Issues in the Implementation of a Remote Memory Paging System

Edward W. Felten and John Zahorjan

Department of Computer Science & Engineering
University of Washington

Technical Report 91-03-09
March 1991

Issues in the Implementation of a Remote Memory Paging System [*]

Edward W. Felten
John Zahorjan

University of Washington
Department of Computer Science and Engineering

felten@cs.washington.edu
zahorjan@cs.washington.edu

February 1991

Abstract

We describe the design and implementation of a remote memory paging system. This system uses the memory of idle workstations on a network as backing store for virtual memory paging of address spaces on active machines. Our prototype implementation runs on a network of DEC Firefly workstations. Performance is superior to paging to local disk, and this speed advantage is expected to grow as the performance of disks falls farther behind that of other components of computing systems. A number of interesting policy issues arise in systems of this type; we discuss several of these issues and the tradeoffs between various policy choices.

1 Background and Motivation

The performance of computer systems has improved dramatically over the last several years, and will continue to improve in the near future. Processor speeds and memory access times are expected to improve by a large factor, and network speeds are expected to grow equally rapidly. In contrast, the performance of disks has been relatively constant, and is not expected to improve much in the next few years. (Currently proposed parallel disks, e.g. RAID [PGK88], are unlikely to help paging as the amount of data per transfer is by nature relatively small.)

Disks provide backing store for virtual memory paging. As the performance gap between disks and other components widens, paging becomes more and more expensive. One solution is to increase the size of physical memory; this is commonly done on workstations. This solution is expensive and merely delays the onset of unacceptable performance due to disk paging.

[*]This material is based upon work supported by the National Science Foundation (Grants DCR-8352098, CCR-8619663, and CCR-8703049), the Washington Technology Center, and Digital Equipment Corporation (the External Research Program and the Systems Research Center). Felten was supported in part by a Mercury Seven Fellowship.
Authors' addresses: Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195.

The only way to truly escape this dilemma is to avoid using disks for paging. This paper presents a scheme which uses the memory of idle machines on a network as backing store for those machines that are active. This scheme provides a performance advantage at the present time; this advantage is expected to become larger in the future as the relative cost of disk accesses increases.

Shared virtual memory (SVM) systems [SZ90, LH86, BCZ90, FBS89], provide services which are similar to our system. Although their implementation is similar to ours, SVM systems are designed for a different function. SVM systems are primarily concerned with providing a single, coherent address space to threads of control running at physically distributed sites.

While some SVM systems provide remote memory paging as a feature, systems designed with sharing in mind do not provide the best platform for applications that do not share data. The semantics of shared memory are more strict than those of primary memory and backing store; this forces implementations with sharing to perform extra invalidations if used simply to provide backing store. In addition, SVM implementations are optimized for performance of shared-memory operations rather than remote paging.

There is a strong analogy to techniques for sharing CPU load between workstations [LLM88]. Systems which execute parallel programs on a network of workstations face different problems, and hence have more complex implementations, than systems which merely migrate sequential jobs to idle machines. In the same way, SVM systems face problems due to supporting parallel jobs, while remote memory paging systems do not.

The main disadvantages of remote memory paging lie in security and fault-tolerance. Storing a job's memory pages on another machine exposes them to risk of unauthorized access, or loss of data if the remote machine crashes. These problems can be mitigated (by encryption and replication, for instance), but we will not focus on such issues here. We believe that many installations will be willing to live with these risks in exchange for improved performance.

2 The Design

The central idea of remote memory paging is to set up "memory server" processes on idle (or lightly-used) computers on the network. When one machine runs out of physical memory, it stores some of its pages in the memory of another machine, using the memory server on the remote machine. A server process is dormant when its machine is in use; it becomes active and accepts pages only when its machine is otherwise idle. This is analogous to systems which use idle machines on a network as compute servers.

Figure 1 illustrates the structure of our implementation. The application program runs on top of a slightly modified kernel; the application makes kernel calls to say which of its pages should be paged remotely. When the kernel decides to read or write one of these pages to backing store, it issues a command to a local clerk thread which runs at user level. The interface between kernel and clerk is similar to Mach's external pager interface [YTR+87]. In principle, some or all of the clerk's functionality could be moved into the kernel; this would provide improved performance and remove the need for the external pager interface.

