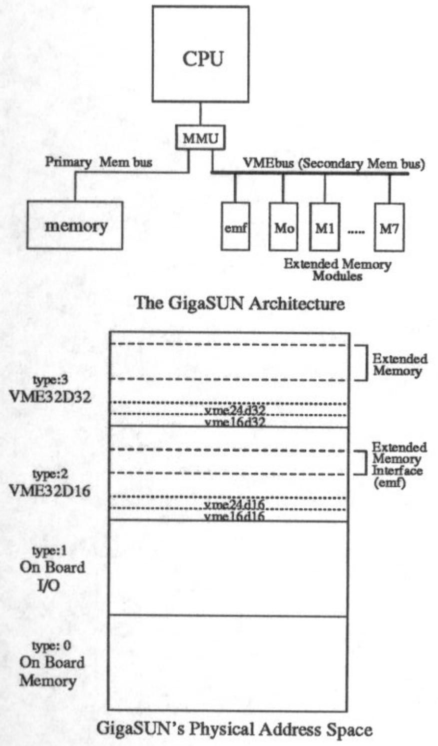

Figure 4: GigaSUN Architecture

Each memory module box contains up to eight 16 Mbyte memory cards and an MC68020 processor to handle power-up, memory diagnostics and clearing. We configured each extended memory module with four 16 Mbyte memory cards, for a total of 512 Mbytes of memory.

The extended memory interface is configured as an A32:D16 VME Bus slave device. It contains the status words for each extended memory module such as interrupt vectors, error vectors, and control registers. The main function of the interface is for diagnostics and setup.

The memory modules are directly connected to the A32:D32 address space of the VME bus. The Memory Management Unit (MMU) hardware was modified to allow direct accesses to the extended memory as part of the physical address space. The 32-bit physical address space on the GigaSUN is partitioned into four 1 Gbyte spaces.

The access time of the fast main memory is about 300ns, and that of the extended memory is about 600ns for 32 bit accesses.

Baseline, caching and direct

The GigaSUN architecture currently runs a modified Mach operating system (version 2.5) [RTY+88] for evaluating three models: baseline, caching and direct. The baseline model does not use any extended memory modules. The other two models are implemented by modifying the Mach virtual memory system. In all three models the page size is 8 Kbytes.

For the caching model, the fast physical memory serves as a cache for pages in the extended memory. The virtual memory system maintains the mapping between the fast and slow memories and the mapping between the slow memory and disk. In the direct model, a virtual page can be mapped to either a fast memory page or a slow memory page. The virtual memory system uses the disk as the backing store for both fast and slow memory.

We used the external memory management interface [You89] to implement both the direct and caching models. For the direct model, we used the device pager in the Mach kernel. The pager creates device memory objects for the extended memory pages using a specified physical address mapping routine. For the caching model, we wrote our own external pager. The pager uses the extended memory as the backing store for main memory and uses the disk as the backing store for the extended memory.

The `vm_map` system call provided by Mach allows client programs to specify which pager manages a range of virtual addresses in a given address space. We used this mechanism to map the extended memory pages into an application's address space.

Experiments

We selected programs with varied memory reference patterns for our benchmark suite to evaluate memory extension approaches. The test suite includes a main-memory database, quicksort, Gaussian elimination, a recurrent number generator, and matrix transposition.

