Program Page Reference Patterns

Robert B. Hagmann
Robert S. Fabry

Computer Science Division — EECS
University of California
Berkeley, CA 94720

ABSTRACT

This paper describes a set of measurements of the memory reference patterns of some programs. The technique used to obtain these measurements is unusually efficient. The data is presented in graphical form to allow the reader to "see" how the program uses memory. Constant use of a page and sequential access of memory are easily observed. An attempt is made to classify the programs based on their referencing behavior. From this analysis it is hoped that the reader will gain some insights as to the effectiveness of various memory management policies.

1. Introduction

Performance and memory reference patterns of programs is an issue that has been studied for many years. Over time, however, the basic assumptions for the memory management algorithms and the characterization of the programs run under these algorithms can change. This can be due to technological advances or to changes in the workload on the computers. This paper shows the patterns of reference of several programs. Some of the programs are large in that they consume much time and memory. Other programs are included because they are used frequently in the environment being considered. From this study it is hoped that the reader can get a better insight into how large programs may reference memory. Of particular interest are the results that the influence of data pages far exceeds that of code pages for large programs, and that radical changes in program locality tend to be quite infrequent.

The motivation for this work was the desire to understand the impact on memory management due to the increase in size of applications running on computers. Examples of these applications are VLSI design aids, image processing and symbolic computation. These programs are much too large to study by simulation: they may run for a day and use many megabytes of memory.

The tracing technique works as follows. Periodically, the operating system invalidates all page table entries of a user process being traced. The operating system records the information about each page fault which occurs in an internal buffer. A second user process copies data from the buffer to secondary storage. After tracing, the faulted page would then be validated or paged in as appropriate. Provided that the period between invalidations is chosen appropriately, the great majority of instructions do not fault and hence the measured process is executed at nearly full speed. A slow down factor in the program's execution of about two to four is typical. This speed is vastly better than that is achieved by conventional simulation. The record on secondary storage gives all pages touched by the process during the period between invalidations. This method is somewhat similar to the typical cooperation between a debugger and the operating system in setting and trapping breakpoints: the processor runs at full speed until something of interest happens.

To gain insight, a series of studies was performed to detect how real programs use memory. The principle results presented here are strip charts showing page reference behavior on one axis and time on the other axis. From these it is possible to see in visual form how the sampled programs reference memory. The usefulness of many memory management policies can be evaluated by simply looking at the charts.

The plotting of page references versus time is not new. In Chu and Opderbeck's paper [Chu76], strip charts are presented for four programs. Burgevin and Leroudier [Bur76] showed a strip chart for a program. Lau [Lau79] shows strip charts both before and after restructuring programs. Alanko, Haikala and Kutvonen [Ala80] also present strip charts.

This research was supported in part by the Defense Advanced Research Projects Agency (DoD) ARPA Order no. 4031 monitored by Naval Electronic Systems Command under Contract No. N00039-80-K-0649.

These studies used instrumented simulators, however, and dealt only with much smaller programs executing for much shorted periods of time than those reported here. The previous papers were not content to analyze the strip charts alone, but proceeded to process the data further.

Four basic classifications of access patterns have been identified. Each typifies the access of a segment of memory during a phase of execution of a program. An access is called *total* if nearly every page in its logical address space is touched frequently. An access is *sequential* if the pages are used in an ascending or descending sequence. This kind of program would benefit from having an operating system prefetch pages to be needed soon and pre-release pages that have been passed. *Regular* access is characterized by conventional locality characteristics. Algorithms like working set, LRU and clock are usually effective. Finally, an access is *random* if the memory reference sequence is nearly memoryless (in the statistical sense). Although some synthetic programs were measured that exhibited a random access pattern, those results are not reported here because this study focused on real programs.

To be classified as belonging to one of these types, a program must exhibit the corresponding behavior over a significant portion of memory for a significant period of time. The scale of significance here is that which would help an operating system in scheduling memory. A sequential scan of a few thousand bytes taking a few tens of milliseconds is an example of a characteristic well below the level of significance.

To get a grasp on what are times that are significant to a typical time-shared operating system, the departmental research machine was examined. This is viewed as a lower bound on the minimal resources given to a user. It is typical to get 3% of the processor during normal working hours. Each active user's fair share of memory is nearly 200 kilobytes. A page of memory is not normally paged out sooner than 30 seconds of real time from its last reference. Hence, about a second of virtual processor time is about the right granularity for the operating system when handling a non-interactive process. This corresponds to a working set τ of about a million references. These values are very conservative: the research community in general and the department in particular have or expect to have in the near future significantly more processor power and memory to be available to the individual users.

