To Appear In The Proceeding of the 1987 ACM-SIGMETRICS Conference.

PERFORMANCE THROUGH MEMORY

Hector Garcia-Molina
Arvin Park
Lawrence R. Rogers

CS-TR-075-87

January 1987

PERFORMANCE THROUGH MEMORY

Hector Garcia-Molina
Arvin Park
Lawrence R. Rogers

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

ABSTRACT

Two of the most important parameters of a computer are its processor speed and physical memory size. We study the relationship between these two parameters by experimentally evaluating the intrinsic memory and processor requirements of various applications. We also explore how hardware prices are changing the cost effectiveness of these two resources. Our results indicate that several important applications are "memory-bound," i.e., can benefit more from increased memory than from a faster processor.

January 30, 1987

PERFORMANCE THROUGH MEMORY

Hector Garcia-Molina
Arvin Park
Lawrence R. Rogers

Department of Computer Science
Princeton University
Princeton, New Jersey 08544

1. INTRODUCTION

A computer has two main resources: processor cycles and memory. One of the fundamental questions in computer design is: How much of each resource should a given computer have? One useful way to visualize the tradeoffs between memory size and processor speed is to plot the values of these parameters for commercial computers. Figure 1 shows such a graph, with CPU speed in MIPS (million instructions per second) on the horizontal axis and memory size in megabytes on the vertical one. It turns out that most computers, past and present, lie within a order of magnitude of a straight line with a slope of 1 MIPS per megabyte. This ratio is known as Amdahl's constant[16]. Hence, today's computer designer or user must decide where to be in this two-dimensional space. In particular, he must decide if a new computer should follow past trends and be close to Amdahl's line, or whether it should be in unexplored but potentially beneficial areas of the spectrum.

One of the reasons why this choice is difficult is that we do not *fully* understand why computers have followed Amdahl's trend. A second reason is that memory prices are changing rapidly, possibly upsetting the balance between CPU cycles and memory [1]. For example, (from our own experience) in October 1983 a VAX 11/750 cost 47,000 dollars, or approximately 78,000 dollars per MIPS. In January 1986, a VAX 8600 CPU cost 300,000 dollars , or approximately 75,000 dollars per MIPS [2]. The drop in price was relatively small. During the same period, memory prices fell by almost a factor of 4, as 64K RAMs gave way to 256K RAMs as the volume selling memory chip [8]. The end result is that a megabyte is now selling for between 1000 and 5000 dollars (at

the board level). Hence, it is now tempting to configure a computer with many more megabytes than MIPS, substantially to the left of Amdahl's line.

Of course, microprocessor prices are dropping faster than mainframe prices, so it is also tempting to configure a machine with many microprocessors, achieving a high number of combined MIPS. Such a machine could very well be substantially to the right of Amdahl's line.

The key to understanding the memory-CPU tradeoffs is understanding the applications. For a highly parallel machine to the right of Amdahl's line, it is important to have applications that can be decomposed into parallel tasks and can effectively utilize the available MIPS [4]. Similarly, for a machine with a large memory it is important to identify applications that are *memory-bound*, i.e., that can effectively utilize the memory. (We use the term memory-bound in an informal fashion here. In Section 3 we propose a more formal definition.)

In this paper we focus on this last issue. In particular, we will try to understand what applications would benefit more from a larger memory than from a faster processor. We will also address the problem of measuring the "memory intensity" of programs, and how to factor in the changing hardware prices into a decision.

Two factors play a key role in determining whether a program is memory-bound: locality of memory references and the ratio of memory references to other CPU operations. A program has good locality if it often reads memory locations referenced in the immediate past, or locations physically close together (e.g., on a single page). A program with good locality does not have large physical memory requirements, for it will infrequently have to access secondary storage under demand paging. Similarly, if a program executes many internal CPU instructions (e.g., register to register move) between each memory reference, the need for a large memory is reduced. Our goal here is to identify and characterize applications that have poor locality and a relatively high ratio of memory to internal CPU operations.

When we study the memory references of a program, our main interest will be on the *data* references. Our view is that large memories are not required for holding programs since instruction references have excellent locality. In other

words, a few megabytes (which cost a few thousand dollars at today's board level prices) can probably hold the active parts of the code. So, if an application is going to require a large memory, it will be because it references a *large data structure*, e.g., a database, the representation of a VLSI chip, or a set of inference rules. These large structures are becoming more common as people attempt to solve "large" problems.

It is interesting to note that although instruction references are well understood [9,10,11,12], not as much is known as to how programs reference their data structures. We were only able to find two references in the literature [13,15] that analyzed instruction and data references separately. (We will comment on [13] later.) Data and program refernces tend to be different. Instruction sequence references with their structured sequentiality and standard looping and subroutine constructs tend to be similar over many programs. Data references on the other hand tend to reflect individual algorithms and data structures. For this reason, one can study instruction references in the general case, but data references are best examined in the specific case. This explains why data references are harder to study, and consequently little research has focused on them. In this paper we intend to examine *data* references across a range of algorithms and data structures. Combined instruction and data analysis were important at a time when memory was expensive and paging mechanisms were being developed. Today, we feel it is more important to study data references exclusively.

In this paper we will *not* consider multi-programming. First of all, we wish to study the processor-memory tradeoffs in the simpler context of a single program. Only when we understand this, will we want to move on to a multiprogramming machine. A second reason is that in today's world of personal workstations and network servers, it is becoming more common to run a single program at a time. Hence, our results will be directly applicable in this context.

Finally, we are going to focus on "standard" programs that have not been tuned to improve their locality or their CPU efficiency. Clearly, if one knows his program is going to run on a machine with insufficient memory or

insufficient CPU cycles, one may try to rewrite the code. For example, to cope with insufficient memory one may pre-fetch blocks from disk before they are needed. However, most users (if not all) want to stay away from such tricky optimizations. If they are buying a new machine, it would be desirable to configure it properly, as opposed to programming around deficiencies. Hence, in our measurements, we will use programs taken directly from standard sources. We will return to this point later.

In the following section we start by presenting a simple model that is helpful in understanding the memory-processor tradeoffs. In Section 3 we study how memory impacts real applications, while in Section 4 we focus on the impact of processor and prices on these same codes. Finally, in Section 5 we discuss the results and their implications.

2. A MODEL

In order to better understand the notion of a memory-bound program, in this section we present a simple model of program behavior. This model will also help us interpret the experimental results of the following section.