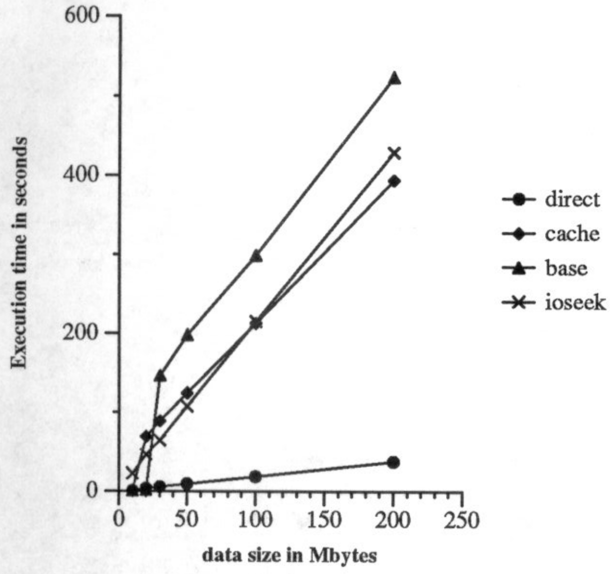

Figure 5: Elapsed time of DB queries

Main-memory database (DB) is a simple database manager which stores its entire database data structure in the virtual memory address space. The database used in our experiments was randomly generated with a fixed record size of 1,024 bytes. Each contains a number of integer and character string fields. There is no index method for query executions; all queries search the database sequentially.

Quicksort (QS) uses the standard quicksort algorithm to sort an array of 1,024-byte records. Each record consists of unsigned long integers and array of records was generated randomly. The entire array is stored in the virtual memory. We implemented two versions of quicksort. The first uses the rightmost field as the key for sorting and the second compares all fields.

Gaussian Elimination (GE) implements the Gaussian Elimination method for solving a linear equation with partial pivoting. Since this application will take a relatively long time for large equations, we used long integer data types rather than floating point in order to reduce the running time of the program while maintaining the same memory reference patterns.

Recurrent number generator (RNG) is a program that generates numbers based on the ten numbers generated previously. The program keeps all the generated numbers in its memory heap. This program was chosen because it exhibits a high degree of locality of reference.

Matrix transposition (MT) builds the transpose of a matrix by simply interchanging the appropiate elements. We chose this experiment because it is a substantial component of signal and image processing applications and is known to have very little locality of reference. Previous work on this problem was focused on avoiding to thrash the memory system by subdividing MT based on the structure of the application using it (e.g., FFT) or on algebraic properties of the problem itself [Ekl]. In our implementation, the whole matrix is loaded into the applications address space and the simple element exchanging algorithm is used.

The implementations of the benchmark programs for all three models (baseline, caching and direct) are the same, except that for the direct model we used an extended memory allocator rather than the `malloc()` in the standard C library, in order to store data structures explicitly in the extended memory space. For each program, we also implemented a version that stores data structures in files. These programs use `read`, `write`, and `seek` system calls to access data explicitly. Such an implementation provides a data point to compare with the three models that use a large memory address space. In all cases, the code segments are always stored in the fast, main memory. In the following figures, we use the label **base** for the baseline model, **cache** for the caching model, **direct** for the direct model, and **ioseek** for the case of storing data structures in files.

Figure 5 shows the elapsed times of the executions of a set of queries on the main-memory database as a function of the database sizes. We observe that the direct method significantly outperforms all other approaches. The direct method is an order of magnitud better than both the baseline case and caching approach. Since a query in the main-memory database examines only one field for each record, it does not exhibit high locality of reference. The caching scheme does not work well because it pays for the overhead of moving the entire record from extended to main memory even though only 40 bytes are actually used in the query execution.

There are two experiments for the quicksort program, as mentioned above. The first accesses only the key field for comparisons and all the fields during exchanges; whereas the second accesses the entire record for every comparison and exchange. Clearly, the first experiment exhibits a lower degree of locality of reference than the second one.