The other bound on the resources is for use of a dedicated machine. A dedicated machine is common where some of these applications are run. Instead of 3% of the processor, the user gets the whole machine. The 30 second residency of a page is nearly an invariant since it is determined by secondary storage access time and operating system overhead. Hence, the τ here is more like 30 million references.

The above classification scheme attempts to grasp the essential character of a program. For example, a program may be viewed as being total even though a small number of its pages are never used. This criterion in classification is reasonable in view of the changing technology of the last few years. Memories are becoming larger and less costly, while the access time to secondary storage has been mostly constant. Hence, if some processor or I/O time can be saved, it is worth the waste of a few pages of memory.

All the programs observed are real applications. In some runs, the data processed by the program is synthetic, but only where the access patterns are not data dependent. Some programs are large by most standards: they use up to 36 megabytes of data and could run for a day or more on a dedicated machine. One is a frequently used program: the parser-code generator, optimizer and loader for a compiler. The programs were not selected to be typical of a process mix on a normal time-shared computer.

This paper has three more sections. The next section describes briefly the hardware, the operating system, and the technique used to capture the data. It may be skipped by readers not interested in these details. Section 3 provides the results and an analysis of what they mean. The last section is for the conclusions.

2. Technical Details of the Data Collection

This work is part of a project to upgrade a Berkeley version of the UNIX † Operating System (VMUNIX) [Bab81]. VMUNIX is an extension of Bell Lab's UNIX/32V. It runs on the Digital Equipment Corporation VAX-11 series computers ‡. The measurements presented in this paper were done on a dedicated VAX-11/750.

The measurement project included three parts: kernel modifications, the development of a user process to move data from buffers in the kernel to secondary storage, and the modifications to the programs to be measured. Each of these will be described in general terms below. The description is intended to serve as a guide to those who may want to implement this technique on their own system. All that is really needed to port this technique to another machine and operating system is paging hardware and sufficient cooperation between the operating system and the processes. Of course, a plotting device is also required, but the growing availability of bit addressable hard copy devices, sometimes called typesetting equipment, means that many sites now have plotters.

There were two principal kernel modifications. The first change was to extract trace data at the time of a page fault. This data was placed in a circular buffer in the kernel address space. A second modification was to allow a process to issue a system call that invalidated all of its page table entries.

The second part of the measurement tool was the monitoring process. Its primary function was to copy the trace data onto permanent storage. It would periodically read the kernel buffer, block it and write it to tape (or disk).

† UNIX is a trademark of Bell Laboratories.
‡ VAX and PDP are a trademarks of Digital Equipment Corporation

No processing of the data was performed.

The final component of the tracing package was the program to be monitored. The traced program was usually a single process. The program source was modified, by the insertion of a one-line call, to execute a routine to setup the measurement tool. At a roughly periodic rate in real time, the monitored process would be interrupted. Software was added, by inclusion of some object code at load time, to handle this interrupt. This software issued the system call that caused all pages of the process to be invalidated. The first reference to each program or data page after this would cause a fault. The fault would be traced, and the page would either be paged in or the page table entry would simply be validated. Normally, only a single fault for a page would occur between periodic interrupts. After some analysis, the period between interrupts was set at 100 milliseconds.

In summary, the pages touched by the monitored process in each 100 milliseconds period of real time would be recorded.

Several minor operating system specific comments may be helpful for a reader already familiar with VMUNIX. First, the system call that invalidated the pages was an extension of the "vadvise" system call. Second, the minimum period of software generated interrupts was one second. This was modified to allow interrupts to occur with the granularity of the line frequency (60 Hz on this machine). Third, a process was allowed to "time stamp" the trace buffer. This was used to record virtual time in the buffer so as to be able to convert the trace data to virtual time. Finally, system calls were added to allow the tracing to be turned on and off.

In any measurement study, overhead is always a problem. The goal is this work was to allow substantial overhead, but not to let the overhead dominate. Most processes took about two to four times the execution time they used when not traced. This could have been substantially lessened by invalidating their pages every 100 milliseconds of virtual (instead of real) time, or by increasing the real time between invalidations.