Consider a program P that accesses a data structure containing V words. Of this virtual space, only M words fit in main memory. Accessing a word in memory takes T_M seconds; servicing a page fault takes T_D seconds. Every time P references memory, there is a probability α that a page fault occurs. Between memory references, P runs on average S seconds of CPU-internal operations. Hence, the expected time P will take from the start of a memory reference to the next one is:

$$R = S + (1-\alpha)\ T_M + \alpha\ (T_M + T_D)$$

or

$$R = S + T_M + \alpha T_D$$

If P executes a total of n references, then P's expected running time is nR. (As

discussed in the introduction, we have ignored instructions fetches.)

If we add sufficient memory to hold P's data structure, then α becomes zero, and the time between references is simply

$$R_O = S + T_M$$

The improvement in total running time due to the added memory is $(nR) \ / \ (nR_O)$, or

$$\frac{R}{R_O} = 1 + \frac{\alpha T_D}{S + T_M}$$

To get an idea of the range of R/R_O values, let us look at two programs which exhibit "extreme" types of data reference patterns. One is a program that probes memory locations at random. In this case,

$$\alpha_r = 1 - \frac{M}{V}$$

The second program is one that scans its V words sequentially. If each block read in from disk contains B words, then

$$\alpha_s = \frac{1}{B}$$

Figure 2 shows R/R_O as a function of M/V (the fraction of P's data structure resident in memory) for the random and sequential programs and for various values of S. We have taken T_M to be one microsecond, T_D to be 10 milliseconds, and B as 1000 words.

The figure shows that there is a wide range of R/R_o values, i.e., that different programs benefit from memory in different ways. If α is large and S is small, it pays to invest in memory. Of course, the next question is: Where do "real" programs fit in this spectrum? The next section addresses this question.

3. MEMORY BENCHMARKS

We evaluated a set of "representative" applications codes to see how they would utilize memory. (We did not study any programs that were obviously CPU-bound.) The tests were run on a VAX 11/785 with 128 megabytes of memory. (Note that the 128 megabyte VAX 11/785 is not a standard product. We purchased a custom made large memory configuration from DEC.) A modified version of ULTRIX 1.1 ‡ allows us to reconfigure the system to utilize only a portion of the memory (between 1 and 128 megabytes). When the usable memory is reconfigured, all system buffers and paging thresholds are scaled proportionately. All I/O was to RA60 disks. Our experiments were run stand alone, with no other processes competing for CPU cycles or memory.

Seven benchmark codes have been run. *Quicksort* is a well known program that sorts a set of four byte integers [7]. The code we used was written by Ken Thompson of Bell Laboratories. The program is run on a random permutation of N integers. (The time to produce the permutation is not counted in the run time. The same is true about setup times for the other benchmarks.) *VLSI Layout* produces a circuit layout for a VLSI chip from a file of cell definitions and constraints [3]. For our experiments, we generated the layout for a control path of a bit-sliced micro-processor. *Floyd-Warshall* is an $O(N^3)$ dynamic programming algorithm to find the shortest paths between all pairs of nodes of a graph [6]. The graph we used was simply a chain, with the first node connected to the second, the second to the third, and so on. All distances were equal. However, note that the data access pattern of our program does not significantly depend on the interconnection. *Matrix Multiply* takes the product of two N by N arrays ($C = AB$) using the $O(N^3)$ method of multiplying one row of A by a column of B to produce a single element of C. Note that the VAX-11/785 has floating point hardware which greatly improves multiplication performance. *Binary Tree* performs a sequence of random word lookups on an English language dictionary which is stored as a balanced binary tree. (The time to generate the balanced binary tree and the random sequence of words is not included in the

‡ ULTRIX 1.1 is similar to Berkeley Unix 4.2

run time.)

The last two programs are included mainly for reference. *Sequential Scan* sequentially reads a fixed size array ten times. The array is initially set to contain zeros (not counted in execution time). This program performs almost no computations between accesses to the array. *Random Probe* accesses words in an array in random order. Instead of computing a random number between probes, we pre-compute an array of random numbers. At execution time these numbers are read sequentially and drive the probing. (Hence, this implementation of the random probe is actually a combination of sequential scan and random probe.) Although both of these programs are interesting as reference points, we believe they also approximate behavior of real codes. For example, a program following a long pointer chain may have very random memory accesses. Similarly, a program that collects statistics or prepares reports may examine large structures sequentially.

To measure performance dependence on memory, one could fix a problem instance and vary the memory size of the computer. In our case this proved to be too time consuming because each memory size change entailed performing a system reboot.

Instead, we varied the size of the problem instance. First, we fixed the main memory size of our computer to the full 128 megabytes and ran a sequence of program with varying data needs. For example, for *Quicksort* the array to sort was of size 16, 24, 32, . . . megabytes. Since the physical memory was larger than the program needs, there was no paging.

Then we fixed the physical memory to a smaller size (e.g., 16 megabytes for *Quicksort*) and ran the same programs again. In this case, only a fraction of the address space fit in memory, so there was paging. For each problem instance, we divided the running time with the small memory by the running time with all data memory resident to obtain a factor equivalent to R/R_O of Section 2. Our results are presented in Figure 3, again as a function of M/V, the fraction of the address space (program and data) that was memory resident. As an example, an R/R_O factor of, say, 10 for $M/V = 0.5$ means that the program ran 10 times faster when its data was memory resident than when it only

had one half of the memory it required.

We should point out that in Unix (i.e., Ultrix) it is very difficult to control precisely how much physical memory a process can utilize. However, from the paging and swapping thresholds that were set, as well as from operating system statistics, we were able to get a rough idea (within 10%) of how much physical memory our program had access to in the small memory machine. Therefore, even though our measure of R/R_O was precise, the value of M/V where it is plotted is approximate.

The results for *Random Probe* and *Sequential Scan* follow the trends we predicted in Section 2. However, notice that for *Sequential Scan* R/R_O does not drop abruptly to 1 when the percent of resident data reaches 100. This is because our computer does not use an exact Least Recently Used strategy for selecting pages to flush out to disk. (In our analysis we assumed that every time a new block was referenced, there was a page fault. Here, it is possible that the next block was not flushed out.)

Like *Sequential Scan, Quicksort* in many cases scans its data sequentially. However, *Quicksort* performs more computations between memory references, so the extra memory does not buy as much. The *VLSI Layout* tool was very CPU intensive, but the probes it did make to memory were very random. Consequently, its performance degraded significantly when main memory size was reduced.

