Massive Memory Means Massive Performance

Arvin Park
Department Computer Science
Princeton University
Princeton, New Jersey 08544

CS-TR-036-86

May 9, 1986

Abstract

While most of the computing world has been exploring the uses of parallelism to increase computational speed, we at Princeton have been examining methods of using large amounts of semiconductor memory to achieve the same end. We have demonstrated that time-space tradeoffs can be exploited across a wide range of applications to speed up computational tasks (On some tasks, our 128 megabyte VAX will outperform even the fastest supercomputers). These time-space tradeoffs coupled with rapidly falling semiconductor memory prices make Massive Memory systems economically inevitable. This paper documents performance results from our 128 megabyte testbed machine, and goes on to investigate promising application domains for Massive Memory computing systems.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0446 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1. Introduction

This paper presents the case for Massive Memory computing systems which we believe will become economically inevitable as the cost of semiconductor memories continue to decrease. We loosely define Massive Memory systems as; "Computing systems that possess greater than ten megabytes of semiconductor memory for each MIPS (million instructions per second) of CPU speed". Such systems, with large amounts of semiconductor memory, will fundamentally alter the way computations are performed. Our performance results show that not only I/O bound tasks benefit from Massive Memory. Time-space trade-offs can be utilized to extract computational speed on seemingly "CPU bound" problems. This paper demonstrates **how** and explains **where** Massive Memory can be used to improve computational performance.

Section two presents the case for Massive Memory, including performance results from our 128 megabyte VAX-11/785. Section three explains our benchmarking procedures and results. We develop a new set of metrics to gauge the impact of massive memory on computational tasks. Section four describes continuing research into Massive Memory computation.

2. A Case for Massive Memory

The Massive Memory project was initiated as a continuing trend in computer hardware costs was recognized; Memory is getting cheaper. And this is occurring at an exponential rate[1]. The amount of memory that one dollar will buy doubles about every 2 years [10]. At present one gigabyte of semiconductor memory costs about $250,000 (at the board level) which is not significantly more than the cost of a large minicomputer system[11]. If this trend continues, the same gigabyte will cost less than nine thousand dollars by 1996.

How can we use this memory? There are two ways. Some I/O bound applications will immediately benefit from large amounts of semiconductor memory. Database applications, Artificial intelligence applications, and VLSI layout problems all fit into this category[2]. For these problems the performance limiting overhead of swapping disk pages into and out of main memory can essentially be eliminated.

Every experienced programmer has encountered programs that "thrash" main memory. And often these programs require painstaking programming and algorithmic development to run in small amounts of main memory. External sorting, and the old "line at a time" editors are examples of such applications. Massive memory will improve all of that. Not only will I/O bound applications run faster, but programming tasks will be greatly simplified.

But massive memory has applications that extend beyond simply improving I/O performance. Memory space can be traded for computational time to drastically increase computational speeds.

Most of the computational world has been exploring ways of using parallelism to increase computational speed, and these efforts have met with some success, but only on very specific applications, or very limited degrees of parallelism. In the Massive Memory Project we are taking a different approach, instead of using parallelism to increase computational speeds, we are exploiting time space tradeoffs to achieve the same end. We expect that Massive Memory systems will improve performance on many tasks that parallel processors do poorly on and vice versa. We contend that both lines of research are important, but Massive Memory has not been studied closely.

Figure one shows a plot of different types of computer systems with processor speeds scaled on the horizontal axis and memory size plotted on the vertical axis. Many highly parallel processors use relatively small amounts of memory compared with a great deal of computational hardware[7] so they fit into the lower right corner of this graph, while most conventional processors fit into a diagonal band in the middle. These conventional systems seem to validate Amdahl's Law which states a processor will have about one megabyte of memory for every MIPS of computational speed. Massive Memory systems fit into the relatively unexplored region of the upper left corner of the graph. Ideally one would want a processor with both large amounts of memory and fast computational speeds (upper right corner), but economic considerations and technological limitations make such a system hard to realize. Trade-offs must be made, and in light of current economic trends these trade-offs will begin to favor Massive Memory.