The results of the two cases are different. In the first case, as shown in Figure 6, when the data size exceeds the capacity of local main memory, the caching approach takes about five times as much as the direct approach. This is because the direct approach accesses only the related fields while the caching approach pages in the entire record for comparisons. Figure 7 shows the second case. As expected, the cache approach performs better in this case, but it still takes twice the time of

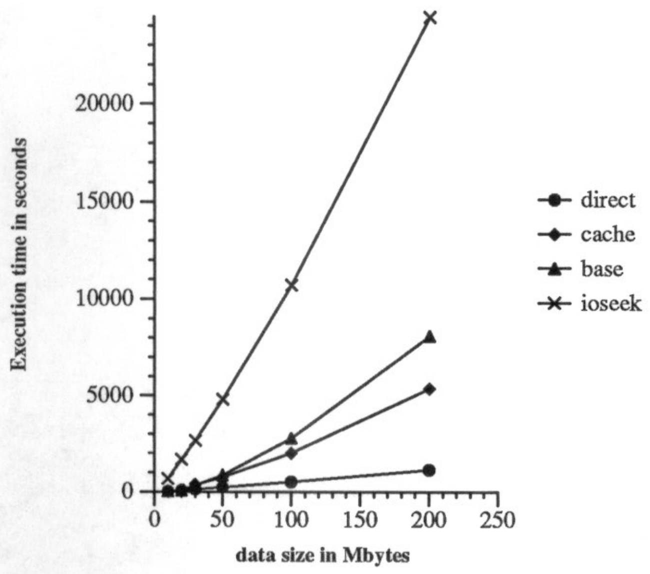

Figure 6: Elapsed time of QS on keys only

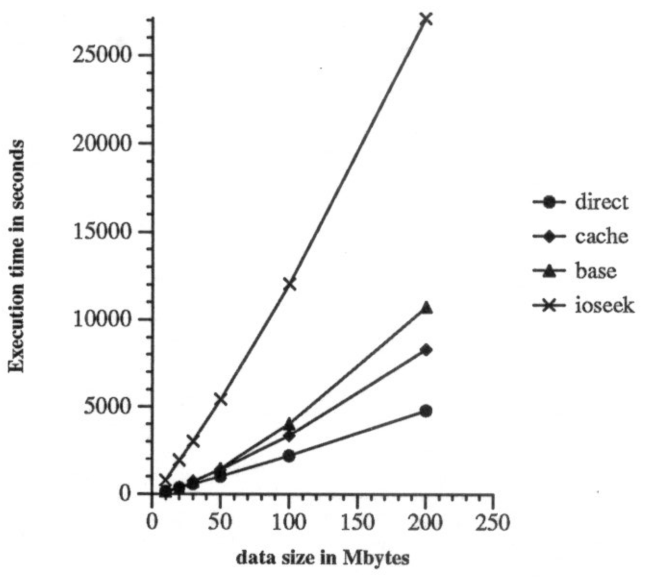

Figure 7: Elapsed time of QS on full records

the direct approach. In both cases, the baseline model performs worse than either the caching approach or the direct approach.

Figure 8 shows the results of the Gaussian elimination program with partial pivoting. The baseline model performs about the same as the case of storing data structures in files. The primary reason is that pivoting is quite random, so that the caching provided by the file system buffers is just as good as the virtual memory page replacement mechanism. The caching approach reduces the elapsed time by about 30% and the direct approach by about 70%.

It seemed counterintuitive to us that the direct approach outperformed the caching approach in the Gaussian elimination program, since iterative, numeric programs exhibit a high degree of locality of reference. We realized later that the main reason for this is the fact that the program code segments were stored in fast main memory; the locality of reference in instruction fetches does not affect the performance of the program. The locality of reference in data accesses to the extended memory is not high enough to make the caching approach worthwhile.

Figure 9 displays the results of the recurrent number generator. In this case, our experiments show that the performance of the baseline and direct model is comparable. As mentioned above, the recurrent number generator exhibits a high degree of locality of reference because each number is generated using the previous 10 numbers. This degree of locality makes the overhead of disk paging insignificant; the cached approach performs worse than the baseline model because the default inode pager resides in the kernel, while the external pager, that manages the extended memory caching, executes in user space, and therefore has a more significant fault processing overhead.

Figure 10 shows the results of the matrix transposition experiment. The direct approach outperforms both paging approaches by over an order of magnitude. While the direct approach performs proportional to the problem size, both the cached and baseline model suffer from serious performance degradation as soon as the matrix becomes larger than the local main memory. This indicates that using such an approach the algorithms used for certain problems, such as matrix transposition, can be made much simpler, by ignoring its influence on the memory system.