The clerk thread communicates with one or more servers through remote procedure call (RPC) [BN84, SB90]. The clerk is responsible for keeping track of which local pages are stored remotely, and which server is holding each such page. The clerk also decides on which server to store newly allocated pages. When a page is allocated, the server provides the client with a "handle" which allows rapid and secure access on subsequent references.

The registry is a name-server that keeps a list of active servers. Servers register themselves when they

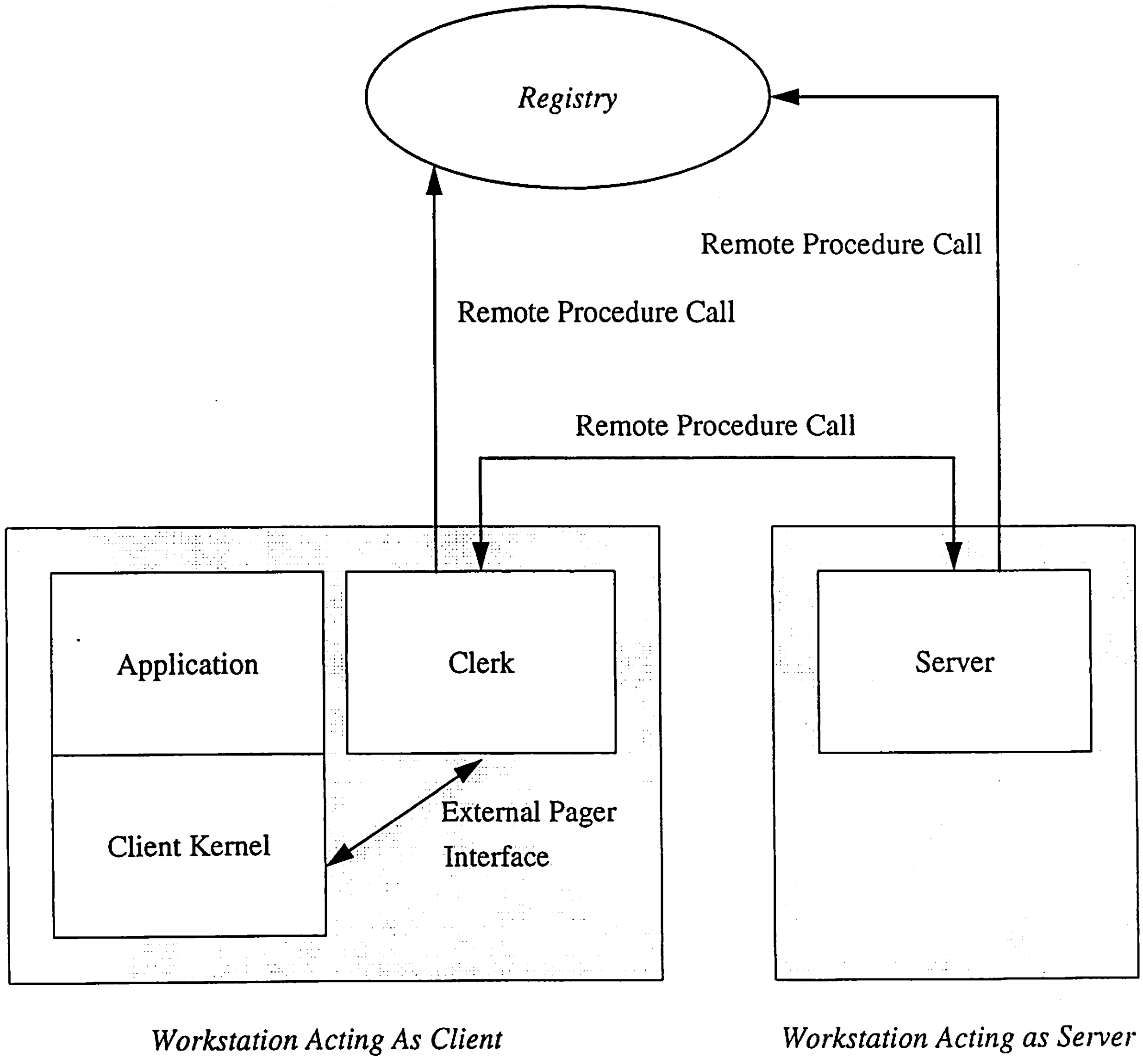

Figure 1: Block diagram of the design of the system.

become active, and de-register themselves if they become dormant again. Each clerk thread queries the registry periodically to obtain a list of active servers.