Other trace collection efforts in this area have used simulators (machine interpreters) instead of the raw hardware augmented by operating system support [Chu76] [Bur76] [Ala80] [Lau79]. Simulators typically impose a slowdown factor of between 20 and 100. The raw data they produce has a much smaller granularity and a greater volume. The fine detail is not needed for studies that focus on the operating system.

3. Description of the Programs Measured and Analysis of the Trace Data

More than a dozen programs were traced. The five most interesting of these are presented below. These programs were selected primarily because they were either very demanding on the memory of the machine or frequently used.

Once the trace data was available, a series of data reduction programs was written. These included simulators of FIFO, LRU, Belady's MIN algorithm [Bel66], clustered page-in extensions of the above and sequential detection prepaging policies. All of the above were found to be unsatisfactory analysis tools because of the number of parameters necessary to run a simulation. A cloud of numbers was generated. They provided very little insight into how a program used memory.

The method of analysis selected is to present the page reference pattern in the visual form of a strip chart. Each scan line represents 100 milliseconds of virtual time Time runs down the page. The other axis is the page number. Pages are 1024 bytes in size. Each dot represents one or more references for a page during a 100 millisecond period. References to the stack are not shown. In some cases, only a fragment of the chart is presented. The charts are not all at the same magnification.

In the analysis of the programs, all conclusions are made from knowing the general application of the program and from examination of the chart. They have not been verified by examination of the programs. This is an advantage of this technique: the analysis does not require detailed knowledge of the programs.

Figure 1 represents IMAGE (called exyiq by its author). This is a program that demonstrates certain features of image processing. It converts a red-green-blue style image to a y-i-q style image. It is written in C and has 15 pages of program (as in all the programs examined, the program pages precede data pages in virtual memory). First of all, notice how little of the address space is used for the code (less than one percent). Second, the data seems quite well organized for this type of processing. The working set is small compared to the total amount of data. What is apparently shown here is a technique commonly used when processing two (or more) dimensional, non-sparse matrices [McK69]. The matrix is subdivided in submatrices by cutting the matrix vertically and horizontally. By adjusting the size of the submatrices, a row or a column of the original matrix can be completely resident in memory by only loading the submatrices needed. Thus a matrix can be processed in either column or row order on a machine with a physical memory much smaller than the entire matrix. The author clearly had memory performance in mind when this program was coded. On the chart, observe that there are sharp changes in the locality of the data, but not in that of the code. There are six segments of data that exhibit sequential behavior, while the procedure segment is total.

Figures 2 and 3 show a trace of a Fast Fourier Transform (FFT) program. The first eighty seconds are used to initialize the matrix. During the actual transform, the initial phases are apparently sequential. However, the period is so fast that paging is impractical. This is a technology influenced decision: faster paging devices and slower processors would make some of this sequential access profitable to exploit. At about three minutes into the execution, the working set drops to about half of the pages. However, there is a transition where all

data pages previously required are no longer needed and all the unneeded pages are now needed (eg. all the even pages are no longer needed but the odd numbered pages are all now needed). The only reasonable way to run this program is to give it all the memory it wants. The program is (probably) total after the initialization phase.

Figure 4 is for VAXIMA. This is a VAX extension of MACSYMA running a demonstration script. This program does symbolic computation (eg. integration, equation solving). It is written primarily in LISP, but has some C primitives. Because of the list structure storage of LISP, there is no distinction between code and data. It is easy to observe that there is some sequential behavior in the upper addresses. Also, many pages are in nearly constant use. However, the outlying points here should dominate performance. Each of them represents a potential fault for a memory management policy. This figure illustrates a potential problem for the visual analysis technique: small outlying dots tend to be ignored. VAXIMA is regular except during garbage collection (seconds 21-25) when it is total for certain segments.

Figures 5 and 6 are for SEARCH. This is a FORTRAN program that searches deep space photographic plates for stars and galaxies. It has 42 pages of program. Figure 5 shows the first 40 seconds. The pattern established in the fifth second continues up through the fourteenth minute shown in Figure 6. The program runs for about six hours loosely repeating the pattern shown in minutes fifteen through seventeen. It is doing software paging of a 36 megabyte disk file. Note that, even though a definite low to high address sweep is visible in Figure 6, virtually all pages are used every few seconds. The horizontal lines apparently occur when some feature is detected in the picture and must be evaluated. This is another program that requires that all its pages be in memory for efficiency. It could also be argued that this program is regular since the periods of total use of memory are separated by a few seconds. Again, this is a technology influenced distinction.

