Traffic Characteristics of a Distributed Memory System

Jonathan M. Smith and David J. Farber

Distributed Systems Laboratory
University of Pennsylvania, Philadelphia, PA 19104-6389

ABSTRACT

We believe that many distributed computing systems of the future will use distributed shared memory as a technique for interprocess communication. Thus, traffic generated by memory requests will be a major component of the traffic for any networks which connect nodes in such a system. In this paper, we study memory reference strings gathered with a tracing program we devised. We study several models. First, we look at raw reference data, as would be seen if the network were a backplane. Second, we examine references in units of ‘‘blocks’’, first using a one-block cache model and then with an infinite cache. Finally, we study the effect of predictive prepaging of these ‘‘blocks’’ on the traffic. We provide a novel representation of memory reference data which can used to calculate interarrival distributions directly. Integrating communication with computation can be used to control both traffic and performance.

Keywords: Distributed Computation, Distributed Memory, Traffic Characteristics, Networking, Computer Networks

1. Introduction

A long-range goal of research is the development of *distributed computing systems* which are well-matched to underlying network technology. Ideally, these systems would work ‘‘well’’ in any environment. However, speed mismatches between components such as computers, memories, software, and the network fabrics dictate certain choices for a high-performance result. Network technology promising speeds in the one billion bit per second (‘‘Gbps’’) range is on the very near horizon [4]. The remaining variable is the software interface between the computers and the network fabric, and in the attempt to remove this as the bottleneck in distributed system performance, The University of Pennsylvania’s Distributed Systems Laboratory and others have proposed distributed shared memory (DSM) as an appropriate model for high-speed interprocess communications. This DSM proposal stems from the observation that the minimal cost communications path between two processes is achieved with shared memory; we try to extend this approach into the network domain.

1.1. Setting

To be concrete, we imagine many high-performance workstations, connected by a point-to-point Gbps network shared by means of high-speed switches. In fact, we expect this imagined network to be realized in a near-future Gbps testbed. One likely topology of AURORA† is shown in **Figure 1**. The Gbps network will link four sites:

- Bellcore’s Morristown Research and Engineering Laboratory in Morristown, NJ
- IBM Research’s Computer Science Laboratory in Hawthorne, NY

† AURORA is a joint research effort undertaken by Bell Atlantic, Bell Communications Research, Inc., IBM T.J. Watson Research Center, Massachusetts Institute of Technology, MCI, Nynex, and the University of Pennsylvania. AURORA is sponsored as part of the NSF / DARPA Sponsored Gigabit Testbed Initiative through the Corporation for National Research Initiatives. NSF and DARPA provide funds to the University participants in AURORA. In addition, Bellcore is providing support to the Distributed Systems Laboratory through the DAWN project.

- MIT’s Laboratory for Computer Science in Cambridge, MA
- University of Pennsylvania’s Distributed Systems Laboratory in Philadelphia, PA