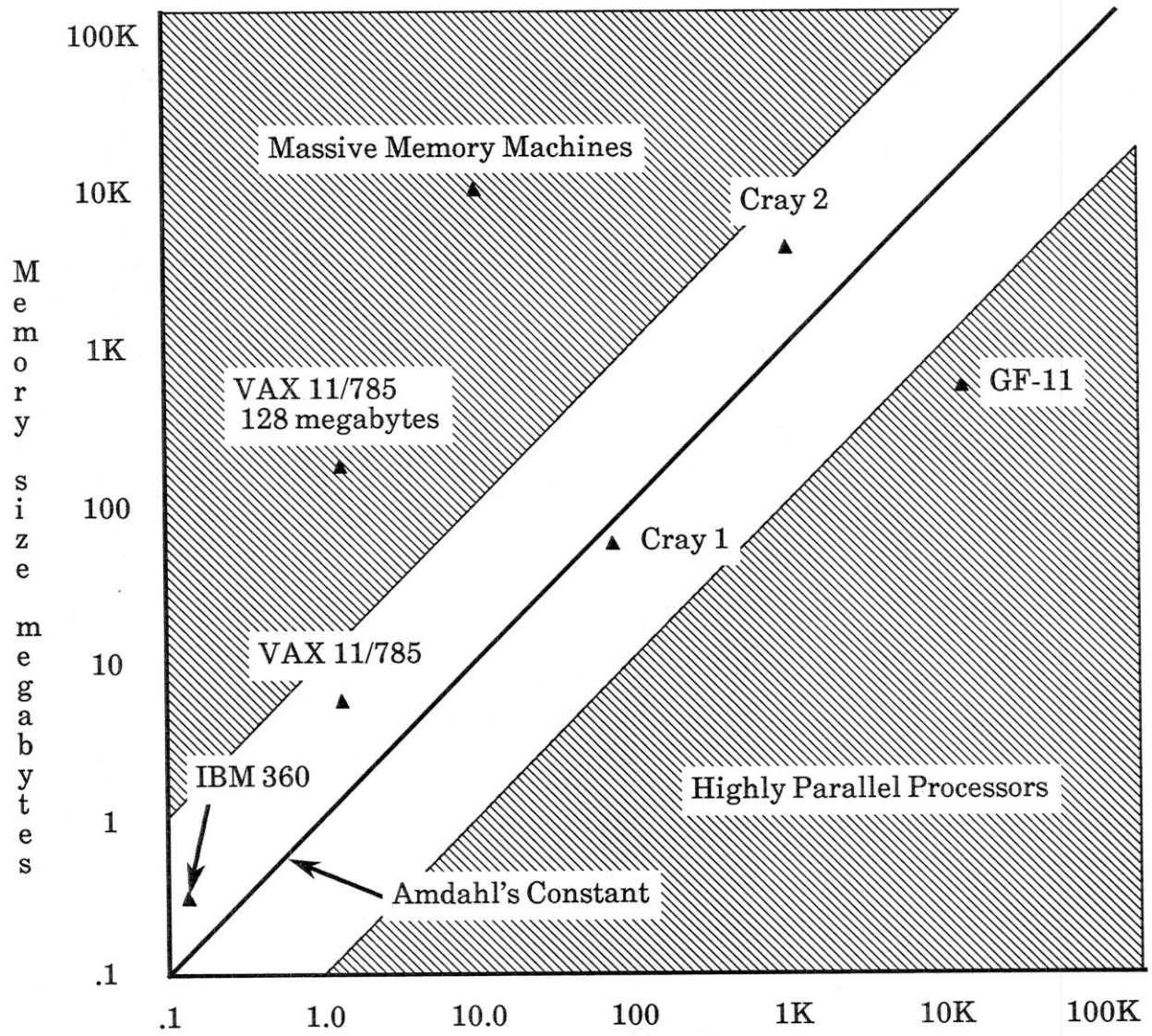


Figure1

But how can time space tradeoffs be used to improve computational speed? A lot of techniques have been developed. And a great body of literature on the subject exists[3][4][5]. But until systems are built with massive amounts of memory many of these techniques will remain infeasible. We have identified several techniques that can be used to trade memory space for time.

1. Precomputation and Table Lookups
2. Subgoal Storage
2. Multiple Access Structures

4. Redundant Code

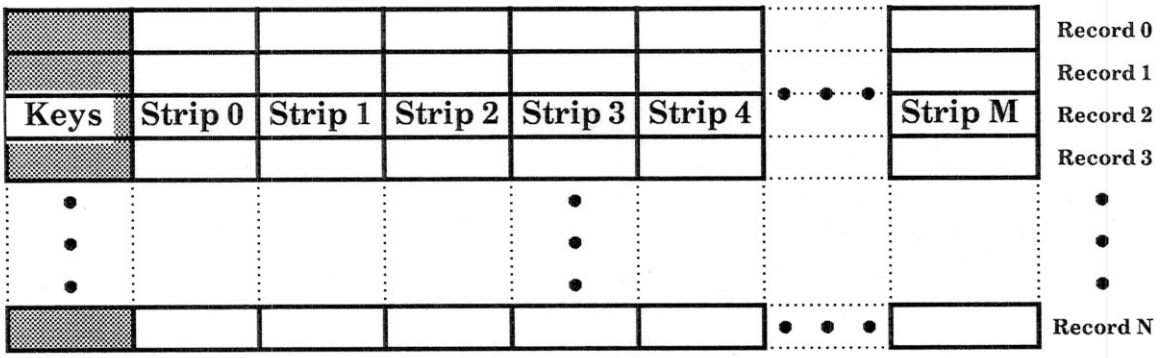
Precomputation involves precomputation of results for commonly referenced functions. We have constructed a program to evaluate transcendental functions (Sines, Cosines, Tangents, Etc.) on our test bed machines. Instead of performing the standard multi-term Taylor Series expansion along with it's associated multiplies and adds. Our system simply performs a lookup into a table of precomputed values. This performance is contrasted with the standard UNIX system routine in the table one. Note that we gain almost two orders of magnitude in speed by precomputing. Also note that there is an accuracy limitation imposed by our table size. We have to produce a larger table to get more accuracy while a Taylor Series approximation has to compute more terms. We can reduce memory usage by interpolation on a smaller table, at the cost of computational time. Fast transcendental functions have obvious applications to **real-time motion control** systems. Note that this technique can be used to evaluate an entire family of functions (trig. functions, exp, etc.)