This system was implemented by making changes to Topaz, the operating system running on our DEC Firefly multiprocessor workstations [TSS88]. Several additions were made to Topaz's virtual memory code. New fields were added to the address space descriptor to support external paging, some initialization code was added, and kernel calls were added to support the external pager interface. The kernel's paging policy is unchanged.

3 Policy Issues

Implementation of a remote memory paging system in a multi-client, multi-server network raises a number of interesting policy questions. Among them are:

- How should a client decide which server(s) to use?
- What happens to a server's pages when the server machine is no longer idle?
- What happens when all servers become full?

We will consider each of these questions in turn, discussing the issues involved and listing several policy alternatives.

3.1 How should a client decide which server(s) to use?

Since clients are not bound to use particular servers, many policy options are available for deciding which server(s) to use. The first time a page is stored externally, the clerk thread on the client machine allocates space for the page in the memory of some server. Subsequent references to the page are addressed to the same server. The clerk may move a page from one server to another, although there is a cost to this operation. We expect that pages will rarely be moved.

Fault tolerance dictates that each client spread its pages across as few servers as possible. If a server machine crashes, the pages it is backing are lost and any threads referencing those pages must be aborted. In order to reduce the chance of such a mishap, clients want to depend on as few servers as possible. (Replication may also provide fault-tolerance, but there is not always room to replicate pages.)

On the other hand, we would like to balance the loads on the servers, in terms of both CPU and memory utilization. A heavily loaded server would have higher response time, and would pose a more difficult problem if the server machine became busy (see Section 3.2).

The policy alternatives for a client include:

1. *Distribute pages equally among all servers.* This policy is simple and achieves good load balance. However, it ignores the fault-tolerance issue. (This may be an acceptable choice, depending on the reliability of the machines involved and the nature of the applications.) It also does not attempt to take advantage of any speed differences among servers. (Again, these differences may be small in practice.)

2. *Find a good server and stick with it.* The client may periodically interrogate the active servers to determine the servers' response time, memory utilization, or other statistics. The client could then

decide which server is currently "best", and use that server exclusively until conditions change. This scheme would tend to use servers with good response times, and would reduce the number of servers on which each client is relying. It might lead to unacceptable load imbalance. Implementation is more difficult than the previous policy, and introduces overhead due to clients' interrogation of servers and the decision-making process. This policy might also lead to "thrashing", when a temporarily attractive server is inundated with new work, only to be overloaded and then abandoned later.

3.2 What happens to a server's pages when the server machine is no longer idle?

Servers are intended to be active only when their machine is idle. When a machine running an active server comes back into use, the server is meant to become dormant. This means it should reduce its load on the machine, and eventually stop storing pages altogether.

It is important that this be done gracefully, especially when the server is a desktop workstation. Ideally, the owner of the workstation would not notice any delay due to the server's existence. In practice, this is not likely since the owner is likely to have left several processes in memory (such as mail-reading tools) and will expect them to still be memory-resident when she returns. If the server uses most of the available memory on the workstation, these idle processes will probably need to be paged back in from disk when the owner returns, perhaps causing a noticeable delay. A rapid and unobtrusive server shutdown process is important for the acceptance of remote memory paging systems.

Several policies are plausible:

1. *Replicate pages to other servers so they can be safely deleted.* If each page is stored on more than one server, any of these servers can (after informing the others) simply delete the page from its memory. After the client has sent a new copy of a page to a server, the server would send a copy of that page to another server; thus the client would not be (directly) affected by replication. A server becomes dormant by informing the holders of all replicas of its pages, then simply deleting the pages. It must remember a forwarding address for each page, and inform the client of the forwarding address; this can be done immediately by a callback, or lazily on the next reference to the page. Replication also makes the system more reliable, since clients can find another copy of their pages if a server crashes.

   This policy allows a server to reduce its memory demands very quickly. However, it makes poor use of server memory space, and causes extra network traffic and CPU load on the servers to keep replication information up to date. Replication is probably only practical in situations of low load, when there is plenty of server memory, network bandwidth, and server CPU capacity.

2. *Migrate pages to other servers.* This is similar to replication, except that the pages are not actually moved until the owning server is about to become dormant. An active server reserves space in the memory of other servers. When a server eventually becomes dormant, it sends its pages to the reserved spaces in one burst of communication. As above, a dormant server must notify clients that their pages have moved.