We observe the average utilization of the secondary memory bus for our experiments using the direct approach in Figure 11. The utilization for QSK, QS, GE and MT is about the same, in the order of 30% of the secondary memory bus bandwidth. The utilization for DB is only about 4%, while for RNG it is almost 60%. A reasonable explanation for the high utilization by RNG is that very little processing (one integer addition) is done between data accesses.

In summary, our experiments show that the direct approach works well for most of the selected application programs using large data structures, and that the direct approach outperforms both the caching and baseline models for all tested applications (Fig. 12). Two conditions are critical in our experiments:

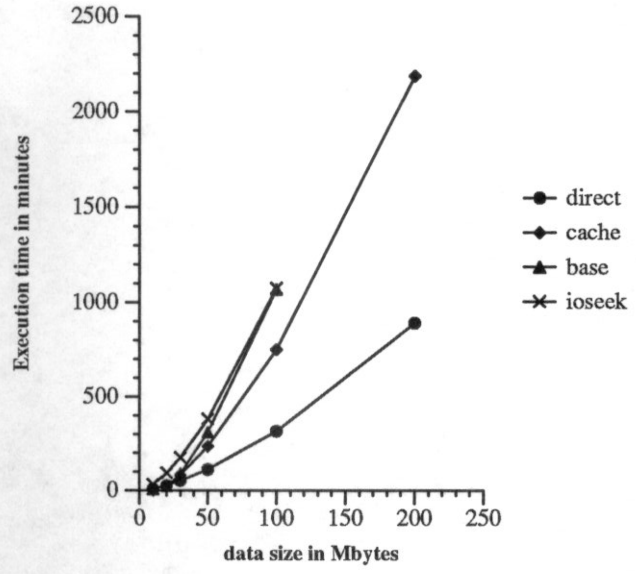

Figure 8: Elapsed time of GE

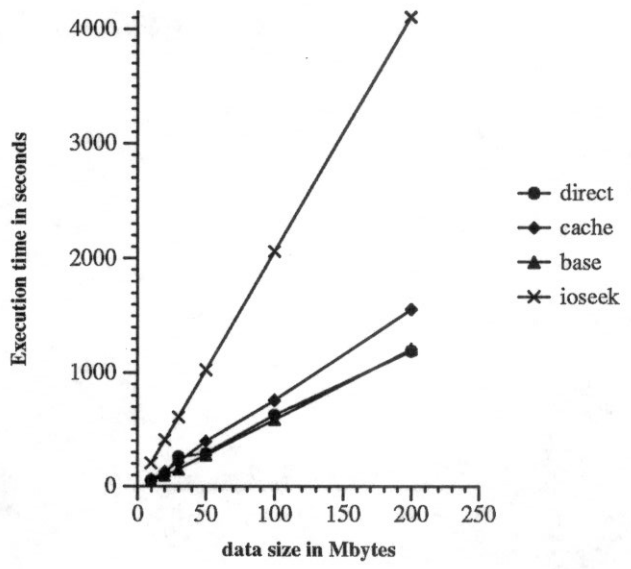

Figure 9: Elapsed time of RNG

- the program code segments are kept in the fast main memory, and
- the access time ratio of the fast main memory to the extended memory is about 0.5.

We expect that the first condition is true for most applications because code segments are relatively small, and that it is easy to achieve the access time ratio of 0.5 in today's memory system designs.

Access time vs. page hit ratios

In this section, we study the relationship between the ratio of the access times for fast main memory to that of extended memory and the degree of locality programs display. Our main goal is to see under what conditions the caching approach would outperform the direct approach.