The *Floyd-Warshall Code* accesses both rows and diagonals of a two dimensional array. As the memory size is decreased, each diagonal access caused a page fault, making *Floyd-Warshall* run hundreds of times slower. (A note on our evaluation of *Floyd-Warshall*: The $O(N^3)$ Floyd-Warshall Code takes several days to many months run on a large problem instance. To speed up the benchmarking procedure, we modified the Floyd-Warshall Code so that the outer loop is executed only a constant number of times. This allowed us to perform many tests without altering the fundamental access patterns and paging performance of the program.)

The inner loop of the *Matrix Multiply* program scans one input matrix in row major order and the other in column major order. As memory size is

decreased, the column major order reference pattern will begin to produce page faults. Because the matrix referenced in column order only occupies one third of the total memory used by the three arrays, this page faulting only becomes significant as the available physical memory approaches one third of the data set size. The Matrix Multiply program is also relatively CPU bound, in spite of the floating point hardware. Multiplication operations take more time than simple addition and comparison operations used in the other benchmarks.

The *Binary Tree* program accesses data in a random fashion approximating the behavior of *Random Probe*. However, interior nodes of the search tree (which are the most frequently referenced) are likely to remain core resident. This improves paging performance as can be seen in Figure 3.

Figure 3 illustrates how "memory-bound" a program is, i.e., programs with higher values of R/R_O require memory more badly. However, is some cases it may be more convenient to have a single number that describes the behavior of a program. To this end, we propose the value of R/R_O at $M/V = 0.50$. That is, we define the *memory-bound factor* (MBF) to be

$$MBF = \frac{P_h}{P_a}$$

where P_h is the execution time when half of a program's address space fits in memory and P_a is the execution time when all of the address space fits in memory.

Using the results of Figure 3, we can now linearly order programs by their MBF. This is shown in Figure 4. (Since our measurements of M/V are rough, our MBF are also approximate.) Although this figure does not have all the information we may want, it does illustrate in a simple fashion that programs like *Floyd-Warshall* and *Random Probe* are much more memory-bound than ones like *Quicksort*.

4. PROCESSOR AND MEMORY COMPARISONS

Having studied the effect of memory on performance, the following questions immediately arise. How would the performance of the benchmarks improve if the processor become faster? Which would be a cheaper way to improve performance: buy memory or buy a faster processor? In this section we address these questions.

Suppose we increase the speed of our processor by ΔP MIPS. This effectively reduces parameter C of our model from say C_1 to C_2. If nR_1 is the execution time with the slower processor and nR_2 is the time for the faster processor, then the gain in performance per added MIPS is:

$$G_{MIPS} = \frac{(R_1/R_2)-1}{\Delta P}$$

As we have seen, memory can also improve performance. Let us define the improvement per megabyte to be:

$$G_{MEM} = \frac{(R/R_0)-1}{\Delta M}$$

where ΔM is the number of megabytes of address space that did not fit in memory when the program was run with limited memory.

If each processor MIPS costs C_{MIPS} and each megabyte C_{MEM}, then the improvement per dollar invested is G_{MIPS}/C_{MIPS} and G_{MEM}/C_{MEM} respectively. Dividing these two quantities, we obtain the relative cost effectiveness of adding more memory *versus* adding a faster processor:

$$E = \frac{G_{MEM}/C_{MEM}}{G_{MIPS}/C_{MIPS}} = \frac{(G_{MEM}/G_{MIPS})}{(C_{MEM}/C_{MIPS})}$$

We can rewrite this as

$$E = \frac{G}{C}$$

where G is the ratio of gains G_{MEM}/G_{MIPS} and C is the cost ratio C_{MEM}/C_{MIPS}.

Of course, in reality things are not as simple as this. The cost of MIPS is not linear as we have implied; faster processors usually come with bigger caches and faster disks, so one cannot simply add MIPS; and so on. In spite of this, we feel we can get some useful insights out of this simple model. In particular, we can experimentally evaluate an approximation to G. Then we can plot E as a function of C, and observe how cost effectiveness varies for "reasonable" cost value.

To estimate G, we ran a new series of experiments. For each of our seven applications, we defined a problem instance that required approximately 2 megabytes of user memory. We ran the programs on a VAX 11/750 that had about 1 megabyte available for the program. Then we ran the same program with enough memory to avoid all paging. Since the increase in memory ΔM was 1 megabyte, the ratio of the execution times gives us G_{MEM}.

Next, we ran the same programs on a VAX 11/785 with only 1 megabyte of memory available to the program. Both machines paged off an RA60 disk, so paging times were comparable. (As a matter of fact, the RA60 has a removable pack. We physically moved the pack, containing the operating system and our programs, from one computer to the other.) The memories of both machines have the same access times, and the CPU have similar instruction sets. Hence, moving to the VAX 11/785 is a reasonable approximation to increasing the processor speed only. The 785 is rated at 1.5 MIPS, while the 750 is a 0.6 MIPS machine. (Some of the improved performance of the 785 is due to a bigger cache and a faster bus. However, for our purpose, we feel it is still fair to treat the 785 simply as a faster processor.) Hence, ΔP is 0.9 MIPS. Dividing the ratio of performances by ΔP we obtain our estimate for G_{MIPS}.

Table 1 presents the results of our experiments. The ratios G_{MEM}, G_{MIPS}, and G are shown in Table 2. Note, the wide range of G, indicating that some

programs are more sensitive to CPU cycles, while others are more sensitive to memory.

To view the effect of costs, we have plotted E versus C in Figure 5. We chose as the central C value 0.1 because this represents our costs: We paid 95,000 dollars for the 1.5 MIPS VAX 11/785 processor, and 50,000 dollars for the 0.6 MIPS VAX 11/750 processor. (Since both processors are discontinued, they now cost substantially less.) DEC (Digital Equipment Corporation) currently charges 5,000 dollars for a 1 megabyte board.

Of course, this value of C is just a "starting point." Costs vary widely among manufactures. As we mentioned earlier, costs do not remain constant across all computational speeds. Upgrading a 0.5 MIPS computer to a 1 MIPS model might only require replacement of an inexpensive microprocessor. However, moving from a 3.5 MIPS to a 4 MIPS machine may require expensive new circuit and packaging techniques. Memory prices, on the other hand, are more constant. Yet, there are also non-linearities. For instance, increasing the physical memory of a VAX from 512 to 513 megabytes is extremely expensive, for it requires architectural changes. (Under the VAX architecture, there are only 29 bits available for physical addresses.) Therefore, Figure 5 is only useful for studying trends, not particular values.