   This strategy involves some of the overhead of the replication policy, since servers must communicate their obligations to each other. However, these are small messages; memory pages are not sent from server to server except in the rare event that a server becomes dormant. Like replication, migration breaks down when the utilization of server memory becomes high. When this happens, the servers are no longer able to make meaningful commitments to accept each other's pages.

3. *Transfer pages to server disk, then deallocate on reference.* Pages are offloaded slowly from the server in this strategy. First the server moves the pages to disk. Then the server thread continues to run, eliminating each page the next time it is referenced. There are three ways a page can be referenced: read, write, and deallocate. If the next reference is a read, the server returns the contents of the page along with a message indicating that it no longer is responsible for storing that page; the client will then find a new home for the page the next time it is paged out. If the next reference is a write, the server simply refuses to accept the page; the client will find another place for it. If the page is deallocated by the client, then no extraordinary action is required of the server. By this procedure, the server will gradually eliminate itself.

   This scheme has the advantage of not imposing any overhead during normal operation of the system. In addition, the cost associated with the shutdown of the server is spread over time, which may make it less noticeable to other users of the server's machine. The main disadvantage is that it involves greater use of the server's disk — this is the very resource required to bring the workstation owner's jobs back into physical memory.

4. *Transfer pages to client disk.* The client disk is where the pages would have been stored if the remote memory paging system were not running. Since all the mechanism is in place to store pages there, it seems natural to take advantage of this by moving pages from shutting-down servers directly to the client's backing store.

3.3 What happens when all servers become full?

When all available server memory becomes full, policy choices are restricted. Replication is no longer a viable option, and issues of replacement policy come into play. Of these, the more difficult is replacement policy; we will concentrate on this issue here.

Clearly, we want to keep the "hottest" pages in server memory, and put less valuable pages on disk. Because, in contrast to file servers, paging servers are doing only extremely simple memory and network operations, we could afford to use an exact LRU policy for deciding which pages to keep. A daemon thread could run in the background on each server, writing dirty pages to disk. A new page brought into server memory would take the place of the oldest clean page.

Pages which don't fit in server memory can be stored either on the server's disk, or back on the client's disk. It is more convenient to put them on the server's disk, but this makes it harder to shut down the server. In addition, access time is faster for a page stored on the client's disk.

4 Performance

The performance of remote-memory paging can be measured in terms of latency and throughput. Latency is the time required for the basic paging operations on a lightly loaded system; we measured latency by running test programs on an otherwise idle Firefly. Throughput is the amount of useful computation that can be done when the system is fully loaded; we used a queueing model to evaluate throughput under a variety of assumptions about workload and system configuration. [1]

4.1 Latency

Using simple test programs, we measured the pagein time (the delay seen by a user program referencing a non-resident but remotely-stored page) to be 13 milliseconds, a factor of 2.5 faster than the Firefly's paging to disk. About half of this pagein time is accounted for by the remote procedure call used to transfer the data from one machine to another; the other half is due to data checking, the cost of crossing the user-kernel boundary, and other factors.

4.2 Throughput

We constructed a queueing model of a remote-memory paging system, in order to study the performance of remote paging under a variety of assumptions about processor and network speed, number of active clients and servers, etc.

4.2.1 The Model

The modeled system consists of N_c identical clients, N_s identical servers, and a network connecting them. Clients run workloads consisting of a fixed number of application jobs; the application is parameterized by the number of pageins (r_{in}) and pageouts (r_{out}) required per work-unit. (We arbitrarily define a work-unit as the amount of computation that takes one CPU-second on a VAX 3500.) We define α to be the speed of the client CPU; thus the client can execute a work-unit in $\frac{1}{\alpha}$ seconds.

Pageouts are assumed to take place "in the background". Each pageout requires service time of t^{out}_{CPU} on the client CPU, t^{out}_{net} on the network, and t^{out}_{serv} on a randomly chosen server. The application job must wait for pageins; a pagein requires service time t^{in}_{CPU} on the client CPU, t^{in}_{net} on the network, and t^{in}_{serv} on a randomly chosen server.

The network, each client, and each server are modeled as FCFS service centers. Application computing and pagein processing are modeled as a single closed class, and pageout processing is represented as an open class with the arrival rate adjusted so that the correct number of pageouts are serviced.

For reference, we also constructed a model of a single client, using a private disk for paging. This allows us to compare performance of our system with a traditional, disk-based paging system.