We use the following factors to characterize the average cost per page for both caching and direct models:

- n — average number of accesses to a page without a page fault,
- C_{fast} — access time (per memory word) of the fast main memory,
- C_{slow} — access time (per memory word) of the extended memory,
- C_{fault} — overhead of a page fault (per page, excluding page movement),
- d — average percentage of replaced pages that are dirty (page out events during replacements), and
- p — page size in words.

The average cost on accessing n memory words under the direct model is straightforward, $n \times C_{slow}$; whereas the average cost under the caching model includes the cost of handling a page fault, the cost of a page replacement and the cost of accessing n fast memory words. Thus, the following inequality must hold for the caching approach to outperform the direct approach:

$$p(1+d)(C_{fast}+C_{slow})+C_{fault}+n\cdot C_{fast}\leq n\cdot C_{slow}.$$

Using the inequality, we can derive the relationship between the access time ratio (C_{fast}/C_{slow}) and the page hit ratio (data locality of reference).

Table 1 shows the smallest data page hit ratios (locality of reference) needed for the cache approach to outperform the direct approach. In this table, we assumed that 50% of the replaced pages are modified, and therefore paged out from main memory to the extended memory during page faults. The measured overhead of a page fault served by the external pager, excluding page moving, is equivalent to 10500 main memory accesses (or approximately 3 msecs on the GigaSUN running Mach 2.5). Each memory word is 4 bytes long. These parameters are dependent on the hardware and virtual memory implementations. The average percentage of replacing modified pages on a fault also depends on the application programs. In our experiments, this percentage varies significantly for the different programs. For example, $d = 0$ in the DB program, because the database queries do not modify any records; whereas both QS and GE modify most pages.

Access time	**Page Size**			
ratios	**1 Kbytes**	**2 Kbytes**	**4 Kbytes**	**8 Kbytes**
0.750	0.999971	0.999973	0.999976	0.999981
0.667	0.999956	0.999960	0.999965	0.999973
0.500	0.999914	0.999922	0.999934	0.999949
0.333	0.999834	0.999853	0.999880	0.999912
0.200	0.999688	0.999735	0.999797	0.999862
0.100	0.999389	0.999525	0.999671	0.999797
0.010	0.997991	0.998876	0.999402	0.999691
0.001	0.997470	0.998718	0.999355	0.999676

Table 1: Page hit ratios needed for the caching approach to outperform the direct approach.