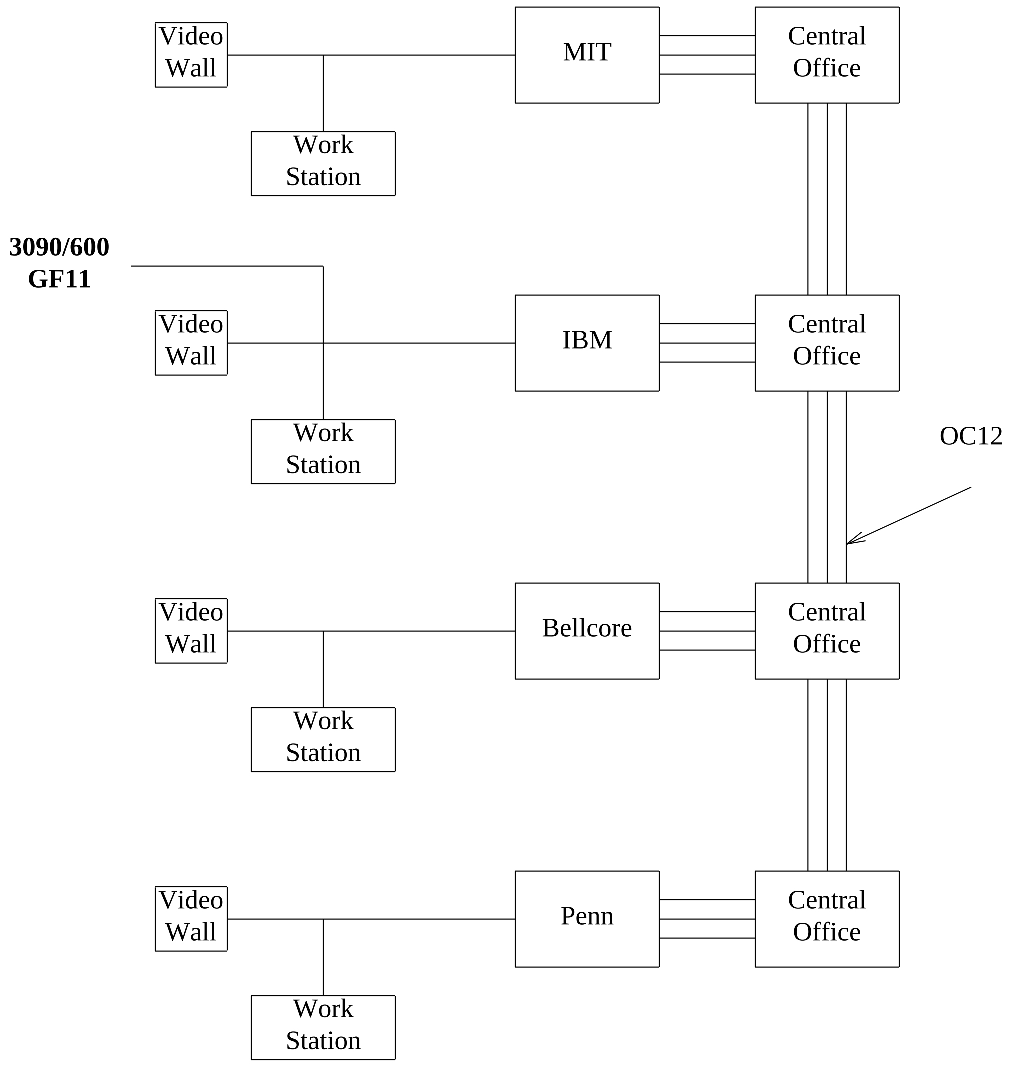

Figure 1: Possible configuration for AURORA testbed

As the illustration shows, we expect each participating site to have a variety of equipment connected via the testbed backbone. The ‘‘Video Walls’’ are Bellcore-developed devices for telepresence, comprised of paired wide-screen NTSC television displays, a pair of cameras, microphones, and studio-quality loudspeakers. Supercomputing facilities are embodied in the GF11 and 3090/600 computers at IBM’s T. J. Watson Research Center. Each site will also have a number of workstations, some subset of which will have direct, non-multiplexed access to the Gbps network. The host interfaces (HIs) of these workstations will receive and transmit packets. The machines will treat the HI as a direct memory access (DMA) device, meaning that the HI is able to read and write processor memory without processor intervention. In addition, the HI may allow portions of its own memory to be processor-addressable. The organization is illustrated in **Figure 2**.

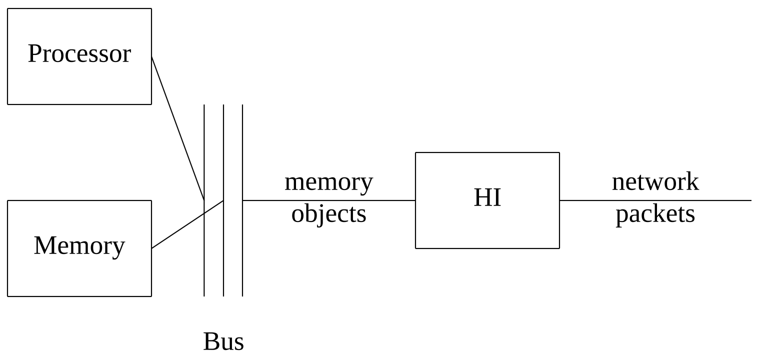

Figure 2: Proposed Host Interface design

The idea is that the workstation and HI technology will support interprocess communication (IPC) across the network fabric with as little processor intervention as possible, and as we argued earlier, the minimal IPC cost is achieved with shared memory. Thus, the workstations will both make and service remote memory requests, treating the network as an extension of the processor bus.

To understand the characteristics of traffic generated by such a design, it is crucial to understand the characteristics of computer memory referencing, as this is reflected directly in the traffic.

1.2. Memory Access

Typical computer environments consist of a set of hardware resources, managed by a supervisory program called the operating system (OS). The OS provides virtual machines to a number of *processes,* each of which represents a programmed sequence of instruction executions on behalf of some computer user. The execution of instructions involves fetching them and their operands from memory, and perhaps modifying the operands (instruction modification is considered dangerous and is often not allowed). Some of the instructions are executed by the OS to provide services to the process, but ideally most instructions executed are process instructions; other instructions are considered overhead. As can be seen from **Figure 2**, computers are organized with processors connected to memory via a bus, which is essentially a highly parallel network. Processors execute instructions rapidly in comparison to memory access (which requires a round-trip time over the bus in addition to delays resulting from the memory technology employed), so designers have employed various techniques to reduce the *average* access time. First, the need for memory access is reduced by the provision for on-processor *registers.* Second, the processor often includes a small, fast local memory, called a *cache,* which retains recently-accessed data. Third, the bus width can be increased (say from 16 bits to 32 bits) to amortize the delay for memory access across more data. Finally, provision can be made for *prefetching* memory objects (e.g., the next instruction to be executed) in anticipation of their use in subsequent processing. Many of these techniques have also been used to bridge performance gaps elsewhere in the memory hierarchy. For example, disk devices are *blocked* into aggregates of 512 bytes or more, and OSs typically provide a main-memory buffer cache for disk blocks. Since disk accesses often involve mechanical motion, the delays involved are significant and require aggressive management to be made tolerable. *Caches* are organized as a number of *blocks* (or *lines*) which are addressed based on some property of the objects they retain, such as its address at some lower point in the memory hierarchy. A step in the memory hierarchy is illustrated in **Figure 3**.

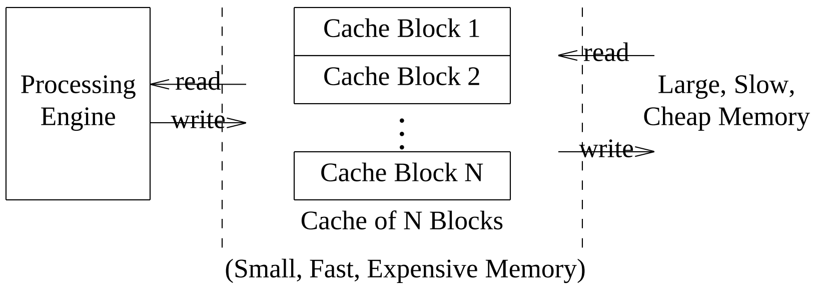

Figure 3: Using cache at memory hierarchy boundaries

The important properties of a cache are (1) the size of the blocks that it employs; (2) the number of these blocks that it has available; (3) the ratio between the cache access time and that of the slower storage it disguises; and (4) the *hit rate* of the cache, which is the ratio between the number of times an item is found in the cache (a ‘‘hit’’; if the item is not found, it is a ‘‘miss’’) and the number of times a reference is made.

The delays involved in accessing data over a wide-area network such as our AURORA testbed, which extends over 300 miles, require aggressive management. All of the management techniques applied in processor design are extensible to the network setting as well. The main focus is reducing the frequency of access, which incurs delays. When memory requests are serviced by the network, the traffic mix is changed significantly as the various management techniques are applied. We parameterize the relation between processor memory requests and network traffic with a measure we call the *gap.* The *gap,* measured in units of processor memory references between external memory requests (e.g., using the network), reflects the success of the management strategy at forestalling external requests. Thus, if a cache miss caused a value to be loaded, and the next 20 references were cache hits, followed by a miss, the ‘‘gap’’ would be 20 references. This value is processor-speed independent, and can be scaled appropriately. For example, the processor speed can be used to convert *gaps* to time intervals, and the size of memory units fetched can determine the packet size. The *gap* can thus be viewed as a measure of interarrival times.