	Sine	**Cosine**	**Tangent**	**Square Root**
Ultrix 1.1 Math Library	312	308	385	624
Table Lookup	9.15	9.15	9.15	9.15

Table 1 Common Function Evaluation Timings (microseconds)

We have found another application for precomputation. Several researchers at Princeton (Lipton, Sandberg North) have developed **Strip Sort**, which is an external sorting algorithm that uses massive amounts of memory (say 500 megabytes) to sort even larger databases (say 10million 1000 byte records = 10 gigabytes). This algorithm sorts these large records with only order N disk accesses, and it has proven to improve performance on real problems by as much as 10 times [6]. The algorithm

proceeds by first sorting the keys of the records (which are assumed to be much smaller than the record size) in core, and in doing so produces a permutation array which can be used to permute the rest of the database one 'strip' at a time (see figure 2). The permutation array is **precomputed** from the small keys and then this is used to sort the larger records very quickly.



Algorithm: assumes all keys or one strip can fit into main memory

First sort the keys in core generating a permutation array

For all strips:

Read one strip in from disk

Use the permutation arrray to sort the strip in core (linear time)

Output sorted strip to disk

Figure 2 **Strip Sort**

Subgoal storage is another powerful technique to trade space for time. The idea is to compute and store intermediate results which can then be used to attain a final goal. Dynamic programming fits into this category. Many dynamic programming algorithms use O(N**2) or O(N**3) and more space. Storing this amount of data becomes impractical on a conventional computing system if N becomes greater than about two thousand. We have coded the Floyd-Warshall [4] all pairs shortest path algorithm. On our 128 megabyte VAX-11/785 we can find the all pairs shortest paths on an 8000 node graph in 60 hours. Doing the equivalent computation on the same machine with a standard 8 megabyte memory configuration takes 50 days!

This is not surprising because we can keep all of our data structures core resident, thereby eliminating all paging overhead.

Another dynamic programming algorithm to find RNA secondary structure takes O(N**3) time O(N**2) space[5]. Finding the optimal folding for a 2000 base strand of RNA can take months on a VAX with standard memory configuration[6]. We are presently programming this run on testbed machine, it should be able to sequence the same 2000 strand base in 20 hours.

Internal sorting fits into the classification of subgoal storage. The data starts out in its initial unsorted order and works its way through intermediate stages until it is finally completely sorted. We have coded and internal quicksort program that can sort 30 million four byte integers in 47 minutes twenty one seconds. The same program run on a standard 8 megabyte VAX will take 3 days to run. Internal sorting is the fastest way to sort[12], and a massive memory machine therefore provides the fastest method of sorting large data sets.

Storing several access structures to a data set can also improve performance. Several sets of indices to a data base can be stored, each sorted on a different key. This facilitates multi-dimensional retrieval. In directed graph problems, storing adjacency lists, reverse adjacency lists, and adjacency matrixes often speeds up computations. Massive Memory will allow further extensions of multiple access structures to extract further performance from important applications.

Space can be traded for time in program code by unrolling loops, and inserting macros in place of subroutine calls. These methods of redundantly storing code avoid the overhead of loop incrementing and subroutine calls.

3. Benchmarking Procedures and Results.

We presently have a VAX-11/785 testbed machine configured with 128 megabytes of main memory interleaved between two controllers. This will soon be expanded to 256 megabytes. We have modified a version of ULTIRX 1.1 so that it can be reconfigured to simulate a machine with any number of megabytes between one and 128. All system buffers and paging thresholds are scaled appropriately.

We first performed tests to verify that the operating system and machine was performing up to published statistics for the VAX 11/785. To do this we ran the Dhrystone benchmark set[9]. which mainly measures performance on a variety of operating system tasks. Our machine benchmarks at 1968 dhrystones, and this concurs with published figures for the 785 (see table 2).

We next measured I/O performance to our RA-81 disk drives (see table two). These include the overhead of moving data through the buffer cache, these results agree with established system performance numbers.

Dhrystone:	1968 Dhrystones per second
Disk Accesses:	50 M bytes in 4K pages (2.5 megabyte buffer cache)
Sequential Reads	323.4 seconds (158.3 K bytes/second)
Sequential Writes	321.5 seconds (159.3 K bytes/second)
Random Reads	622.3 seconds (82.3 K bytes/second)

Table 2 **Standard System Performance Benchmarks**

We wanted to gauge how performance degrades as main memory size is reduced, and a program begins to "thrash". To do this we isolated two types of data access patterns. A simple sequential scan of memory, and random probes of memory. Programs that manipulate arrays access memory in a sequential manner. While hashing, binary searching, and pointer manipulation programs access data in a relatively random fashion.