The workload shown in Table 1 is based on values measured for a large integer application [FOM]. This application uses a large virtual-memory cache to avoid recalculating values. Accesses to the cache are mostly reads, so the ratio of pageouts to pageins is large (10:1). Applications with a better balance of pageins and pageouts (but the same total number of paging operations) would exhibit slightly better performance.

4.2.2 Results

We evaluated four architectures, consisting of either VAX 3500s or DECstation 5000s (the 5000 is about five times faster than the 3500), interconnected with either 10 Mbit/second Ethernet or a 100 Mbit/second

[1] We would have liked to take a few measurements under full load to validate the queueing model, but unfortunately a bug in Topaz prevented this.

Workload Parameters

Symbol	Meaning	Typical Value
r_{in}	pageins per work-unit	2
r_{out}	pageouts per work-unit	20
J	jobs using remote paging (per client)	1

Architecture Parameters

Symbol	Meaning	Typical Value
N_c	number of clients	5
N_s	number of servers	5
α	client CPU speed	1
t_{CPU}^{out}	CPU time to service a pageout	5.5 msec
t_{net}^{out}	network time to service a pageout	200 μsec
t_{serv}^{out}	server time to service a pageout	5.5 msec
t_{CPU}^{in}	CPU time to service a pagein	5.5 msec
t_{net}^{in}	network time to service a pagein	200 μsec
t_{serv}^{in}	server time to service a pagein	5.5 msec

Table 1: Parameters of the queueing model, and typical values for a set of VAX 3500s connected by an Ethernet.

network such as FDDI. For each of these configurations, we did two experiments. In the first experiment, we held the number of servers constant at five, and varied the number of servers. In the second, we used equal numbers of clients and servers, and varied this number. The results of these experiments are shown in Figures 2 and 3.

We can draw several conclusions from these graphs. First, remote-memory paging shows significantly better performance than disk paging in all models using the faster CPU. Second, network contention is not a serious bottleneck. Network utilization is very low except in the DECstation 5000/ Ethernet configuration with more than 40 clients and servers. The main cause of performance degradation is contention for servers, but even this effect is small since figure 2 shows that five servers can support up to 50 clients with less than a 25% performance degradation.

The queueing model results are extremely encouraging. They indicate that relatively large numbers of clients and servers can coexist on a single network, that the performance advantage of remote-memory paging can be large for a real application, and that this performance advantage grows as CPUs (and networks) get faster. Since we expect networks and CPUs to increase in speed in the future, while disk performance stays relatively constant, this approach should show an increasingly large advantage over disk paging.

5 Related Work

Comer and Griffioen [CG90] have implemented a similar system. They envision a few dedicated memory-server machines, with each client workstation hard-wired to a particular server. Servers have very large memories that are used for storing clients' pages; servers also contain disks which are used if server memory overflows. Comer and Griffioen also devise a network protocol for communication between clients and memory servers.