It is interesting to note that for all values of C in Figure 5 (actually, for all values of C less than 0.67), all curves except for *Matrix Multiply* are above the $E = 1$ value. This means that for almost all reasonable cost values, it is more cost effective to improve performance via memory. For some of the programs, memory is orders of magnitude more effective for improving performance. (It should be noted that *Matrix Multiply* can benefit more from added memory than our figures indicate. *Matrix Multiply* begins to page significantly when less than one third of its data set is core resident. This is not indicated by our value for G is based on measurements with half the application's data set core resident.)

In the introduction we argued that memory prices are falling faster than processor prices, at least for high performance machines. This represents a leftward shift in the C values of Figure 5. So if this trend continues, memory will dramatically increase in effectiveness over the next few years.

5. DISCUSSION AND CONCLUSIONS

Although our results are limited in several ways, we believe it is safe to reach the following conclusions:

(1) Paging is not as efficient for data as for instruction fetching. Since disks are so much slower than memory, references to virtual memory must be quite localized for paging to work properly. References to code appear to have good enough locality, but in most of the programs we tested, data references did not exhibit good locality. We believe that this is the case for a substantial number of programs. This thrashing caused by data references only becomes apparent when the data structures involved are relatively large.

(2) For a spectrum of applications, memory appears to be substantially more cost effective than processing power. Thus, if one is configuring a computer with a *limited amount of funds*, it may make more sense to first purchase enough memory to hold all of the data programs required, and only then to buy the fastest processor one can afford with the remaining money. This is *not* the way most customers operate today. Customers typically select the processor first, then skimp on the memory. Then they complain that their processor is not fast enough, while in reality their applications are simply thrashing. Of course if funds are not limited, then one may buy additional memory as the need arises.

(3) Computers substantially to the left of Amdahl's line (i.e., with considerably more megabytes than MIPS) may be (or should be) more common in the near future. These machines will be useful for solving large memory-bound problems, e.g., the design of a large VLSI chip. This is not to say that all computers will move in this direction. There are always CPU-bound problems that have small data requirements, and customers who can afford both a large memory *and* a very fast processor.

In the rest of this section we make some additional observations about the memory-processor tradeoffs.

We mentioned in the Introduction that most studies of memory reference patterns have focused on combined instruction and data accesses. When the

data is much larger than the code, data accesses become the dominant factor. The one paper we found that did analyze data references separately [13], supports our contention that many programs are memory-bound. In this paper, D. W. Clark analyzes in detail the data references of three "real" LISP programs: a chemical structure generator, a parser for a speech understanding system, and a program that builds and executes partially ordered plans of action. In summary, Clark discovered that the programs do have substantial data locality of reference. For example, between 85 and 95 percent of the references fall within the most recently accessed page (512 bytes). So even if we only had a single data page at a time in memory, the miss ratio, i.e., the probability of having to fetch the referenced data from secondary storage, would be 0.15 to 0.05.

However, if we assume, as Clark does, that a reference to secondary memory takes about 5000 longer than a reference to primary memory, we clearly see that this "low" miss ratio gives very poor performance. The solution is, of course, to keep more pages in main memory. Unfortunately, the miss ratio decreases slowly as more pages are kept in memory (with a LRU replacement strategy). To obtain a miss ratio of 0.001, 40 percent of the total data space for one program, and 80 percent for another program, must be resident in memory. (The miss ratios of the third program are not reported in the paper.) And because these programs are memory intensive, even this miss ratio of 0.001 slows down the programs by roughly a factor of 6, as compared to a program that had all of its data in memory. This clearly illustrates that for these programs it is more effective to purchase memory to hold a substantial fraction of the data space, than it is to purchase a faster processor.

One of the "limitations" of our study is that we focused exclusively on a uni-processing environment. Another way of stating this is that we studied program response time and ignored throughput (which is only interesting in a multi-processing environment). Clearly, in many situations response time is the dominant factor: a user wishes to sort numbers or design a chip as fast as possible. However, in other cases, a computer multi-processes a large number of short requests, and the response time of each request may not be as significant. For example, in an airline reservation system, reducing the time to make a reservation from 2 seconds to 0.01 seconds may not be important. Throughput,

i.e., how many requests can be processed per second, is more important. (If requests are chained together then, of course, response time is still important. For instance, the difference between 2 seconds and 0.01 seconds for an inference will be important to an expert system that must perform millions of inferences to reach a conclusion.)

If throughput is the main measure of performance, then a small memory may not be as bad. That is, when one request needs data from disk, it can be suspended, and the CPU can work on another request in the meantime. Hence, a faster processor may be more useful than a larger memory.

Nevertheless, the decision may not be as clear cut as it seems. Each page fault or request for data on disk causes CPU overhead: a buffer has to be found for the data, the request must be initiated, and interrupts (with context switches) must be handled. In some systems (e.g., Unix file I/O or any database system) the data must be copied one or more times within memory before the user can access it in his space. Hence, the disk activity will have an impact on CPU utilization and on throughput. Furthermore, running more than one user request concurrently makes them compete for memory, possibly creating many more page faults, and reducing the throughput further.

For example, suppose we purchase a computer to run the chemical structure generator referenced by Clark, and that our performance measure is throughput. Say, we buy a 10 MIPS processor with 10 megabytes. However, assume that each program accesses 100 megabytes of data. To keep the CPU busy, we run say 5 programs concurrently, leaving each program less than 2 megabytes. In all likelihood, the programs will have a very high miss ratio, making the fast processor spend most of its time managing the disk and waiting for data from the disk. In this case, it may have been better to purchase 100 megabytes of memory for a slower processor.

A second limitation of our study is that, as mentioned in the introduction, we did not consider programs that were tuned to cope with small memory. In other words, there are two ways to cope with memory-bound programs. One is to add memory (what we advocated here); the other is to rewrite the program and restructure the data. For example, for our sorting application, we could

have selected another algorithm with better locality (e.g., merge sort). Similarly, we could have written a program that managed the disk itself, as do external sort algorithms. With these approaches it is possible to improve performance without adding more memory.

However, we believe that this is not a desirable alternative in most cases. We feel that forcing programmers to analyze the reference patterns of their programs and to structure their data accordingly is a step in the wrong direction. The cost in terms of programming effort can easily be much higher than the cost of the memory or processor that one is trying to save on. (Sorting may be an exception to this since external sorting is well understood. In this case, we are benefiting from a lot of good work done when programming was "cheap" and computer hardware was not.)