Two programs where produced, one that simply scans memory sequentially, and one that makes random probes. The main memory size was then varied to produce the two graphs in figures three and four. Time factor (on the vertical axis) is a measure of the increase in a program's running time. A program's running time with all data in core is 1. As one would expect, the sequential scan of memory was relatively less

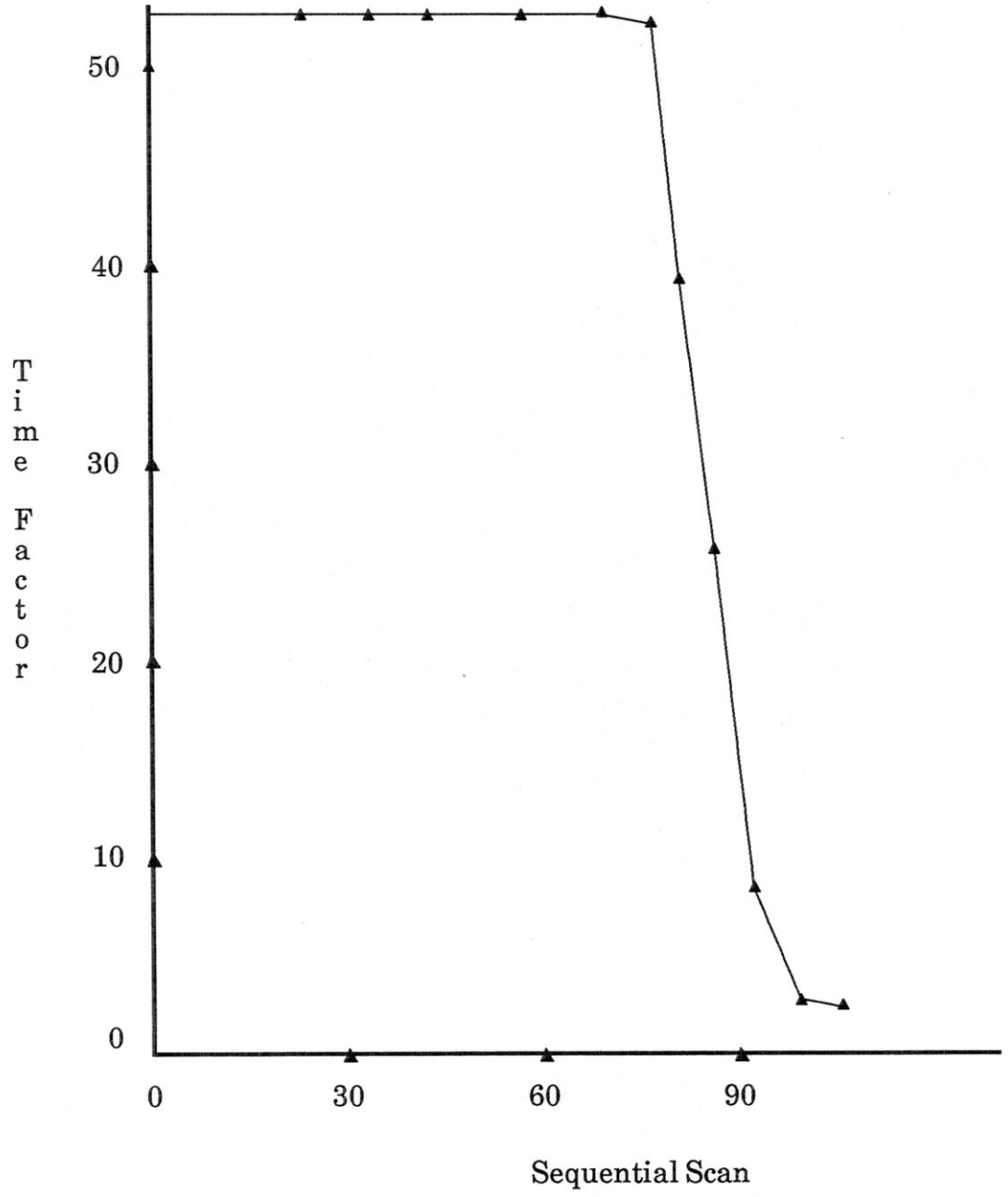


Figure 3

affected by reductions in main memory size. This is because for each expensive disk seek, the program reads all four thousand bytes on the disk page, while for the random probe, the program just reads four bytes from the four thousand byte page that was swapped into memory . Figure five shows how quicksort behaves as memory size is decreased. Quicksort marches linearly down an array with pointers so this program behaves much like the linear scan. Our VLSI layout tool, which is fairly CPU intensive, also pays a big penalty as main memory size is reduced. This is