1.3. Organization

In an attempt to understand the traffic characteristics of a distributed memory system, we devised an experimental apparatus which allows us to accurately measure the traffic generated by a node. This apparatus is described in Section 2. Sections 3 and 4 consist of measurements and analyses derived via the apparatus. Section 3 focuses on issues of cache block size and cache size, with some attention paid to data analysis techniques, and Section 4 focuses on a cache management strategy known as prefetching or *anticipation.* Section 5 concludes the paper and draws the conclusion that integrating computation and communication can have far-reaching benefits for designers of both computing systems and networks.

2. Experimental Apparatus

‘‘*One real experiment is worth far more than half a century of discussion about the meaning of a thought experiment....*’’ [8]

In the introduction, we proposed a model of a distributed system which consists of a virtual memory distributed across a set of high-performance workstations on a high-speed network. However, such a system is not available to measure. Yet, we want usable estimates of traffic in order to determine delays and achievable throughputs.

We chose to use an existing workstation, namely, a Hewlett-Packard HP9000 Series 300. These workstations are configured with 16 megabytes of main memory, and operate a UNIX variant, HP-UX Version 6.5. This workstation serves as a traffic generator. Using such a workstation (with its circa 3 million instructions per second (MIPS) performance) is not as far-fetched as it might at first seem, as (1) all RISC (reduced instruction set computer) technology [15, 16] workstations we are aware of operate UNIX or a variant; (2) operational distributed shared memories [12, 21] using UNIX as a base exist; and (3) our measurement methodology, described next, is speed-independent. One potential flaw in the analysis is that RISC architectures have many registers and compilers generate code with heavy register usage in order to reduce memory traffic. This flaw is partially addressed by the fact that many workstations will continue to be CISCs (complex instruction set computers), and partially by the fact that we address caching† later in this article.

To measure the traffic generated during a session, we constructed a tracing tool, *rw_trace.* The use of a software tool is desirable, since hardware bus monitors may miss traffic which is captured by on-processor caches. Our tracing tool executes binaries without recompilation. *Rw_trace* is invoked with a command string [3] and an output file as parameters, e.g.,

```
        rw_trace -foutput emacs file
```

The command string consists of a command name (in this case, ‘‘emacs’’) followed by a list of parameters. *Rw_trace* uses the UNIX debugging interface, *ptrace(),* to execute the command in a ‘‘single-step’’ fashion. At each step, the current program counter is fetched and used to fetch the current instruction. The instruction and its operands are decoded (as per [13]) to generate a ‘‘memory reference string’’ consisting of <byte address, read/write> 2-tuples. This memory reference string is written to the specified output file (the output must be sent elsewhere so that use of interactive programs remains possible while tracing). The only references not captured (at present) are those made through system calls such as *read()* and *write(),* most of whose activity is captured in the trace by later copying in and out of the program’s buffers.

In order to reduce the amount of data we must present, we will limit our discussion to traces derived from a single command. After a large number of commands were traced (both out of curiosity and to test the tracer!), we found that the command generating (1) the most complex, and (2) most voluminous, memory traffic was the GNU [20] (GNU is an acronym for Gnu’s Not Unix) Emacs screen editor. On our system, the executable file is the largest (593920 bytes of executable code) of the frequently-executed programs. The program generates over a million memory references between invocation and display of a window for editing files. Our version of the editor is especially complex, as it includes code to interface with the a window management system [7] and creates and manages its own window when started in an X window. In the next section, we describe our observations which are derived from tracing the execution of this editor.

3. Traffic Properties

The gross memory traffic created by the editor is indeed voluminous; a trace of the editor’s execution references approximately 28,000,000 bytes of data, although of course many of these bytes are accessed repeatedly, and many of them are accessed in the aggregate form of 32-bit *word* data objects. If the editor is being traced, the execution cycle requires about 9 hours of clock time, due to the overhead incurred by single-step execution, context-switching, and instruction decoding. Untraced, the editor takes about 1 second of OS time, and 2 seconds of user CPU time to start; this is accomplished in about 6 seconds of wall clock time. Thus, about 4.7 megabytes worth of referencing is done, per second. Measurements show that, on average, each instruction causes approximately 8 bytes of memory access to take place. With no caching of any type, this means that the offered traffic is a steady-state phenomenon, driven by the instruction execution rate of the CPU. Thus, for a 3 MIPS workstation such as this, assuming that the requests can be serviced infinitely fast, 24 megabytes, or about 200 megabits per second of traffic are generated. Of course, assuming that the server is infinitely fast is ridiculous, as it may be servicing requests from slow disks or other high latency devices. These latencies can be folded into the network latency. Assuming that there is a round-trip delay per request of 5 milliseconds, and that each instruction execution consists of two fetches, one for the instruction, and one for its operands, we can execute one instruction every 10 milliseconds, or one hundred instructions per second. Thus, we must use bigger aggregations of data and caches.

The next two subsections look at caches of one block and ‘‘enough’’ blocks (i.e., infinite) under the following assumptions:

1. All writes are preceded by a read of the same block
2. All writes are retained until necessity (e.g., a full cache) forces a write-through. This is the ‘‘copyback’’ policy; dirty cache blocks are flushed at program termination, generating a burst of network traffic.
3. Blocks are 4096 bytes in size, as on the HP9000.

† These large register sets behave much like a compiler-managed cache memory for data objects.

3.1. One block cache

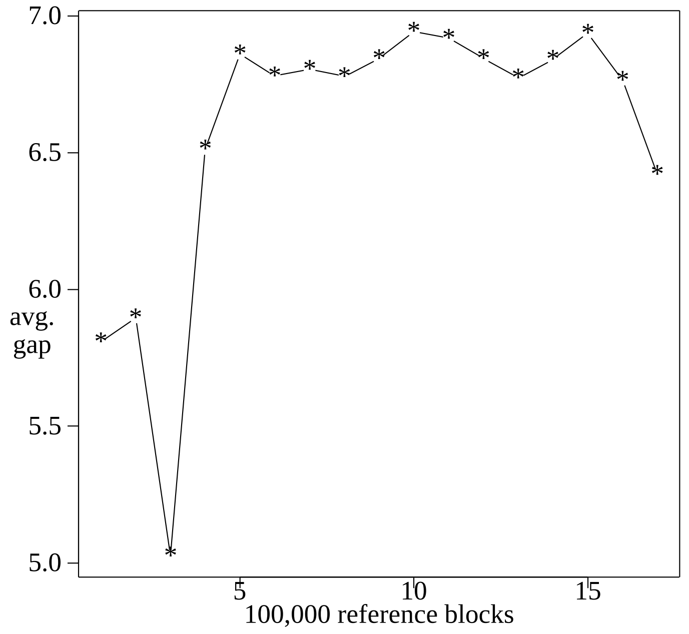

Figure 4: Average gap for 100,000 reference groups

One important observation we can make is that both aggregation (i.e., using larger data units, with the goal of reducing the average latency of smaller data units) and caching (retaining frequently used data in order to avoid rereferencing) can help and in fact, are widely used to cope with various types of speed mismatches. Of more interest to us is the effect such techniques have on the traffic offered to the network.