A good analogy can be drawn with virtual memory: it is clear that virtual memory is not necessary if we ask programmers to overlay their programs and data (i.e., have the programs explicitly state what resides in memory when). Furthermore, using overlays can be more efficient than using virtual memory (at least from the point of view of the computer). Yet, since overlays are so painful and difficult to use, we do not see many people who advocate their return.

Furthermore, in some cases it may be difficult to improve performance through programming, even if one were willing to expend the effort. To illustrate, consider a database that contains data on departments and the employees who work in them. If we place the employee records close to (i.e., on the same disk page) as the record for their department, we get good locality when we access a department and its employees. However, if we need a list of all departments, then we have to visit many pages. On the other hand, if we place all department records close together, we can find all departments quickly, but now finding a department and its employees takes longer. In other words, unless we expect a single type of query, it is not possible to improve locality significantly.

Finally, it can be argued that what we really need are tools for *automatically* restructuring data to improve locality. This is certainly a good direction to pursue, but we do not expect such tools, if they ever become available, to be

useful for solving *general* problems. Incidentally, the paper by Clark describes one effort to automatically improve locality. The idea is to periodically compact the used data space, moving all the free space to a single area. Intuitively, it seems that this may help because it increases the density of data and increases the probability that a pointer leads to a nearby page. Unfortunately, as Clark reports, the effort to restructure the data is substantial (it involves traversing the entire data structure, writing it out to disk, and reading it all back in), and the improvements in locality are significant, but not great. (For example, the program that required 80 percent of its data to be resident to achieve a 0.001 miss ratio, now only needs 50 percent of its data resident; the program that required 40 percent, now only requires 35 percent.)

Our results indicate that memory is a very useful resource. Still, there are a number of problems that must be solved before memory can be used in really large quantities. One problem is that today's paging mechanisms are sometimes not designed to handle large physical memories. For instance, we had to perform a few modifications to Unix so that a single process could access the 128 megabytes on our computer. The page replacement algorithm had to be changed because the old one spent too much time scanning our larger than usual page table.

There are also potential problems on the hardware side. The paging hardware on many of today's computers is not designed to manage large memories. For example, on our VAX 11/785, the Translation Lookaside Buffer (TLB) can only map 256 kilobytes [5]. If a data structure is larger than this, memory accesses may cause TLB faults and degrade performance. Thus, our *Random Probe* program ran *twice* as slow when it was accessing a 100 megabyte array (all resident in memory) than when it was accessing a 100 kilobyte array.

Another potential problem is bus delays. As we attach more and more memory to a processor, the bus delays for accessing the memory may grow, decreasing the advantage of having the large memory. Today, there appears to be substantial growth capacity before we hit bus limitations. With 1 megabit chips it is not difficult to configure computers with one or more gigabytes. Furthermore, as densities increase, the amount of memory accessible with a

conventional bus will grow proportionately. Nevertheless, if one wishes to grow memory at a faster rate, or access it very quickly, there may be a need for "unconventional" buses or interconnection schemes [14].

Of course, to go beyond a 4 gigabyte physical memory one must first solve a different problem: address bits. Although processors may use more than 32 bits internally, we know of no processor that outputs more than 32 bits for physical addresses (4 gigabytes). New processors or segmentation registers will be required to go beyond this limit.

Finally, reliability may also be a critical issue. As the memory grows, the probability of an error increases. Furthermore, in some applications (eg. databases), if we do away with disks, we must then make memory non-volatile.

In summary, just like building faster processors or highly parallel machines, building very large memories presents some interesting challenges. Given the current hardware prices and the important memory-bound applications that exist, we believe that these challenges will have to be addressed soon. As a matter of fact, we have recently started a project at Princeton to study solutions for the issues we have outlined here.

6. ACKNOWLEDGEMENTS

We would like to thank Peter Honeyman, Richard Lipton, Steve North, and Jonathan Sandberg for dozens of helpful suggestions. K. Balasubramanian and Panduranga both helped with hours of C coding. 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 N000014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948.

7. REFERENCES

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

[2] Digital Equipment Corporation, Price quotations for Princeton University Computer Science departmental equipment purchases.

[3] 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, pp. 467-474 June, 1982.

[4] Special Issue on Highly Parallel Computing, *IEEE Computer*, January, 1982.

[5] G. W. Gorsline, *Computer Organization Hardware/Software,* Second Edition, 1986, Prentice-Hall, Inc., Edgewood Cliffs, New Jersey.

[6] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM,* Volume 5, Number 6, pp. 345, June, 1962.

[7] C. A. R. Hoare, "Quicksort," *Computer Journal*, 5:1, pp. 10-15, May, 1962.

[8] R. Linser, Monolithic Systems Corporation, Personal Communications.

[9] W. F. Freiberger, U. Grenander, P. D. Sampson, "Patterns in Program References," *IBM Journal Research Development*, Vol. 19, No. 3, May, 1975, pp. 230-243.

[10] S. S. Sisson, M. J. Flynn, "Addressing patterns and memory handling algorithms," *Proc. AFIPS Fall Joint Computer Conference,* Vol. 33, Part 2, December, 1968, San Francisco, CA., pp. 957-967.

[11] A. J. Smith, "Cache Memories," *ACM Computing Surveys,* Vol. 14, No. 3, September, 1982, pp. 473-530.

[12] J. R. Spirn, P. J. Denning, "Experiments with program locality," *Proc. AFIPS Fall Joint Computer Conference,* Vol. 41, Part I, December, 1972, pp. 611-621.

[13] D. W. Clark, "Measurements of Dynamic List Structure Use in Lisp," *IEEE Transactions on Software Engineering*, Vol. SE-5, Num. 1, January 1979, pp. 51-59.

[14] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A Massive Memory Machine," *IEEE Transactions on Computers*, Vol. C-33, Num. 5, May 1984, pp. 391-399.

[15] A. J. Smith, "Cache Evaluation and the Impact of Workload Choice" *The 12th International Symposium on Computer Architecture Proceedings*, June 1985, pp. 64-73.

[16] D. P. Siewiorek, C. G. Bell, A. Newell, Computer Structures: Principles and Examples, McGraw-Hill Book Company, 1982.