This addressing pattern is an artifact of the operating system primitives provided by VMUNIX. Ideally, the data file would be coupled into the address space instead of using software paging. If files were coupled, then this program would have sequential behavior for one or two segments.

Figures 7, 8 and 9 show three processes of the "C Compiler". The three processes shown are the only ones that use significant amounts of time. It should be emphasized that the structure of this compiler is an artifact of its creation: it was built for the PDP-11 computer which had a very limited address space.

The lexical analyzer through code generator part of the compiler is CCOM. Its plot is shown in Figure 7. It uses 70 pages of program. Although there are sweeps through small sections of memory here, the period is short. It may be too short a time period to be useful to an operating system. This trace is for a particularly large compilation of a program that is about 1000 lines long. Smaller programs benefit even less by optimizations based on this sequentiality. CCOM should be considered total.

The code optimizer is C2. Its plot is shown in Figure 8. It uses 23 pages of program. Although part of the memory is used sequentially, the period of use is around a second. This compilation was performed on the same program used for CCOM. C2 must be viewed as total, but the size of the segment changes.

The VMUNIX loader is LD. Its plot is shown in Figure 9. Note that there are two phases. This plot is for the loading of the VMUNIX Operating System itself, and uses more memory and runs longer than a typical use of LD. Although some sequentiality exists, the program seems mostly total.

Notice how densely the parts of a "C Compile" (CCOM, C2 and LD) use the address space. Since they are total, for these processes swapping would be more effective than paging.

4. Conclusions

This paper has presented a technique to study the memory reference behavior of programs. The technique is fairly simple to implement and is vastly more efficient than methods normally used to gather memory reference data. This data is only suited to studies of paging or higher level memory management, and not suited for studies of cache behavior.

Data references appear to significantly dominate the paging behavior of large programs. By comparison, the code references tend to be to a small area of memory and tend to cover this area very densely. Since data references dominate the paging behavior, this implies that restructuring efforts should focus on data restructuring and not on code restructuring [Fer76].

The only predictive memory management policy that was sometimes found to be useful was that of sequential access. Of the programs measured, only IMAGE and SEARCH (with files coupled into the address space) would be amenable to prediction. IMAGE could be classified as a multi-segment sequential program. What is impressive is the data restructuring already used by IMAGE and SEARCH. Both apparently use the division of a matrix into submatrices to reduce the working set. The other programs basically need all pages in memory to get good performance. Some reduction in paging might occur if larger pages or clustering of pages were used.

A major result is the absence of locality changes in the reference patterns for the program pages (as opposed to the data pages). IMAGE, FFT and SEARCH seem to have no significant change in code locality. They do have changes in data locality. VAXIMA appears to change locality only during garbage collection. LD and C2 exhibit only a small change in locality. CCOM has an initial phase that lasts for about five seconds, but after this there is no significant change. The conclusion is that with the larger time granularity appropriate to modern paging operating systems, changes in code locality appear to be insignificant. It should be noted that the programs measured

here had not been structured to maximize code locality, but an examination of the charts suggest that the payoff would not have been large.

Ideally, a clever and simple policy should be devised that would detect all the characteristics discussed above and manage memory properly. Failing that, it is desirable to identify primitives that can be provided by the operating system to allow a program to hint at its expected behavior. The standard VMUNIX Operating System provided only one such opportunity at the time of this study: a hint could be given that the behavior would be anomalous and page replacement should become random. This was very effectively used by LISP during garbage collection. An experimental version of VMUNIX now also allows the hint that a program is about to exhibit sequential behavior.

Based on the data presented here, the following seems to be a good choice for primitives. Programs may be regular (have some locality), random (no locality), sequential (ascending or descending) or total (very inefficient to run unless all pages can be memory resident). The whole program or only a segment of the address space or only a phase in the program's lifetime may exhibit the hinted behavior. Operating systems should be equipped to handle hints from programs as to the type of their memory referencing behavior. It is expected that such hints would be provided only for highly used programs and for programs with special performance problems.

5. Acknowledgements

This work was started as a joint class project with one of the authors and Douglas Terry. William Joy did the kernel modifications and provided much valuable advice. Eric Cooper also assisted in the kernel work. Professor Domenico Ferrari provided guidance throughout this investigation and reviewed early copies of this paper.