The technique we have used is to study the *gap* behavior of memory references. When blocking or caching are applied, a steady stream of traffic will be converted to a sequence of block requests of some size. This size is typically fixed by the operating system software and thus can be considered constant for any given application execution. What changes with blocking and caching, is the interarrival distribution, which can be approximated by the interval between successive memory requests. As we mentioned in the introduction, we measure this interval, the ‘‘gap’’, with the number of memory references made between block requests.

In this paper, we are trying to understand the traffic generated by processor memory references, and factors affecting that traffic. As we will show, the traffic can be greatly affected by the choice of cache parameters and cache management strategies. In order to limit the directions we must pursue, we have focused on several extremal cases of these choices. One extremal case deserving study is that of a single block cache. We used 4096 byte blocks, as mentioned before. As the traces were very large, we produced aggregate data by considering block references in 100,000-block reference groups. **Figure 4** shows that the average gap consists of about 5 to 7 references, or about 1 instruction execution. Thus, this tiny cache is not very effective at this block size.

Suspecting that the averages were disguising some significant behavior of the traffic, we plotted the *maximum* gap value, in references, observed across the same set of 100,000 block reference aggregates. This is shown in **Figure 5**, where we see that there are some large gaps initially (e.g., 4926 bytes, which is greater than a 4096 byte block) but none later in the program’s execution.

To study the effect of block size, we reran the 1-block cache traces with a 65536 byte block size, a factor of 16 increase over the original 4096 byte size. If the references are sequential, or remain within 65536 byte bounds, we should have seen an increase in the gap. However, the average gap and maximum gap values were almost the same in magnitude! That is, with 65536 byte blocks, the maximum gap observed in the first 100,000 65536 byte block references remained at 4926.

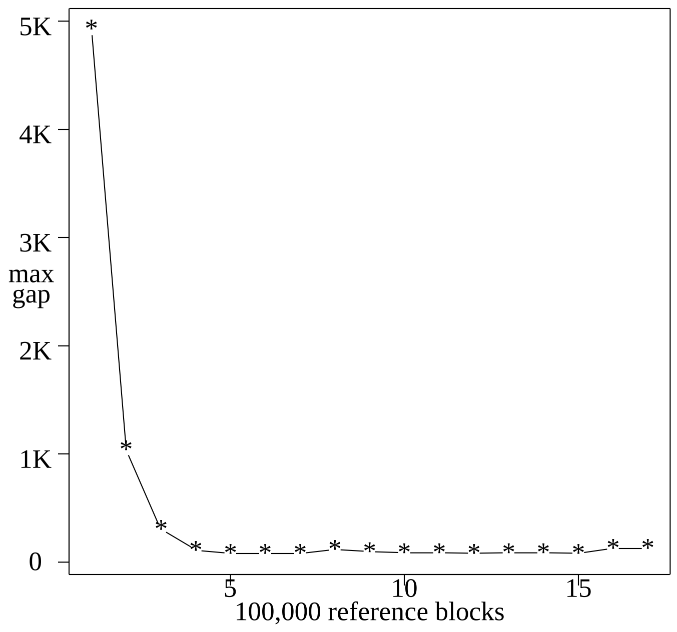

Figure 5: Maximum gap for 100,000 reference groups

Thus, we have paid an increase of a factor of 16 in traffic for no reduction in the interarrival rate. Note the implications, for this program mix, of any simplistic prefetching scheme, since the gap value is directly correlated with the hit rate.

But the model of a single block cache, while instructive with respect to the assumptions about block size and its effect on cache performance, is clearly far from ideal; aside from the fact that data and instructions form separate streams (however, a 2 block cache is not much better!) we would want (1) a smaller block size, and (2) a larger cache size. In the next section, we discuss the impact on network traffic of another extremal case, that is, the case where enough memory is available to hold all items referenced over the life of the program. While we model the cache behavior as ‘‘copy-back’’, it is trivial to modify traffic for a write-through cache by treating the written traffic as we treated, e.g., the 1 block cache case. The writes generate a steady traffic stream corresponding to the rate at which writes are generated. Since this is relatively easy to model (and our trace data has read/write flagging) we chose to ignore write-through caching in our study.

3.2. ‘‘Enough’’ cache (infinite)

To analyze the effects of successful caching in the face of increasing main memory sizes for workstations, we chose the extremal case of ‘‘enough’’ main memory for caching all page references made during execution of the program. This is similar to the strategy used by the Andrew File System [9], which caches referenced files on its local disk to avoid rereferencing over the network. Remember that a *gap* is the number of memory references made between cache misses. Thus increases in the gap correspond to decreases in the offered traffic. The distribution of the gaps is equivalent to the *interarrival distributions* used in traffic and queueing analysis when the memory references are mapped to the network. For a given gap plot, two things matter; the block size for references and the instruction processing rate of the CPU. The first matters because it defines the ‘‘packet size’’ for analyses, and the second matters because it allows the gap values (stated in memory references) to be translated to absolute time figures.

We began our analysis by pre-processing our reference data into the form of <block number, byte reference number> pairs. Each pair is generated upon a cache miss discovered in the data; that is, the reference stream contained a reference to a block not contained in the cache. The block number of the 2-tuple is the byte address causing the miss, divided by the block size (4096). The byte reference number is a running counter of the number of references which have been made to this point in the stream. Each of these misses can be assigned a sequence number according to the order of occurrence within the reference stream, which we call the *miss number*. The difference between byte reference numbers for two successive misses yields the gap. For our Emacs execution, 1732 of these records were generated, corresponding to 1732 4-kilobyte pages being fetched. We plotted the miss number versus the gap; this is illustrated in **Figure 6**.