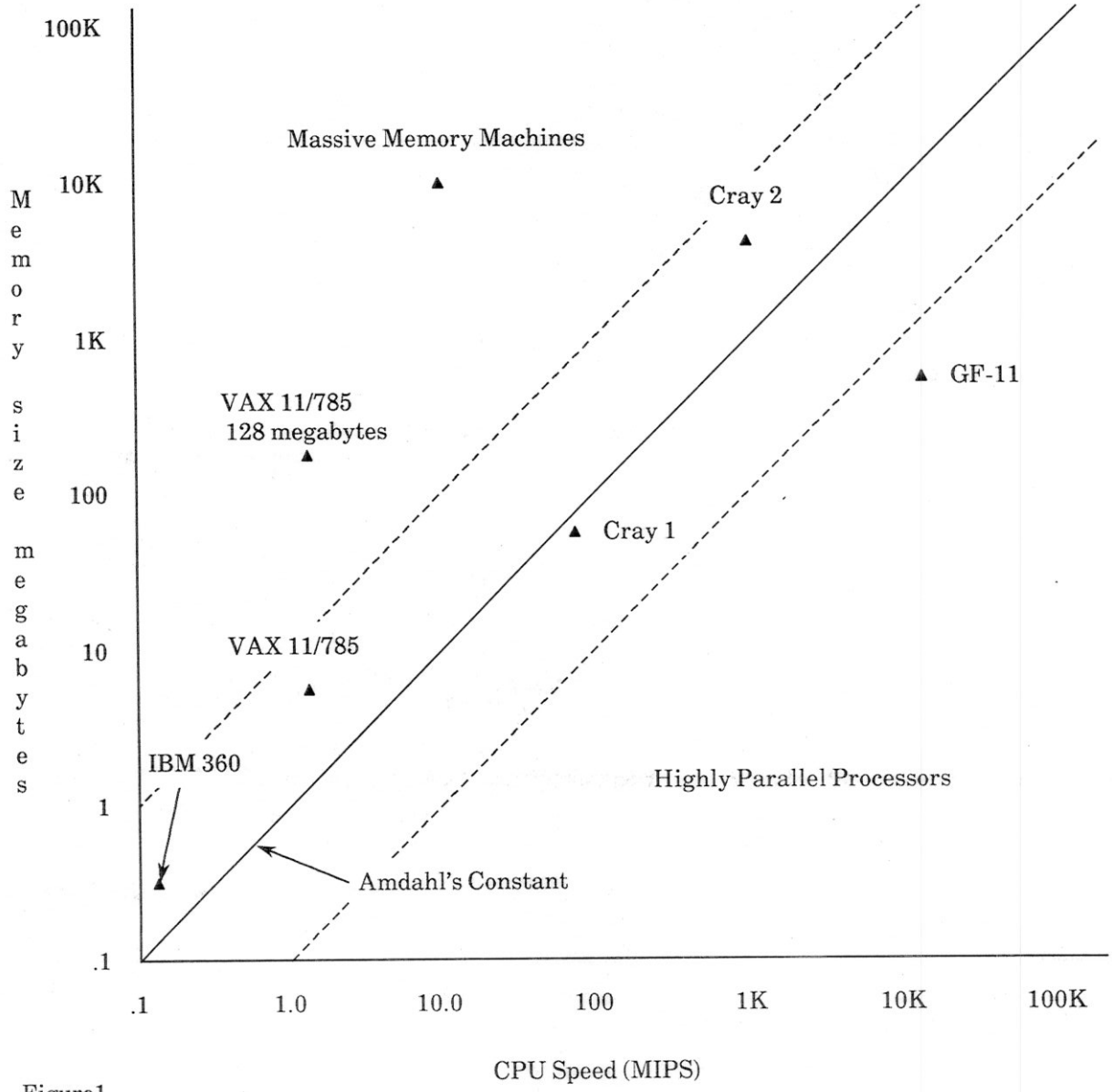
100K
10K
1K
100
10
1
.1
Memory size megabytes
Massive Memory Machines
Cray 2
GF-11
VAX 11/785
128 megabytes
Cray 1
VAX 11/785
IBM 360
Highly Parallel Processors
Amdahl's Constant
.1
1.0
10.0
100
1K
10K
100K
CPU Speed (MIPS)

Figure1

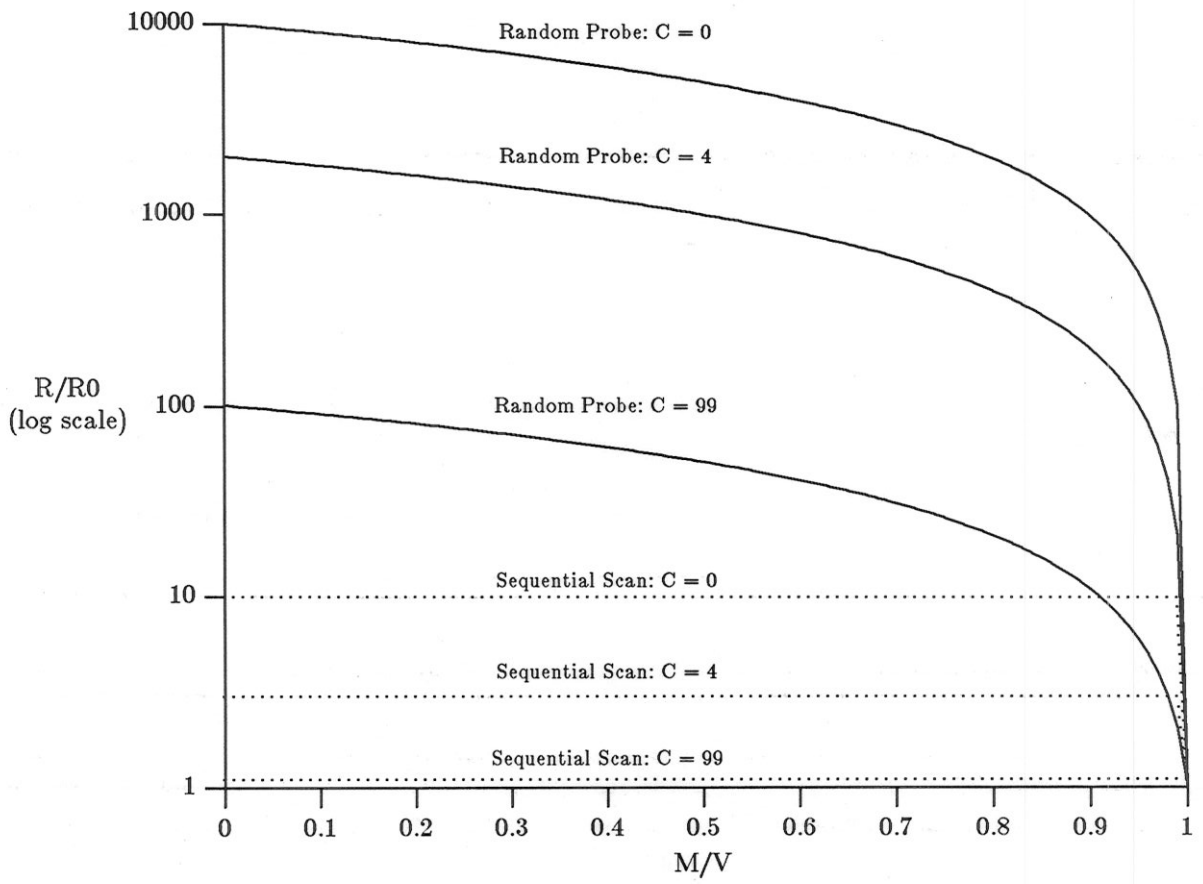
10000
1000
100
10
1
R/R0
(log scale)
Random Probe: C = 0
Random Probe: C = 4
Random Probe: C = 99
Sequential Scan: C = 0
Sequential Scan: C = 4
Sequential Scan: C = 99
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
M/V