Special thanks go to Steven Shafer (IMAGE) and John Jarvis (SEARCH) who contributed software to be measured.

6. References

[Ala80] T. O. Alanko, I. J. Haikala and P. H. Kutvonen, *Methodology and Empirical Results of Programs Behaviour Measurements*, Proceedings of Performance 80, in Performance Evaluation Review, Vol. 9, No. 2 (Summer 1980), 55-66.

[Bab81] O. Babaoglu and W. N. Joy, *Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Reference Bits*, Eighth Symposium on Operating System Principles, in Operating Systems Review, Vol. 15, No. 5 (Dec. 1981), 78-86.

[Bel66] L. A. Belady *A Study of Replacement Algorithms for Virtual Memory Computers*, IBM System Journal, Vol. 5, No. 2 (1966), 78-101.

[Bur76] P. Burgevin and J. Leroudier, *Characteristics and Models of Program Behavior*, ACM '76, Proceedings of the Annual Conference (1976), 344-350.

[Chu76] W. W. Chu and H. Opderbeck, *Program Behavior and the Page-Fault-Frequency Replacement Algorithm*, Computer, Vol. 9, No. 11 (Nov. 1976), 29-38.

[Fer76] D. Ferrari, *The Improvement of Program Behavior*, Computer, Vol. 9, No. 11 (Nov. 76) 39-47.

[Jar81] J. F. Jarvis and J. A. Tyson, *FOCAS: Faint Object Classification and Analysis System*, The Astronomical Journal, Vol. 86, No.3, (March, 1981), 476-495.

[Lau79] E. J. Lau, *Performance Improvement of Virtual Memory Systems by Restructuring and Prefetching*, Ph.D. dissertation, University of California, Berkeley, 1979.

[McK69] A. C. McKellar and E. G. Coffman, Jr., *Organizing Matrices and Matrix Operations for Paged Memory Systems*, CACM Vol. 12, No. 2 (March, 1969), 153-165.