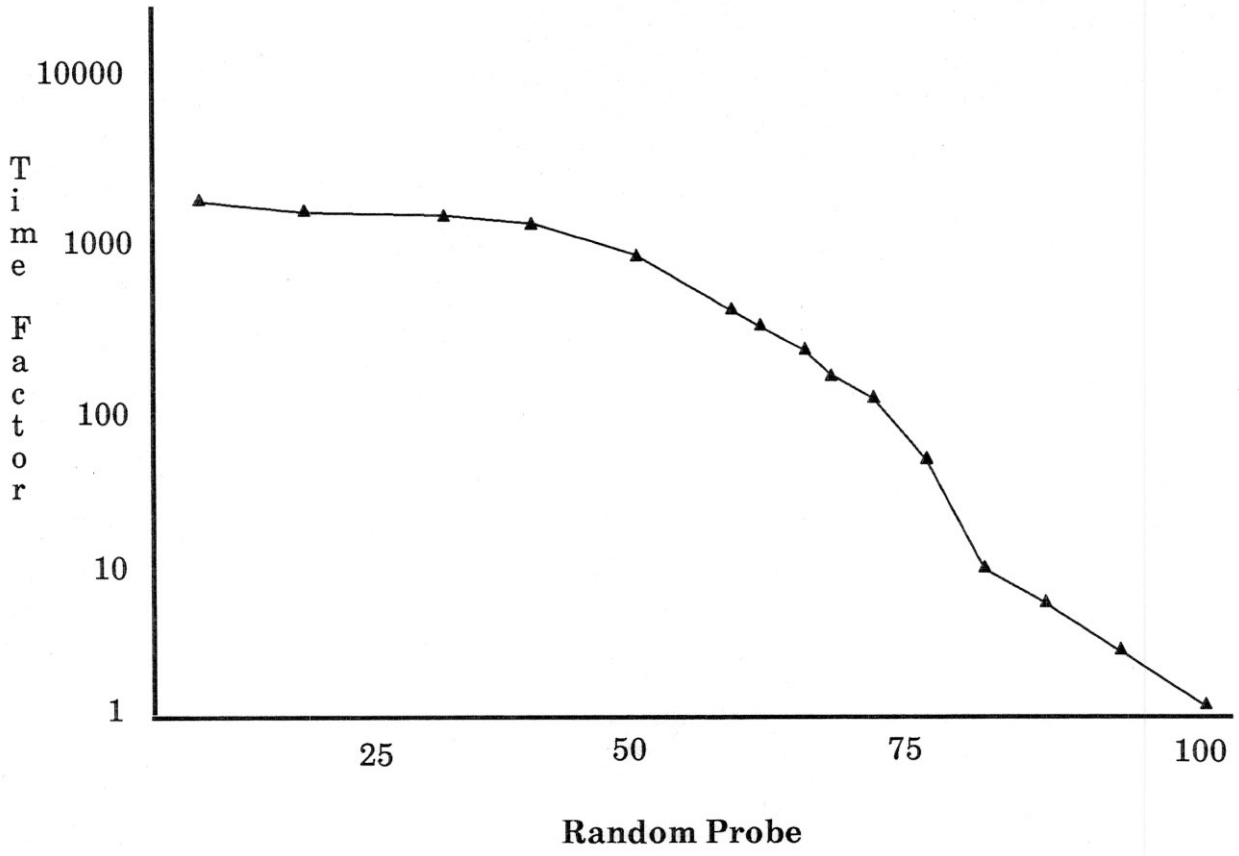


Figure 4 Percent of data in core

because the memory few accesses it performs are relatively random(figure 7). Floyd Warshall references both rows and diagonal of a two dimensional array. As main memory size is decreased, each diagonal access begins to cause a page fault. This leads to two orders of magnitude increase in running times (figure 8).

We have developed a rough metric to measure the memory improvability of a task. We first run a program and provide only enough main memory for half of its data set. We then run the same code and provide enough memory for the entire data set to fit in core. The ratio of these two numbers (running time with half data in core/ running time with all data in core) is a quantity that we call "memory improvability". Table 3 shows memory improvability for a number of tasks. Inherently CPU bound jobs are at the far left end, and they don't benefit from