Figure 2

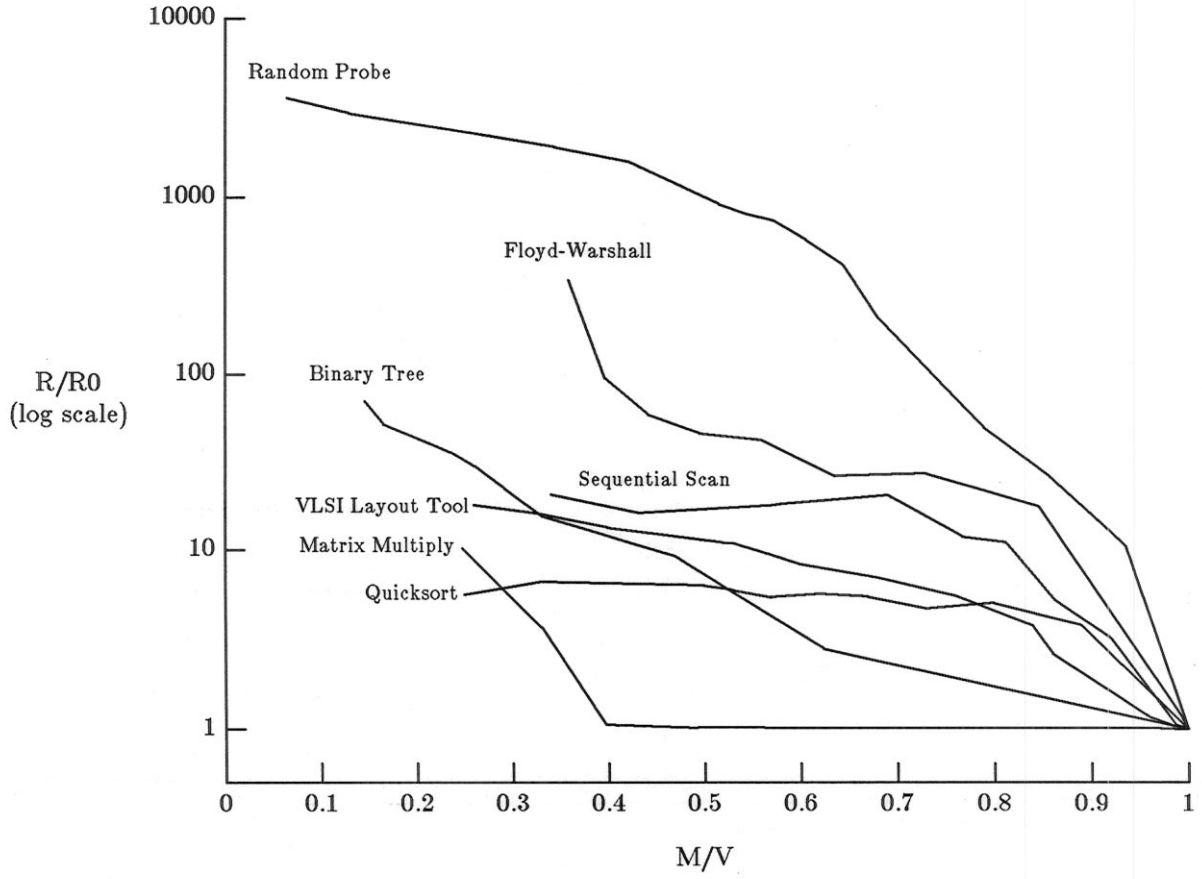
10000
1000
100
10
1
R/R0
(log scale)
Random Probe
Floyd-Warshall
Binary Tree
Sequential Scan
VLSI Layout Tool
Matrix Multiply
Quicksort
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
M/V

Figure 3

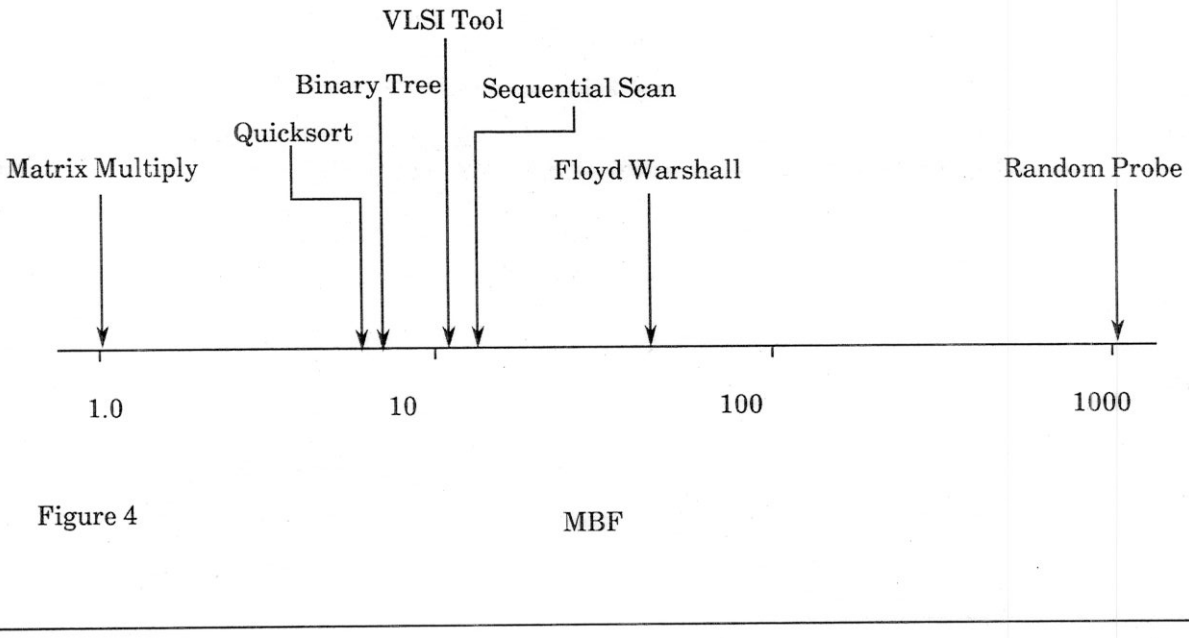
VLSI Tool
Binary Tree
Sequential Scan
Quicksort
Matrix Multiply
Floyd Warshall
Random Probe
1.0
10
100
1000
MBF