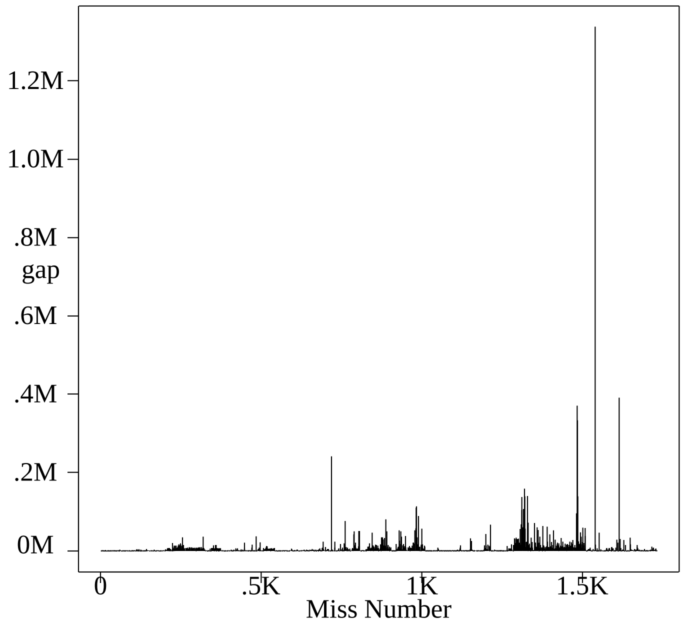

Figure 6: Miss number versus gap

We show this figure in order to drive an experimental analysis of the data; in fact, this was our original model for data presentation and subsequent analysis. However, we have found more effective ways to discern behaviors of interest, which we develop in the following section. The graph of Figure 6, while no doubt interesting, has the visual difficulty caused by the outlier (a gap of 1.3 million memory references made without leaving the cache). Thus, any behavior exhibited by the other references is obscured. We plotted the gap logarithmically, but the result was a ‘‘cloudy’’ scatterplot, which obscured the data. Smoothing the data was a productive technique. We used the `lowess()` routine [5] of the S system [2] with various parameters to smooth the scatter plots. The main value varied was `f`, the fraction of the data set used for smoothing at each `x` point. The larger the `f` value, the smoother the fit.

While the figure illustrates a general trend which should hold with a cache, that is, the cache becomes more effective as it becomes loaded with data, it obscures the detailed behavior of the traffic. The smoothed curve would indicate, for example, that there was never a gap larger than 3000 references, i.e., less than one page size. A lesser degree of smoothing allows a somewhat more detailed analysis, as **Figure 8** illustrates.

The figure clearly illustrates the bursty nature of the traffic which the higher degree of smoothing obscured. There is still some obscuring of the data due to smoothing, e.g., the 1.3 million reference gap has been scaled down by a factor of 50 (1732/32=54) but the general behavior is clear. A last attempt to understand the data is given in **Figure 9**, with f=1/8. In this case, there are several clearly illustrated ‘‘peaks’’ in the gap plot. These peaks and the corresponding valleys are due to phase changes [6] in the memory referencing during execution of the editor. Emacs is changing its working set as it moves from one phase of execution to another. When its working set has been loaded into the cache, the gap rises, as references must leave the cache less frequently. On a phase change, the shift to a new working set causes the program references to leave the set in the cache, and thus the gap decreases.

The figure illustrates the editor going through 4 phases (the phases correspond to the valleys, where the cache is warming up, not the peaks, where it has become effective); this phase behavior is consistent with that observed by virtual memory researchers in the past. The peaks indicate an increase in the interarrival distribution; that is, the

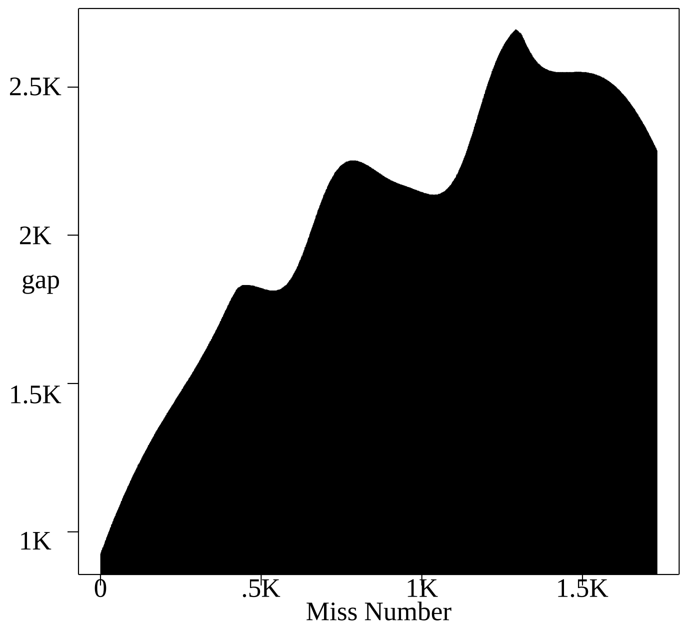

Figure 7: Miss number versus smoothed gap, f=1/2

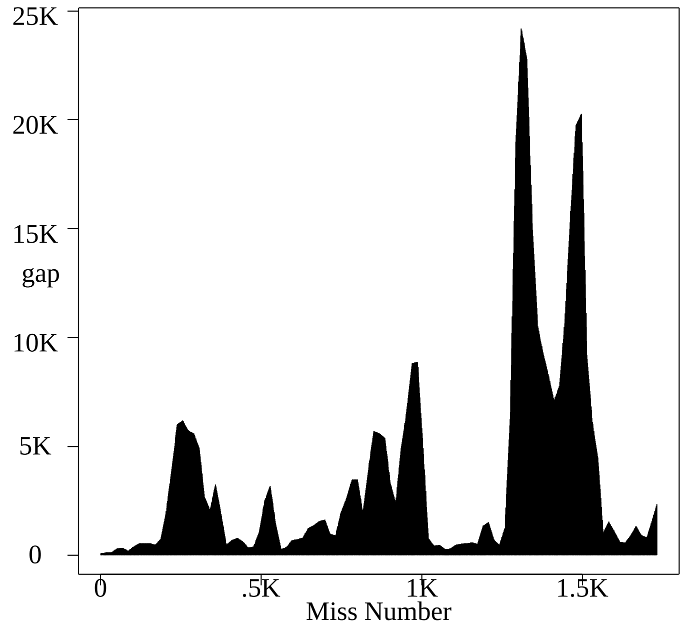

Figure 8: Miss number versus smoothed gap, f=1/32

onset of a peak indicates that the interarrival rate is dropping off, and the termination of a peak indicates that the interarrival rate is increasing.