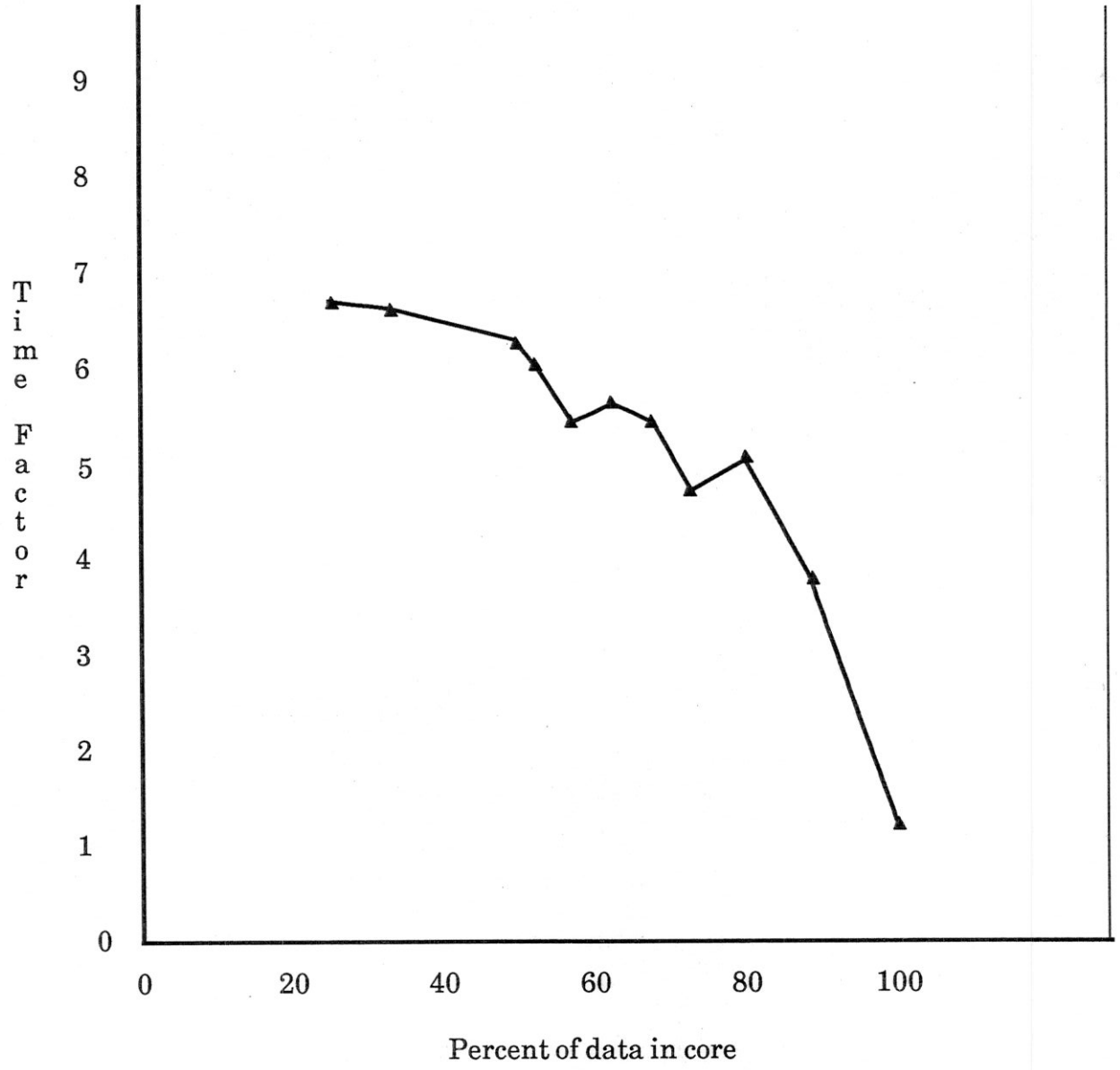


Figure 6 **Quicksort**

massive memory. At the far right end is the completely I/O bound random probe program that benefits an incredibly from massive memory. In between these two extremes many different types of applications fit. There is several orders of magnitude variance in memory improvability between different applications. Note that this classification is algorithm specific. New memory intensive algorithms can be found which will change the location of an application in this hierarchy.