Figure 4

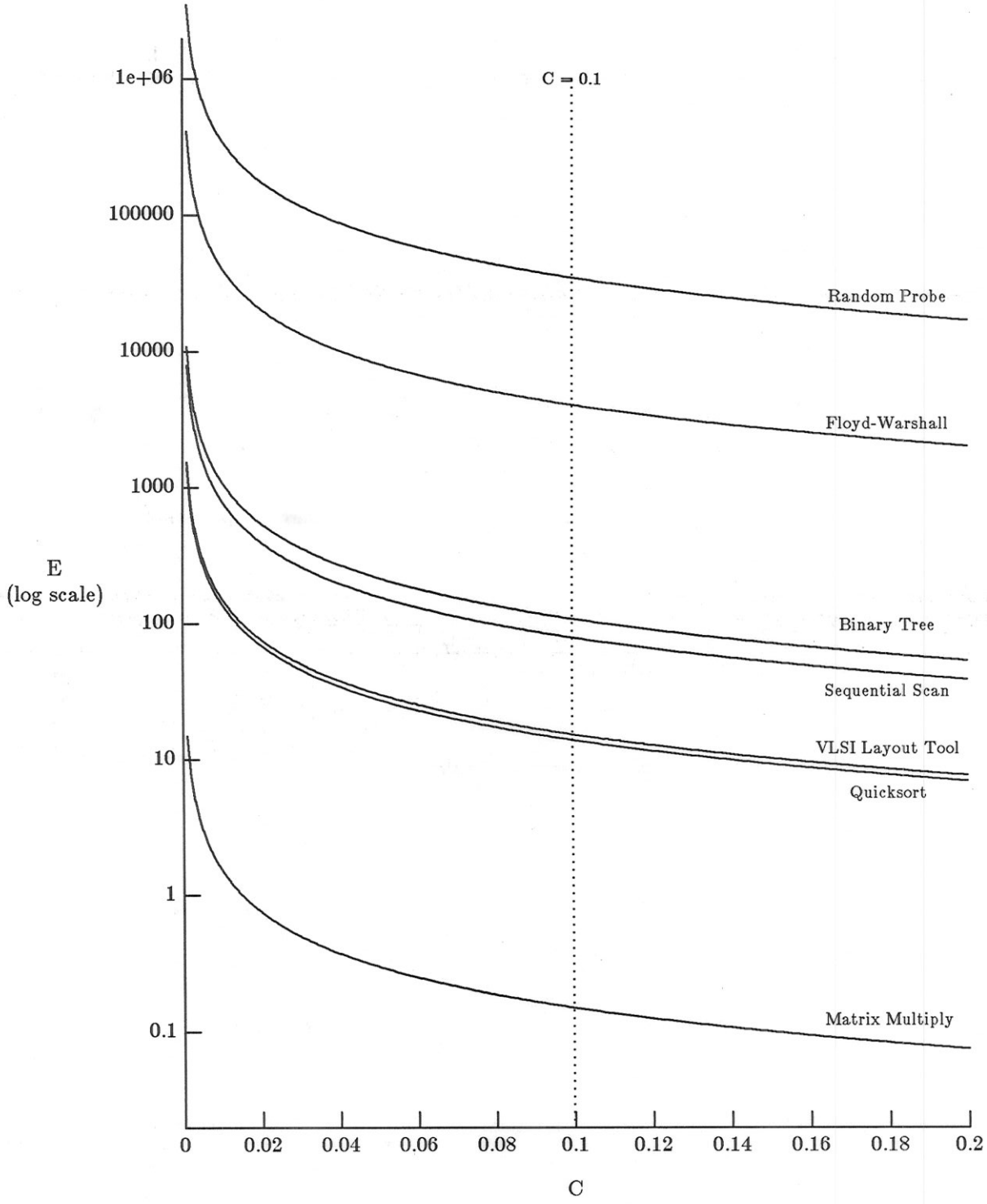
1e+06
100000
10000
1000
100
10
1
0.1
E
(log scale)
C = 0.1
Random Probe
Floyd-Warshall
Binary Tree
Sequential Scan
VLSI Layout Tool
Quicksort
Matrix Multiply
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0.18
0.2
C

Figure 5

Application	Time (sec)	More Memory		Faster CPU	
		Time(sec)	%gain	Time(sec)	%gain
Quicksort	458.4	82.60	455	116.74	292.67
Random Probe	26.7	.02	133400	19.96	33.77
Sequential Scan	65.9	3.01	2089	19.65	235.4
Floyd Warshall	24200.0	223.00	10752	19610.00	23.41
VLSI CAD Tool	408.0	251.20	62.42	299.03	36.44
Matrix Multiply	2199.0	2146.0	2.470	886.0	148.2
Binary Tree	1743.0	320.0	444.7	1277.0	36.49

Table 1

Application	G_{MEM} gain/meg	G_{MIPS}	G_{MEM}/G_{MIPS}
Quicksort	4.550	3.252	1.399
Random Probe	1334	.3752	3555
Sequential Scan	20.89	2.616	7.985
Floyd Warshall	107.5	.2601	413.4
VLSI CAD Tool	.6242	.4049	1.542
Matrix Multiply	.02470	1.647	.01500
Binary Tree	4.447	.4054	10.97

Table 2