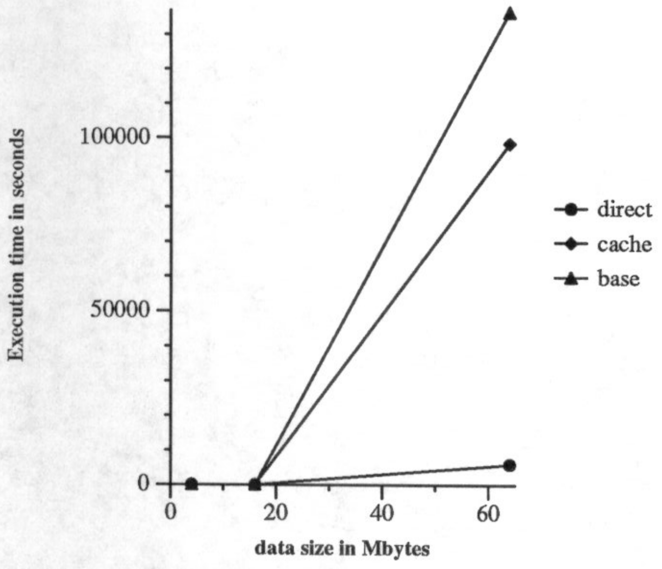

Figure 10: Elapsed time of MT

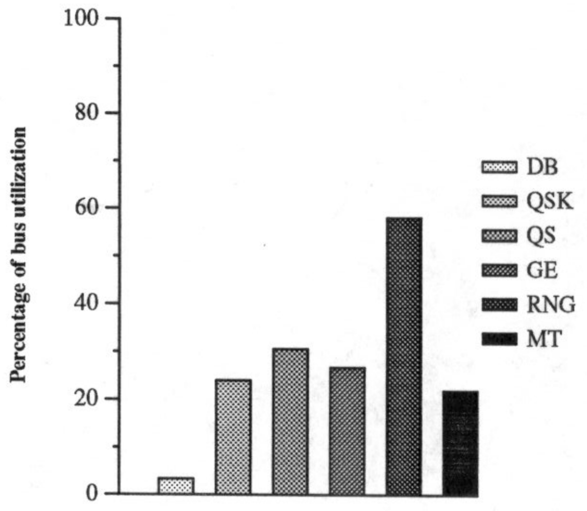

Figure 11: Average utilization of the secondary memory bus for the direct approach

The table tells us that in order for the caching approach to outperform the direct approach on the GigaSUN extended memory system, application programs need to have a page hit ratio greater than .999949. Based on the algorithms, data placements, and the GigaSUN configuration, we measured the page hit ratios for DB, QSK, QS, GE, RNG and MT to be .9875, .99952, .9999105, .999692, .9999511 and .7262 respectively. These estimates further confirmed our experimental results.

Other design issues

Although our experiments have given us some insight in design of extended memory systems, many problems in architecture design, virtual memory design, and behaviors of memory-bound applications still remain open.

It is clear that a large, fast main memory would improve the overall performance of our benchmark programs for all three models. A large main memory implies fewer page faults for both the baseline model and the caching approach. The direct approach should also benefit from a large main memory because a large portion of the data structures of application programs can then be kept in the fast memory, reducing the total overhead of data accesses.

A physical data cache that caches both the fast main memory and the extended memory would benefit programs with a high degree of temporal locality of reference. Programs such as the main-memory database may not benefit since each query scans the database exactly once for each execution. It is still an open question whether most applications with very large data structures exhibit a high degree of temporal locality of refer-

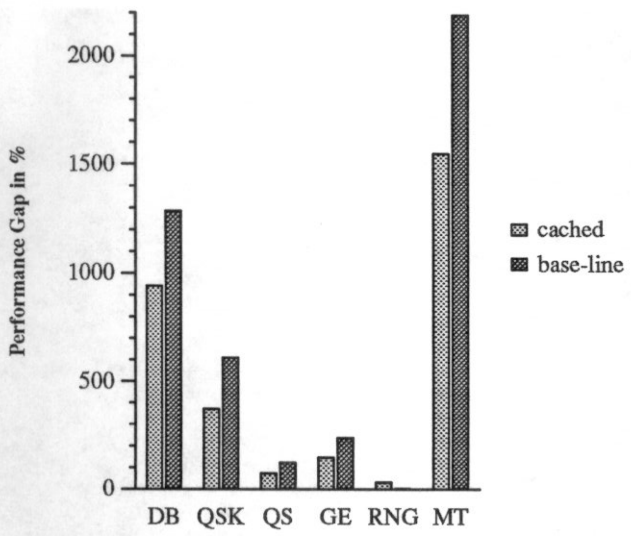

Figure 12: Performance gap of the direct approach vs. the cached and baseline approaches

ence.

Changing the caching unit size will affect the performance of the caching model. In our implementation, the caching unit is a virtual memory page of 8 Kbytes. Small page sizes may help applications that only access small amounts of data within a page. On the other hand, small page sizes will introduce greater paging overhaed for applications that reference most memory locations in each page due to a larger number of page faults. It requires careful study to determine the optimal page size for the caching model with respect to other system parameters.

Another open question is whether it is better to have a combination of the direct and caching approaches. In other words, can programs with large data structures benefit from a dynamic physical page migration mechanism that move frequently accesses pages from the extended memory to the fast main memory? For parallel programs, LaRowe and Ellis [LE90] observed that on multiprocessors, dynamic page placement does worse on occasion due to artificially induced page faults for page location reevaluation. A similar phenomenon is likely to occur when the extended memory on the secondary memory bus is large.