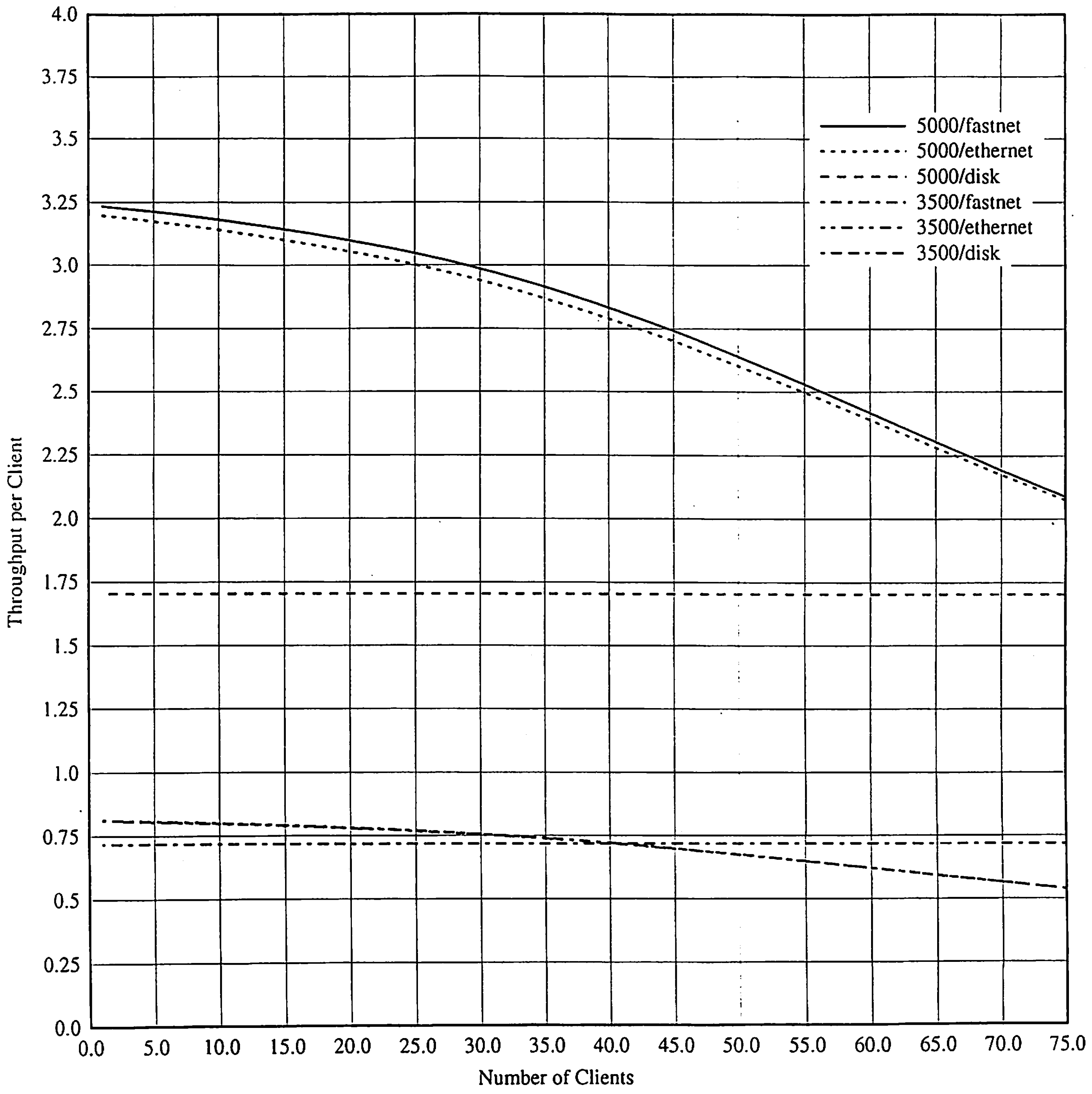

Figure 2: Performance of queueing model of a remote-memory paging system with five servers. Note that the 3500/ethernet and 3500/fastnet curves lie on top of each other.