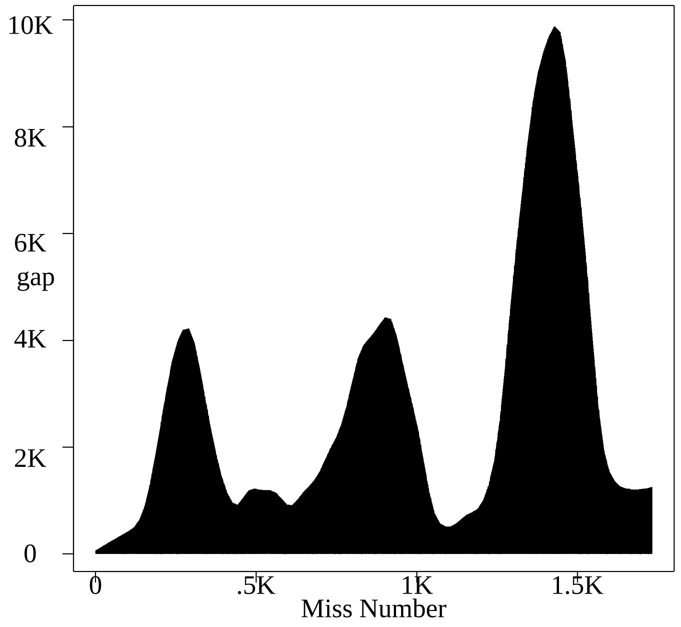

Figure 9: Miss number versus smoothed gap, f=1/8

4. The effect of anticipation

Anticipation, also known as prefetching, is the idea of pre-loading some buffer area (perhaps the cache) with data in order to reduce future misses. If the anticipation strategy is successful, data will be available in fast storage when needed by the program, thus reducing the miss rate. Prefetching has been studied in many settings in the past. Smith [17] discussed sequential prefetching in the general case of a multi-tiered hierarchy of memories. Based on traces derived from IBM 370 architecture machines, he showed that prefetching was most effective with small page sizes (e.g., 32-64 bytes), and suggested the conclusion that the CPU's cache was therefore the most effective point at which to implement prefetching. The potential for 10-25 percent increases in CPU speed was reported. Smith's article provides a short survey of pre-1978 research in the area which need not be repeated here. Smith [18] also reported the effectiveness of prefetching in data base settings where sequential access is prevalent. Ritchie and Thompson's UNIX operating system prefetches file data when sequentiality is observed in the virtual block numbers referenced; the actual algorithm is discussed in some detail in Bach [1]. Recently, Kotz and Ellis [11] reported potential performance increases for multiprocessor systems, specifically a BBN Butterfly^{TM}, by prefetching file system data. Based on their data, the improvements in response time they observed exceeded 15 percent in most cases, with improvements extending up to 70 percent. The best performance occurred when a prefetch was shared by many processes in subsequent references. However, they also observed that if the benefit of prefetching was poorly distributed between processes, the total execution time for the workload could increase.

As we argued in the introduction, we are interested in applying anticipation in a somewhat different setting, namely that of Gbps networks. Gbps WANs are distinguished from Gbps LANs by their higher latency due to propagation (and other, e.g., switch and repeater) delays. Gbps WANs are distinguished from current lower-speed WANs by their increased bandwidth, e.g., almost one thousand times a T1 channel. This high bandwidth*delay product suggests that approaches such as anticipation might be more profitable than in other settings, for two main reasons. First, a ‘‘miss’’ is very expensive. We have assumed workstations with speeds approaching 50 MIPS, so that a 5 millisecond round-trip delay (e.g., Philadelphia,PA - Cambridge, MA) represents 250,000 instruction times on such a machine. Thus, techniques which reduce misses are highly desirable. Second, the high bandwidth makes us less sensitive to unused prefetches.

Smith [14, 19] has suggested anticipation to reduce the amortized latency seen by a program execution which is using the network to service memory requests. Anticipation changes the traffic generated by a node in the following ways. Successful prefetches increase traffic initially, as they fetch more data than is strictly necessary.

However, their success means that some future references have been made unnecessary, thus reducing traffic later in the program’s execution. Unsuccessful references, on the other hand, add traffic without reducing traffic in the future. Thus, they just raise the level of traffic.

In this section of the paper, we will study the effect of an anticipation strategy in terms of its aggressiveness, that is, how many blocks are prefetched at each page fault. As has been pointed out by A. J. Smith in the several paper referenced, software prefetching must be done when a fault occurs; otherwise it cannot be scheduled. We look at the extremal case of success, which means that we always pick the right pages to prefetch, as if we had some guide. In fact, for many program executions we can achieve such a guide using reference histories. This idea is modeled after the RH-Tree scheme devised by Iyengar [10] which works as follows, to build a ‘‘reference-history’’ tree:

1. Each time a page is referenced, if we are at that point in the tree, we increment a counter. If a page that we have not accounted for is referenced, we create a new branch.
2. To prefetch, we can choose either the most highly probable set of pages or the most likely "path"

Over time, the tree is built up, so that time-varying referencing behavior is built into its history. This data is especially useful on program startup, when programs almost always follow the same sequence of instructions. That is the case with Emacs, which faults in the same pattern for its first thousand or so page faults each time. So emacs was run once to warm up the ‘‘history’’, and once the history was in place, emacs was re-traced assuming a prepaging strategy which used the previous trace data. The variable we applied was the degree of lookahead, which corresponded to 4 pages of 4096 bytes (‘‘prefetch(4)’’) and 16 pages of 4096 bytes (‘‘prefetch(16)’’). The effect this has on traffic is two-fold. First, the ‘‘packet size’’ increases by factors of 4 and 16, respectively. Second, the interarrival times of these packets are greatly increased. We plot data similar to that of section 3.2, assume an infinite cache, and limit ourselves to the 1/8 smoothing case.