Direct Memory Access (DMA) between the extended memory and disks will benefit applications whose data structures cannot be stored entirely in the extended memory. Although the GigaSUN architecture does not have DMA disk controllers on the secondary memory bus, our experimental results still stand because the data sizes of all the programs in our experiments do not exceed the capacity of the extended memory.

If it would be a good idea to have DMA page transfers between fast main memory and extended memory remains an open question, as well. It is unclear whether such DMA page transfers improve the performance of extended memory architectures, since paging between fast main memory and the extended memory differs from paging between the main memory and disks. On traditional architectures, a DMA data block transfer between main memory and a disk usually takes tens of milliseconds of which only a very small portion is due to the actual data transfer and a large portion corresponds to seek time for data blocks on the disk storage media. During this sufficiently long period of time the CPU can do other useful work. On extended memory architectures, we expect the time for a DMA page transfer between the fast main memory and extended memory to be reduced by at least an order of magnitude. If the overhead of context-switches and setting up DMA registers is significant, the DMA page transfer may not be worth while. The answer to this question also affects the effectiveness of dynamic page placement policies in this environment.

Although slow DRAM chips are less expensive than the DRAMs used in main memory, building a massive extended memory can be quite costly. An idea to reduce the cost of building a large extended memory system is to share the extended memory system among several machines. We call such an extended memory a *memory server* (as opposed to a compute server). A memory server is an extended memory system available to a set of processors. These processors can directly access the memory locations in the extended memory systems. Page frames in the memory servers are allocated to processors upon request. Such an architecture can be viewed as a multiprocessor system without maintaining cache coherence.

Related work

There is a large body of literature related to memory hierarchies and extensions for uniprocessor systems. Most work concentrates on multiple level memory hierarchies [Bel66,MGST70], program memory access patterns [BG68], virtual memory systems [Den70,Den80], memory caches [Smi82], and disk caches [Smi85].

The idea of massive memory architectures was first proposed in [GMLV83,GMLV84]. They pointed out that the performance of applications requiring very large data structures can be significantly improved if the data structures can be stored entirely in memory. They also proposed several approaches to designing massive memory systems including the secondary memory bus approach. Garcia-Molina and his colleagues also show that in many cases, database queries on a uniprocessor with massive memory can outperform those on a database machine [GMCHL84].

The IBM 3090 system provides an extended memory system called *expanded memory* [Tuc86,CKB89]. The expanded memory system only supports 4K page block data transfer between itself and the main memory. This operation is controlled by the operating system. Similar to the caching model of our implementation, the expanded memory runs synchronously with respect to the processor that requests the transfer. The design decision for synchronous transfer was based on a study comparing the CPU time required for both asynchronous and synchronous approaches. The CPU overhead for the asynchronous approach, basically consistent of repeated context switches, was found to be substantially greater than for the synchronous approach. Unlike our GigaSUN prototype, the IBM 3090 system does not allow the processor to access the expanded memory directly.

Recently, much attention has been dedicated to page placement, replication and migration methods for Non Uniform Memory Access (NUMA) architectures [LE90,CF89,BFS89]. Their results do not directly apply to uniprocessor systems.

The research on shared virtual memory [LS89,LH89] showed that the performance of paging can be improved by an order of magnitude when using the physical memory of another processor node as a temporary backing store. The replication of pages in a shared virtual memory system can further reduce the overhead of paging.

The research at Purdue [CG90] and at the University of Washington [Fel90] explores the idea of using remote memory as a fast backing storage device for uniprocessor systems. The work to date has been concentrated on how to manage remote memory pages and disk backing stores.

Conclusion

We have evaluated both caching and direct approaches for designing extended memory systems. Our experiments on the GigaSUN prototype machine shows that an extended memory system can significantly improve the performance of applications with large data structures.