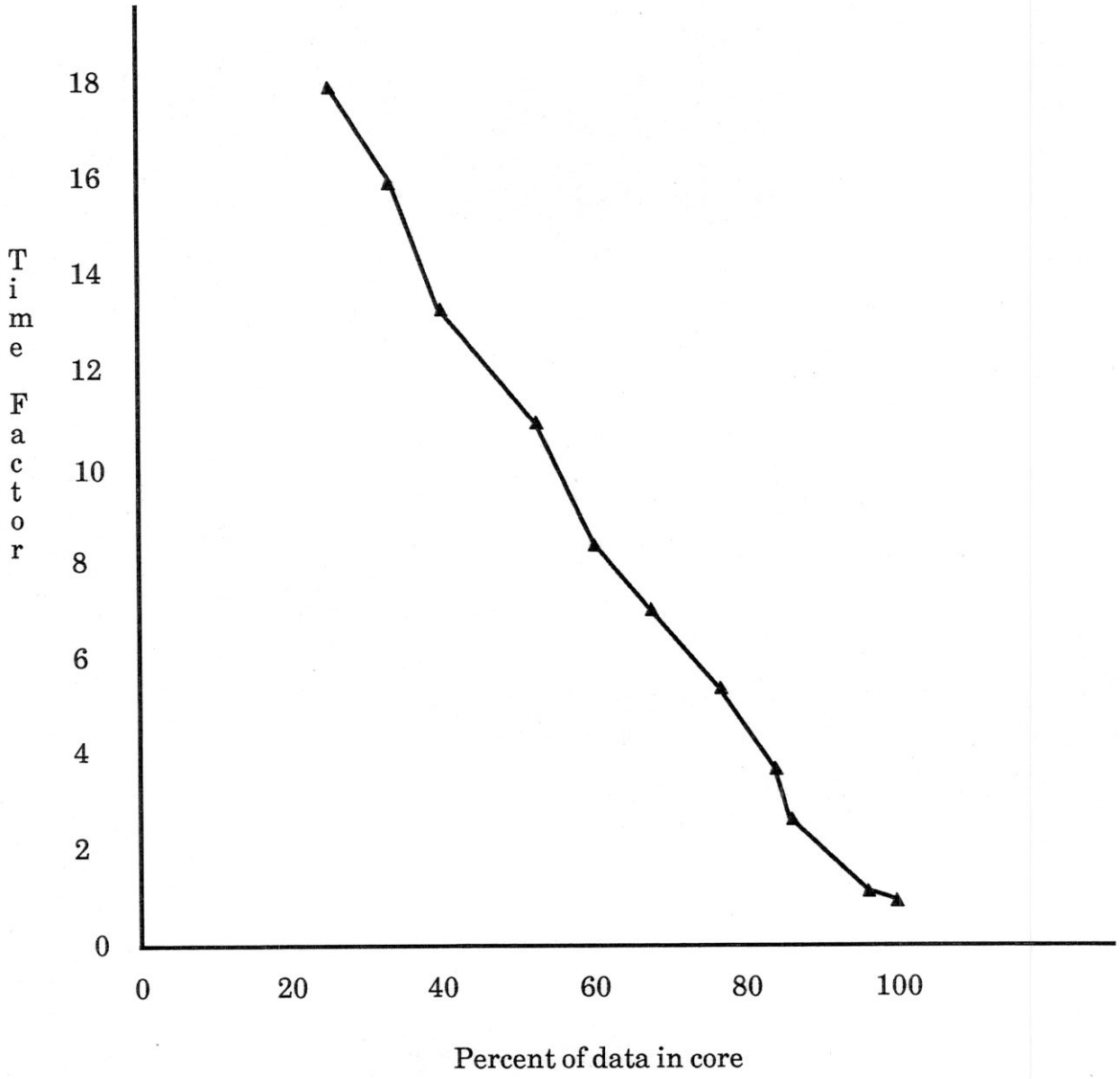


Figure 7

VLSI Layout Tool

Random probe and the sequential scan behave much differently as main memory size is decreased. The scan takes fifty times longer to complete when ninety percent of its data set is swapped out while the random probe takes many thousands of time longer. It is more practical to run a program whose data does not fit entirely in core if the data is accessed sequentially. A ten minute job will take as long as tens of hours if its data set does not fit entirely in core, this is a significant degradation in performance, but the job will complete. A program that performs random probes will take weeks and months to complete if its data set is not entirely core resident.

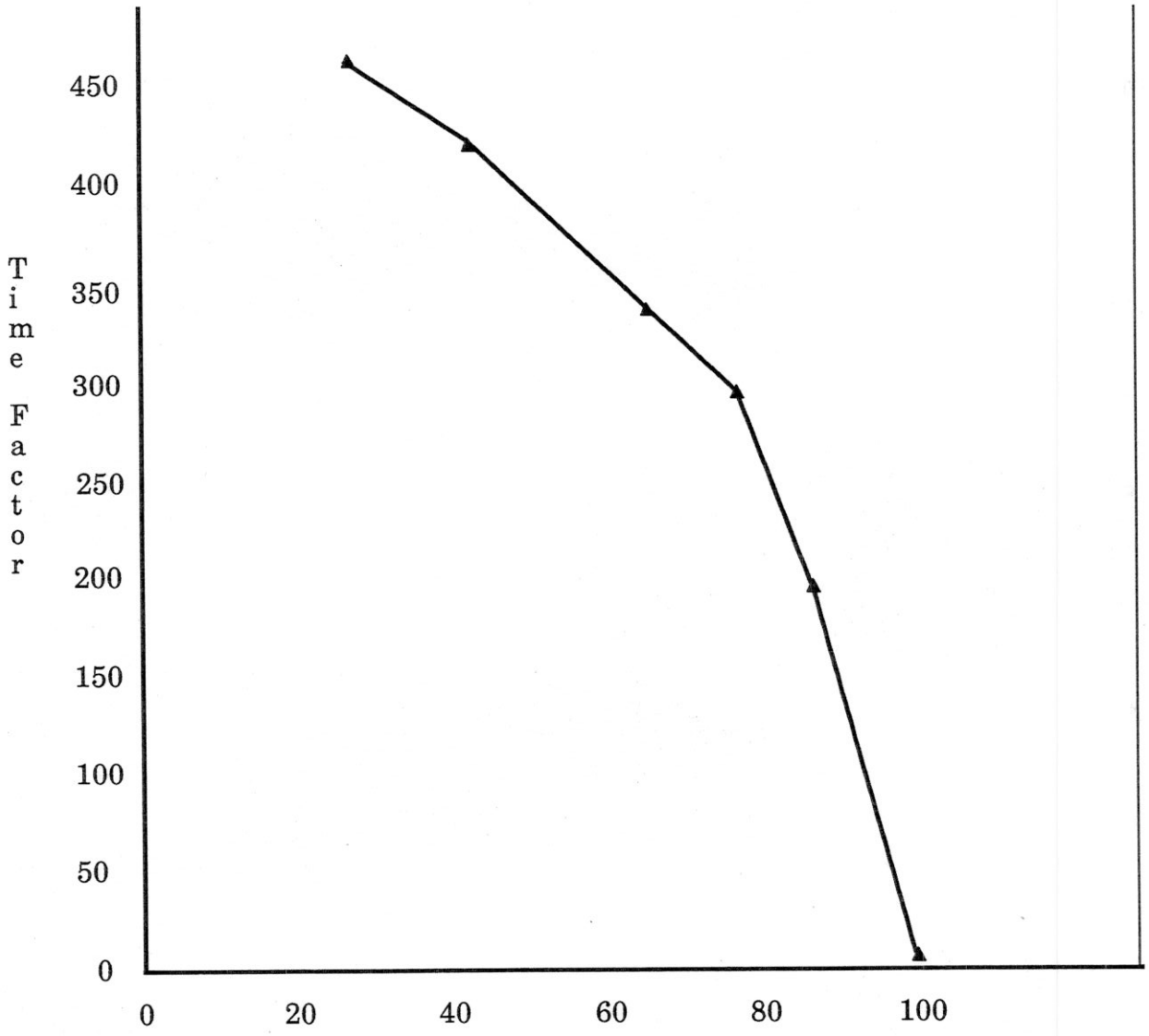


Figure 8

4. Future Research Directions

Many applications will benefit directly from large amounts of memory. But more interesting research lies in exploiting time space tradeoffs to greatly speed up existing applications. We are investigating several application domains. Data bases are an obvious area where massive memory can be used. When data bases are moved