4.1. 4 block lookahead

The case of four block lookahead means that whenever we could make a memory request for a single block, we request four blocks instead. If we successfully predict which three blocks should be fetched in addition to the faulting block, we can achieve both a reduced number of requests (since we are aggregating requests into 4 block groups) and an increase in the interarrival ‘‘gap’’ we discussed earlier. We retraced the program execution using data from a previous run, and the gaps were computed as before. The results were smoothed and graphed with various smoothing parameters; **Figure 10** illustrates the results with the smoothing parameter set at 1/8.

As can be seen from the graph, the shape of the curve is remarkably like that of the miss numbers graphed in **Figure 9** with the same smoothing (in fact, the shapes were so similar that we omitted prefetching data smoothed with f=1/2 and f=1/32); some behaviors are a little more visible because there are fewer data points to make up an eighth; in particular the split in the peak of the last gap is visible. The fact that the shape is similar is not surprising; the interesting artifacts are actually the scale changes visible on the bottom and sides of the plot. The vertical dimension of **Figure 10** has changed by a factor of five from **Figure 9**, and the horizontal direction has decreased by a factor of four (433 misses versus 1732). What this means is that successful anticipation increases the gap, and thus lengthens the interarrival times. However, depending on the degree of aggressiveness exhibited, the packet size may be much larger; in this case it was 16334=4*4096 bytes. If the latencies are large and cache misses pace the program, the execution time will decrease dramatically, since the fault count has gone down by a factor of four.

4.2. 16 block lookahead

It is interesting to extend the experiments one step further, and test the effect of anticipating even further into the ‘‘future’’. We do that by prefetching 16 blocks, instead of 4 as in the previous measurements. **Figure 11** shows the effect of this modification; this graph can be compared to **Figures 9** and **10**.

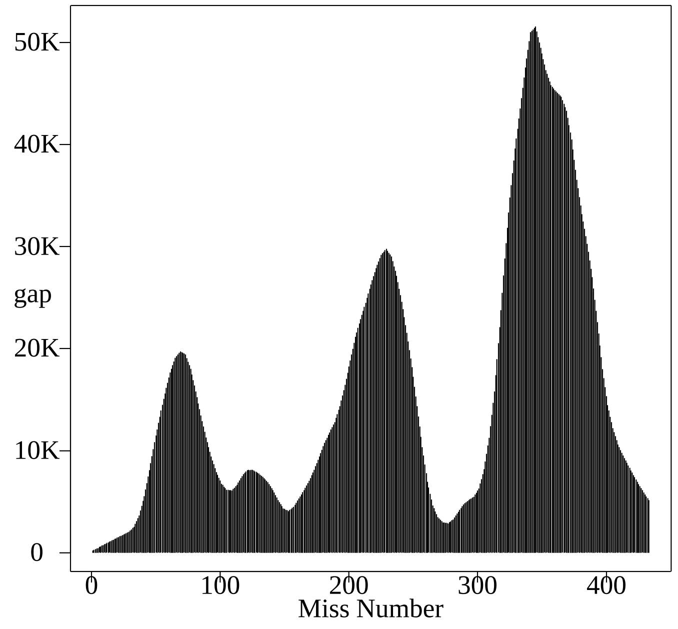

Figure 10: Prefetch(4) miss number vs. smoothed gap, f=1/8

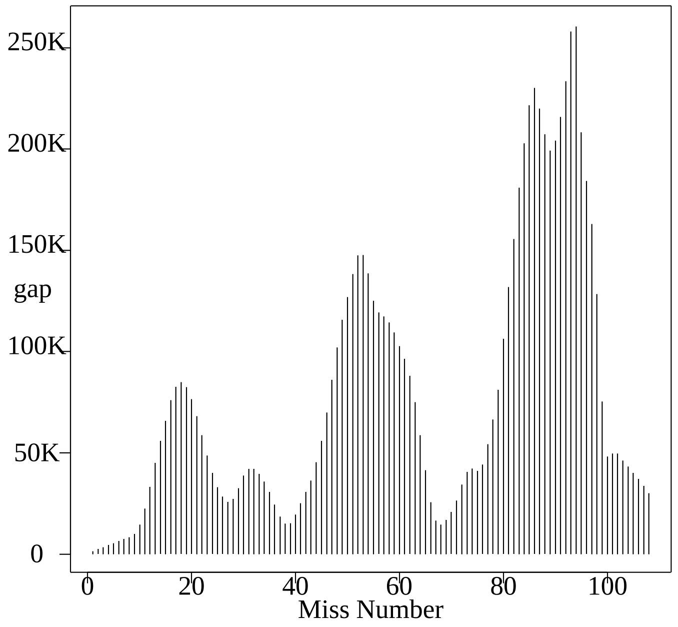

Figure 11: Prefetch(16) miss number vs. smoothed gap, f=1/8

We see that the splitting of the last peak in gap value is more visible. Once again note the change in scales; the vertical axis now ranges to a gap of 250,000 references, as opposed to 10,000 in **Figure 7**. This is due to both the increase in gaps and to the way smoothing is done; note also the reduction of the horizontal access to 108=1732/16 references. What we have seen here, with a prefetch of 16 blocks, is a continuation of the trends we have seen before; successful prefetching dramatically reduces the number of packets, increases the sizes of packets, and increases the gap between misses.

5. Conclusions and Summary

We have examined memory references as a source of network traffic, with the goal of building distributed computing systems based on a shared memory model of interprocess communication. For concreteness, we traced the execution of a memory-intensive screen editor. From these traces, we saw that:

(1) raw memory traffic is very smooth, with each instruction generating about 8 bytes of traffic, on average. Thus, the traffic rate is paced by the instruction execution rate. With an intervening cache, the traffic reaching the network fabric can be scaled down by the ‘‘miss rate’’, but is still sensitive to the execution rate of the processor. The cache also removes some of the smoothness of the traffic, as programs generate small ‘‘humps’’ as they move through phases of execution. These phases exhibit sufficient locality of reference to stay within the cache.