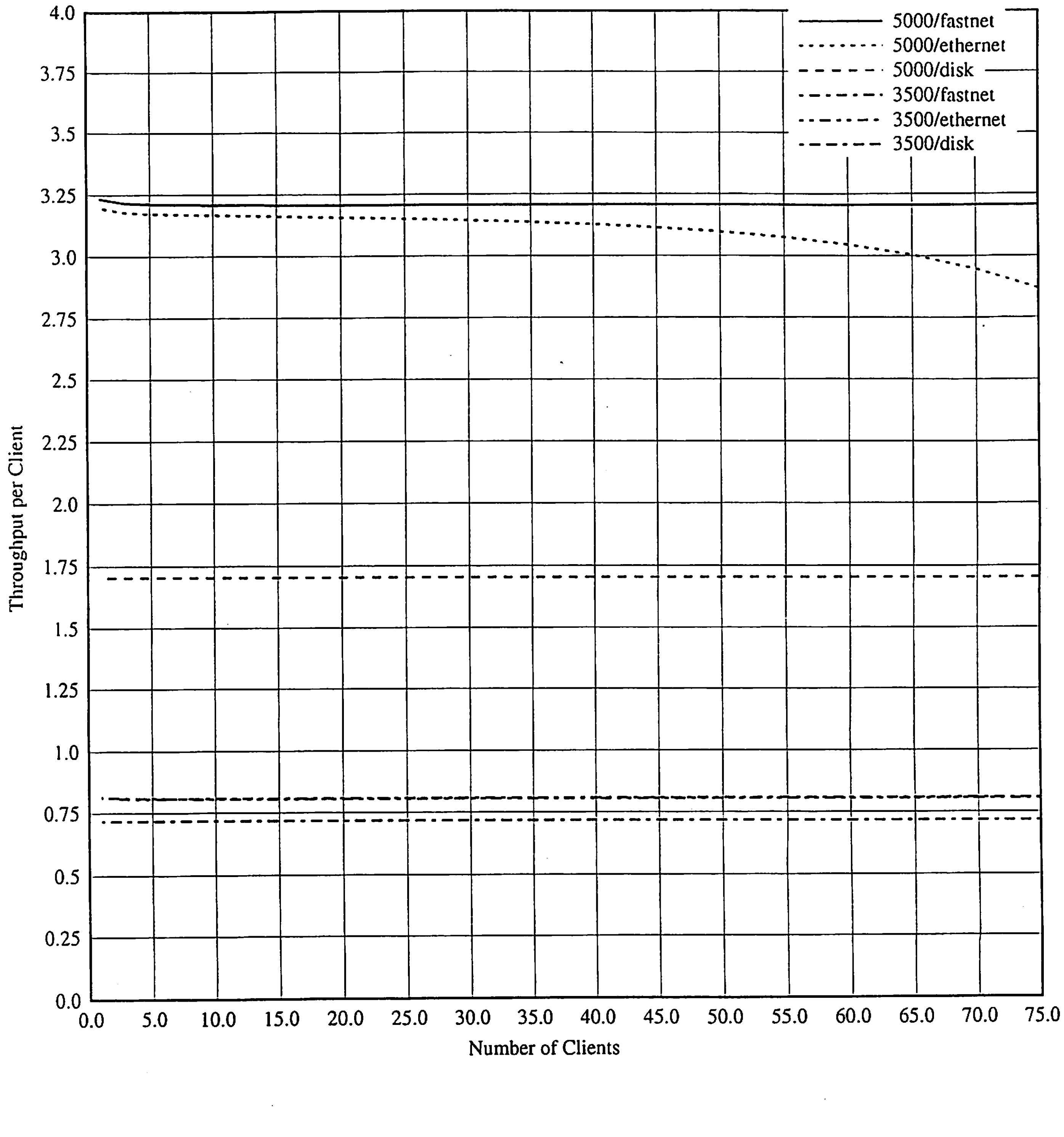

Figure 3: Performance of queueing model of a remote-memory paging system with equal numbers of clients and servers. Note that the 3500/ethernet and 3500/fastnet curves lie on top of each other.

In our scheme, idle workstations volunteer to act as servers, and clients may choose dynamically to use any or all of the active servers. This opportunistic approach allows us to use memory resources which are already in place, rather than requiring the purchase of new, dedicated hardware. On a typical evening in our department, well over a gigabyte of memory is sitting idle. Since memory can account for as much as half of the cost of a workstation, this represents a substantial investment.

As above, there is a good analogy to CPU load sharing techniques. Systems that do load sharing between peer workstations are more flexible and economical than systems that use specialized "compute servers".

6 Conclusions

Remote memory paging is a useful way to avoid the poor performance of disk paging. It provides advantages of performance and simplicity over shared virtual memory systems implementing the same functionality. Remote memory paging is substantially faster than disk paging on our hardware, and technology trends indicate an even larger advantage in the future.

Design of a remote memory paging system raises interesting policy issues. There are several design choices, and more investigation is needed to decide which policies are the best.

7 Acknowledgements

We would like to thank Ed Lazowska for useful comments on an earlier version of this report, and Brian Bershad for helping us understand Topaz and the Firefly system.

References

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Mar. 1990.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comp. Sys.*, 2(1):39–59, Nov. 1984.

[CG90] Douglas Comer and James Griffioen. A new design for distributed systems: The remote memory model. In *Summer USENIX*, 1990.

[FBS89] Allessandro Forin, Joseph Barrera, and Richard Sanzi. The shared memory server. In *Winter USENIX*, 1989.

[FOM] Edward W. Felten, Steve W. Otto, and Olivier Martin. A new monte carlo algorithm for the traveling salesman problem. In preparation.

[LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *ACM Conf. on Principles of Distributed Systems*, pages 229–239, 1986.

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, June 1988.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference on the Management of Data*, pages 109–116, 1988.

[SB90] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SZ90] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 32(5), May 1990.

[TSS88] Charles P. Thacker, Lawrence C. Stewart, and Edward H. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, Aug. 1988.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *ACM Symposium on Operating Systems Principles*, Nov. 1987.