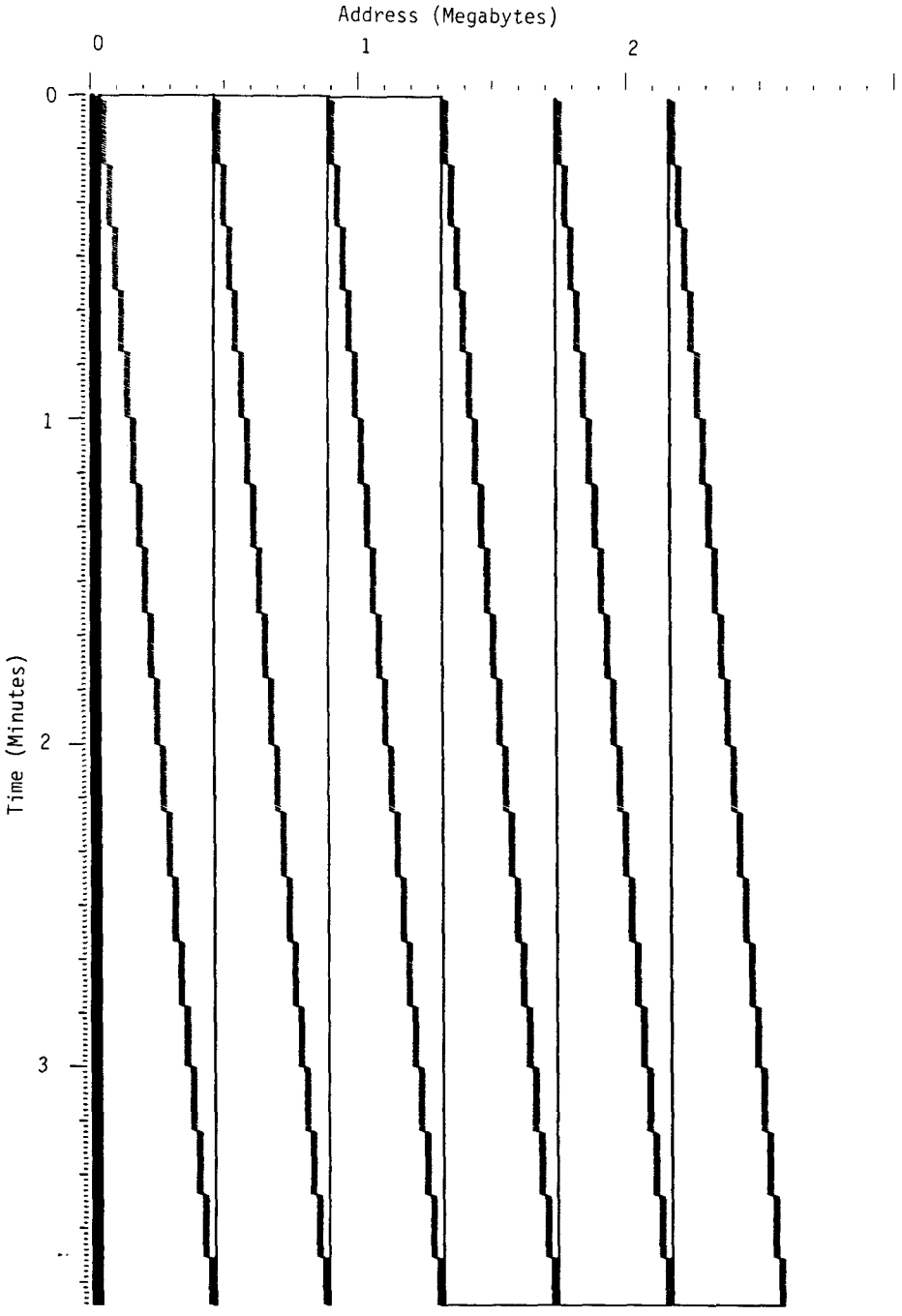

Figure 1: IMAGE

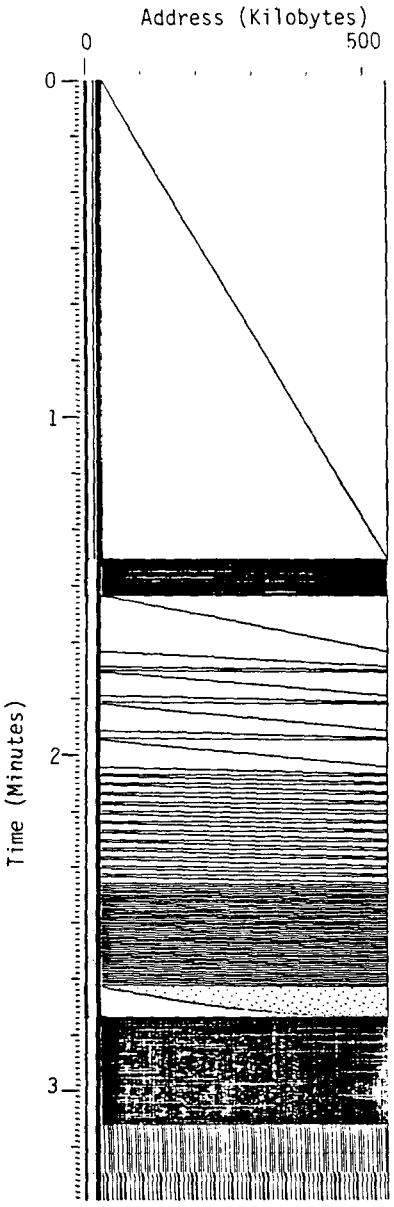

Figure 2: FFT Part 1

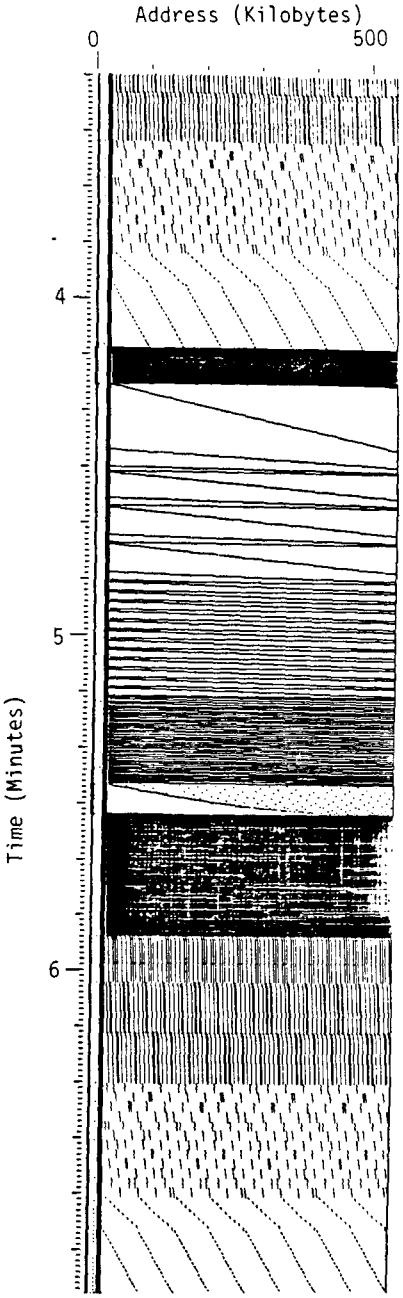

Figure 3: FFT Part 2