The access time ratio of the fast main memory to the extended memory is the key factor that determines whether the direct approach is better than the caching approach. On our prototype machine with the access time ratio of about 0.5, the direct approach outperforms the caching approach in most applications with large data structures.

When evaluating extended memory design alternatives, one should separate data memory references from instruction references because program code segments are usually small and they should always be stored in the fast main memory. Our experiments show that data memory references of many memory-bound applications do not have a very high degree of locality of reference.

It is possible to combine the direct approach with the caching approach; frequently accessed pages in the extended memory can be migrated to the fast main memory. This approach can be attractive to applications with a high degree of temporal locality of reference.

Our experiments also indicate that the average utilization of the secondary memory bus will probably be in the order of 30% and that it is feasible to let multiple processors share an extended memory, leading to the concept of memory servers in a computing environment.

Acknowledgement

We would like to thank Daniel Babará, Mary Fernández, Hector Garcia-Molina, Mark Greenstreet, Anne Rogers, and our reviewers for their valuable comments.

References

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[BFS89] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[BG68] 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.

[CF89] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor: Experiences with platinum. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[CG90] D. Comer and J. Griffoen. A new design for distributed systems: The remote memory model. In *Proceedings of the USENIX Summer Conference*, pages 127–135, Anaheim, California, June 1990.

[CKB89] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM Systems Journal*, 28(1), 1989.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, September 1970.

[Den80] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[Ekl] J. O. Eklundh. Efficient matrix transposition. In T. S. Huang, editor, *Topics in Applied Physics, Two Dimensional Digital Processing II*, volume 43, pages 9–35. Springer Verlag, Berlin Heidelberg New York.

[Fel90] Edward Felten. Distributed virtual memory. Presentation in Topaz User Group Meeting, DECSRC, June 1990.

[Gec74] J. Gecsei. Determining hit ratios for multilevel hierarchies. *IBM Journal of Research and Development*, 18(4):316–327, July 1974.

[GMCHL84] H. Garcia-Molina, R. Cullingford, P. Honeyman, and R. Lipton. Mmm performance on certain database benchmarks. Technical Report 327, Department of EE and Computer Science, Princeton University, May 1984.

[GMLV83] H. Garcia-Molina, R. J. Lipton, and J. Valdes. Analysis of the massive memory architectures. Technical Report 313, Department of EE and Computer Science, Princeton University, May 1983.

[GMLV84] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *IEEE Transactions on Computers*, C-33(5):391–399, May 1984.

[LE90] Richard P. LaRowe and Carla S. Ellis. Experimental comparison of memory management policies for numa multiprocessors. Technical Report CS-1990-10, Duke University, 1990.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LN88] Kai Li and Jeffrey F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the International Symposium on Database in Parallel and Distributed Systems*, pages 177–187, December 1988.

[LS89] Kai Li and Richard Schaefer. A hypercube shared virtual memory. In *Proceedings of the 1989 International Parallel Processing Conference*, volume Vol:I Architecture, pages 125–132, August 1989.

[MGST70] 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.

[Ous90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX Summer Conference*, pages 247–256, Anaheim, California, June 1990.

[PACS73] A.V. Pohm, O.P. Agrawal, C.W. Cheng, and A.C. Shimp. An efficient flexible buffered memory system. *IEEE Transactions on Magnetics*, MAG-9(3):173–179, September 1973.

[RTY+88] R.F. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August 1988.

[SGM90] Kenneth Salem and Hector Garcia-Molina. System m: A transaction processing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[Smi82] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi85] Alan J. Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Sto84] Michael Stonebraker. Virtual memory transaction processing. *Operating Systems Review*, 18(2):8–16, April 1984.

[Tuc86] S. G. Tucker. The ibm 3090 sytem: An overview. *IBM Systems Journal*, 25(1), 1986.

[VME85] Vmebus specification manual. Revision C, February 1985.

[You89] M. Young. *Exporting a User Interface to Memory Management from a Communication Oriented Operating System.* PhD thesis, Carnegie Mellon University, 1989.