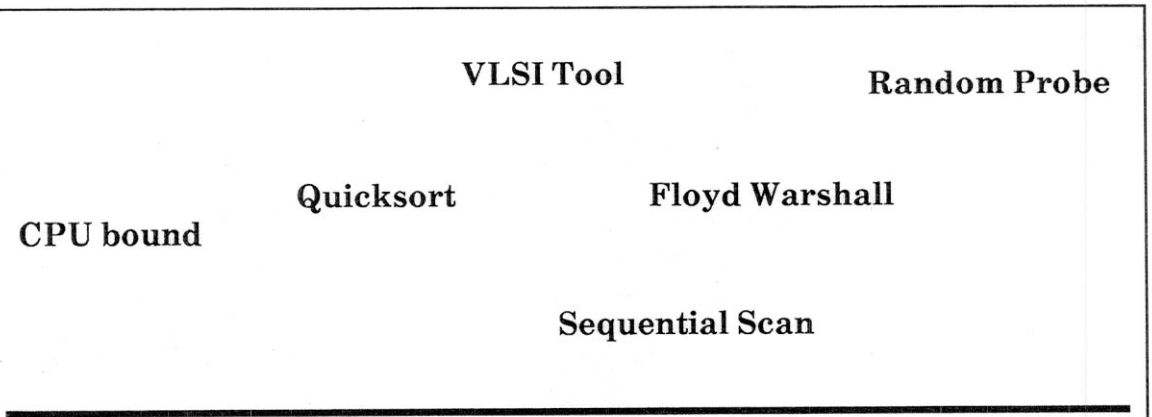


Table 3 **Memory Improvability**

from rotating storage to semiconductor memory, many of the issues change. Transactions can now be processed serially, and efficiently.

To get an estimate of these gains we have compared the UNIX system utility program "dbm" to a core resident perfect hashing program[13]. The results of this test (table 4) show two orders of magnitude performance difference on inserts and retrievals into a 1,000,000 record database.

	1,000,000 insertions	**1,000,000 retrievals**
UNIX version 5 dbm	4025.3 seconds	3803.9 seconds
Perfect Hashing Code	14.283 seconds	13.913 seconds

Table 4 Database Package Timings (4 Byte Records)

Many artificial intelligence applications manipulate huge dictionaries which are referenced in a random manner. Massive memory systems will allow these to operate efficiently.

5. Conclusion

We have demonstrated how massive memory improves computational performance by orders of magnitude on many applications. Some benefit immediately. Our VLSI Layout program, Data base retrieval system, and Quicksort codes have for example. There also exist many memory intensive algorithms that trade space for time (our table lookup functions, and Strip Sort). Little research in computer science has been devoted to studying massive memory computations. This will change. It is economically unavoidable.

Acknowledgments

I'd like to thank Larry Rogers for dealing with some seemingly insurmountable operating systems headaches. I'd like to thank Peter Honeyman, Steve North, and John Sandberg for dozens of helpful suggestions. My office mates K. Balasubramanian, and Panduranga both helped with long hours of C coding. Luen Heng produced a remarkable perfect hashing program.

References

[1] F. G. Withington, "Winners and Losers in the Fifth Generation", Datamation, December, 1983

[2] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, G. Vijayan, "ALI: A Procedural Language to Describe VLSI Layouts", Proceedings of the Nineteenth ACM-IEEE Design Automation Conference, Las Vegas, Nevada, June, 1982, pp. 467-474.

[3] P. J. Downey, R. Sethi, R. E. Tarjan, "Variations on the Common Subexpression Problem", JACM, Volume 27, Number 4, October, 1980, pp.758-771

[4] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[5] J. E. Savage, "Space Time Trade-Offs for Banded Matrix Problems", JACM, Volume 31, Number 2, April, 1984, pp. 422-437.

[6] R.J., Lipton, J. S. Sandberg, S. C. North, "Strip Sort", Proceedings of the Nineteenth International Conference on System Sciences, Volume II, Honolulu, Hawaii, January, 1986.

[7] Special Issue on Highly Parallel Computing, IEEE Computer, January, 1982

[8] Welsh, D., (Molecular Biology Department, Princeton University), Personal Communications

[9] Weicker, R. P., "Dhrystone: A Synthetic Systems Programming Benchmark", Computing Practices, Volume 27, Number 10, October, 1984.

[10] Linser, R., Monolithic Systems, Personal Communications, February, 1986

[11] Linser, R., Monolithic Systems, Personal Communications, February, 1986

[12] Knuth, D. E., Sorting and Searching, 1973, Addison-Wesley, Reading, Mass.

[13] Cormack, G. V., Horspool, R.N.S., Kaiserwerth, M., "Practical Perfect Hashing", The Computer Journal, Volume 28, Number 1, 1985.