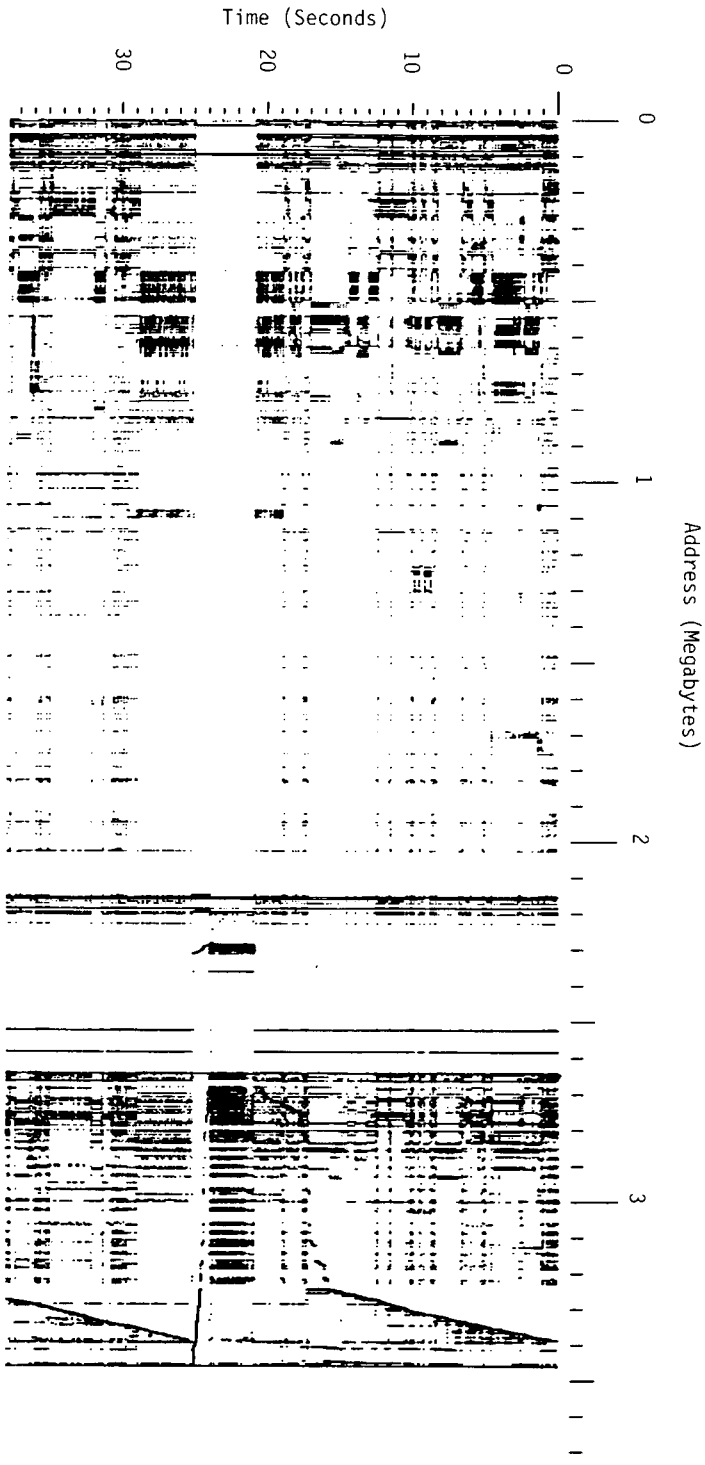

Figure 4: VAXIMA

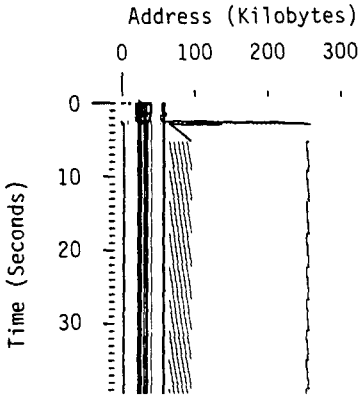

Figure 5: SEARCH Part 1

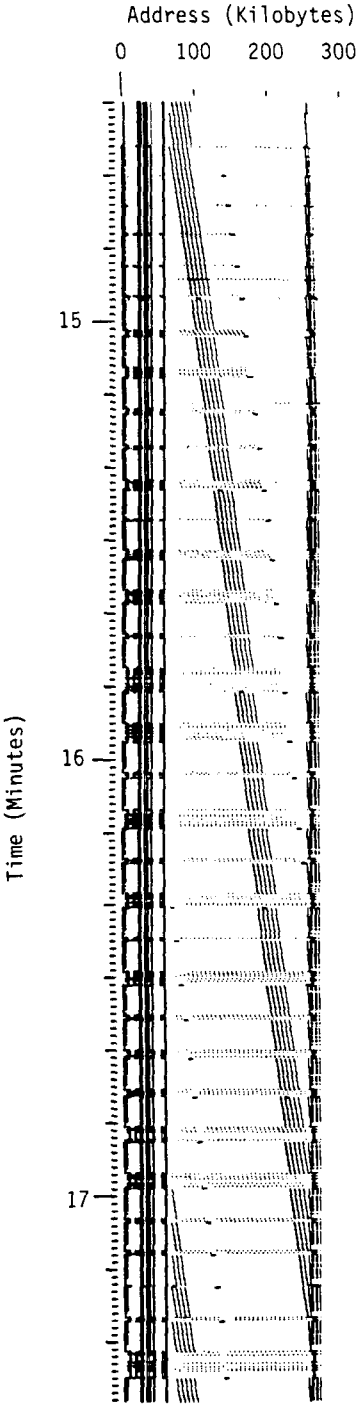