(2) ‘‘blocked’’ memory traffic is less smooth, as it is characterized by less frequent requests for larger objects. In the minimal (1 block cache) case, the traffic pacing was determined by the locality of reference exhibited by the executing program. For large blocks, the traffic bursts are larger, but less frequent. While there is insufficient data presented to draw a firm conclusion, the data suggest that the best application of finite cache space is achieved by a large number of small blocks. In any case, a higher hit rate translates into a longer interarrival time. For the maximal (infinite, or ‘‘large-enough’’) cache case, the traffic rate is initially high, as the cache goes through a ‘‘cold start’’ phenomenon, and then drops as the cache fills. This dropoff corresponds to the working set of the program being acquired; the traffic only rises again on phase changes.

(3) anticipatory behavior, used to reduce the penalty imposed by latency in WANs, reshapes the traffic generated by a cached system in the time domain. In particular, prefetching causes an initially higher traffic rate, with the eventual result of a lower traffic rate later in the execution. If the prefetches are largely successful, the traffic resembles an impulse function, with humps at each working set phase transition. If the prefetches are largely unsuccessful, the reduction in request rate achieved by the cache is thwarted to some degree by the additional traffic due to unsuccessful prefetching attempts. The extreme case of prefetching success is prepaging the entire working set.

Since caches are necessary for reasonable performance, and anticipation is one of the few techniques available to reduce latency in high-speed WANs, we expect that the eventual traffic observed will correspond to cached memory traffic reshaped by whatever success is achieved by the lookahead strategy. Our presentation of data in the form of reference ‘‘gaps’’ allows scaling across processor speeds and can be directly translated into interarrival time distributions. The packet sizes are a direct function of the page size, but may be affected by an anticipation scheme.

Certainly, much remains to be done. While we were rather selective in the choice of our application, we have nonetheless presented data from only a single application, and the multiplexing of traffic induced by multiple processes on a single processor, and multiple processors, remains to be studied. Individual applications, however, will represent the primitive traffic sources in any memory-based communications scenario, and thus serve as the basis of this study.

What is perhaps the most important conclusion from our study is the observation that network traffic can be controlled when it is integrated with the operating system layer which provides memory management services to computations. We were able to adjust the packet size and arrival rate, and processor scheduling can be used (although it has not been shown here) to smooth traffic. In addition, successful prefetching strategies can dramatically change the characteristics of a system, as illustrated by the scale change between **Figures 9, 10,** and **11**. We believe that the ability of the operating system (in particular, the portion of the operating system usually referred to as the ‘‘memory manager’’) to adjust traffic characteristics under this distributed memory scheme suggests that memory is a potent network abstraction, and that the integration of communication and computation is a rich and rewarding ground for study, which we intend to explore further in the setting of the AURORA Gbps testbed.

6. Acknowledgements

Criticism of this work from an early presentation at AT&T Bell Laboratories helped to refine the work. Constructive suggestions and critical reading by H. Rudin and P. Heinzmann greatly improved the presentation.

7. References

[1] M. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall (1986).

[2] Richard A. Becker, John M. Chambers, and Allan R. Wilks, *The NEW S Language - A Programming Environment for Data Analysis and Graphics,* Wadsworth, 1988.

[3] S.R. Bourne, ‘‘The UNIX Shell,’’ *The Bell System Technical Journal* **57**(6, Part 2), pp. 1971-1990 (July-August 1978).

[4] Karen L. Bowers, in *Proceedings of the First Gigabit Testbed Workshop*, ed. Karen L. Bowers, Corporation for National Research Initiatives, Washington, DC (December 7-8, 1989).

[5] W. S. Cleveland, ‘‘Robust Locally Weighted Regression and Smoothing Scatterplots,’’ *JASA* **74**(368), pp. 829-836 (December 1979).

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

[7] Jim Gettys and Robert Schiefler, *The X Window System*, 1985.

[8] John Gribbin, *In Search of Schrodinger’s Cat: Quantum Physics and Reality,* Bantam (1984), p. 222.

[9] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, ‘‘Scale and Performance in a Distributed File System,’’ *ACM Transactions on Computer Systems* **6**(1), pp. 51-81, Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles* (February, 1988).

[10] Anand Iyengar, *A Predictive Paging Algorithm*, Unpublished Memorandum, April, 1990.

[11] David F. Kotz and Carla Schlatter Ellis, ‘‘Prefetching in File Systems for MIMD Multiprocessors,’’ *IEEE Transactions on Parallel and Distributed Systems* **1**(2), pp. 218-230 (April 1990).

[12] Ronald G. Minnich and David J. Farber, ‘‘Reducing Host Load, Network Load, and Latency in a Distributed Shared Memory,’’ in *Proceedings, 10th International Conference on Distributed Computing Systems*, Paris, France (June 1990).

[13] Motorola, *MC68020 Microprocessor User’s Guide,* Prentice-Hall, Englewood Cliffs, NJ (1984).

[14] Craig Partridge, ‘‘Workshop Report - The Internet Research Steering Group Workshop on Very-High-Speed Networks,’’ *ACM SIGCOMM Computer Communications Review* (1990).

[15] D. A. Patterson and C. H. Sequin, ‘‘RISC I: A Reduced Instruction Set VLSI Computer,’’ in *Proceedings of the 8th International Symposium on Computer Architecture* (1981), pp. 443-457.

[16] G. Radin, ‘‘The 801 Minicomputer,’’ *ACM SIGARCH Computer Architecture News* **10**, pp. 39-47 (March 1982).

[17] A. J. Smith, ‘‘Sequential program prefetching in memory hierarchies,’’ *IEEE Computer*, pp. 7-21 (December 1978).

[18] A. J. Smith, ‘‘Sequentiality and prefetching in database systems,’’ *ACM Transactions on Database Systems* **3**(3), pp. 223-247 (September 1978).

[19] Jonathan M. Smith, *Anticipation in Very High Speed Networks,* Distributed Systems Laboratory, University of Pennsylvania (1991). Distributed Systems Laboratory Technical Report and Working Paper

[20] Richard Stallman, *GNU Emacs Manual, Fourth Edition, Version 17,* Free Software Foundation, Inc., 100 Mass Ave., Cambridge, MA 02138 (February 1986).

[21] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber, ‘‘A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems,’’ *ACM Operating Systems Review* **24**(3), pp. 40-67 (July, 1990).