Figure 6: SEARCH Part 2

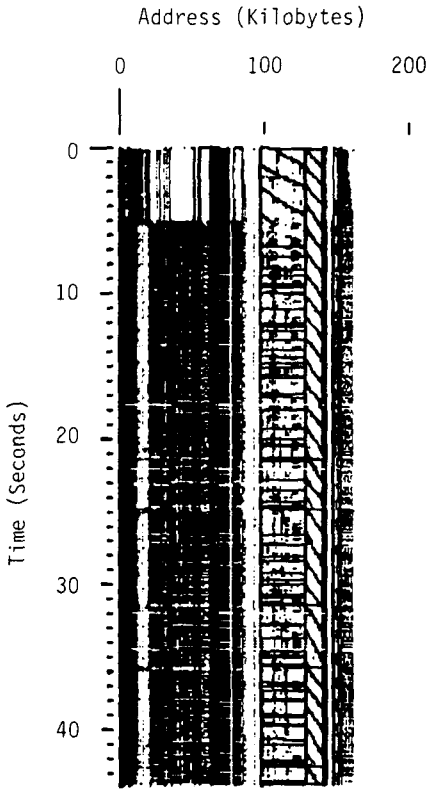

Figure 7: CCOM

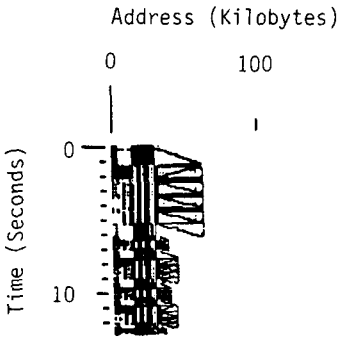

Figure 8: C2

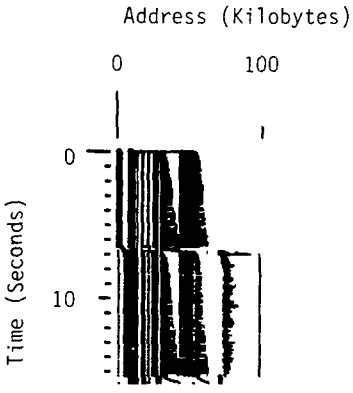

Figure 9: LD