Remote Memory Paging in Networks of Workstations [1]

Guy Bernard and Salima Hamma

Institut National des Télécommunications
9 rue Charles Fourier 91011 EVRY Cedex France
Phone: +33 1 60 76 45 67
Fax: +33 1 60 76 47 80
e-mail: bernard@int-evry.fr, hamma@int-evry.fr

[1] Appeared in the Proceedings of *SUUG International Conference on Open Systems: Solutions for Open World*, Moscow, Russia, 25-29 April 1994

Remote Memory Paging in Networks of Workstations

Abstract

In networks of workstations, the global computing power is most of the time underutilized. "Remote memory paging" consists in using the memory of idle machines on a network as backing store for those machines that are active. This paper provides experimental performance results about remote memory paging in the optimal case (machines used as backing store really idle) and about cross impact between remote memory paging and local processing. In networks of workstations, machines are often dedicated to an "owner". Algorithms are proposed for preserving the interest, in terms of process response times, of both users of machines doing remote paging and owners of machines used as backing store. Whereas our testbed consisted in a network of Sun workstations running SunOS 4.1.2, our methodology for getting basic performance results and setting threshold values can be adapted for tuning and evaluating remote memory paging on any configuration of machines and operating systems.

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 [Felt91].

Traditionally, backing store for virtual memory paging is provided by disks. 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. Another solution is to avoid using disks for paging by using the memory of idle machines on a network as backing store for those machines that are active. We call this strategy "remote memory paging".

In a network of workstations, the global computing power is most of the time underutilized. Reported measurements include 20% for average processor load during the busiest time [Stum88], one third of the workstations completely unused, even at the busiest times of the day [Thei89], well over a gigabyte of memory sitting idle on a typical evening [Felt91]. It is therefore possible most of the time to find an inactive workstation suitable for backing store for an active one. However, since workstations are often dedicated to an owner, the user of a (previously) inactive workstation would not be happy to suffer from long response times because his/her machine is currently used as backing store for another workstation.

Consequently, designing and implementing a remote memory paging facility in a network of workstations should involve two components: *(i)* a *mechanism* for handling page faults, and *(ii)* a *policy* for using this mechanism in an appropriate way, namely in preserving the interest, in terms of process response times, of both users of machines doing remote paging and owners of machines used for remote paging.

Remote memory paging has been studied in several papers. [Come90] describes a mechanism for remote memory paging on a dedicated server machine ("remote memory server") and evaluates an implementation for Sun-3 workstations running the Xinu operating system. [Fori89] describes the design and implementation of a virtual shared memory server for the Mach operating system, with emphasis put on the evaluation of several memory coherency algorithms. [Minn89] and [Minn90] describe the Mether system for distributed shared memory and give implementation details taking benefit from the memory management scheme of SunOS. [Pu90] describes the architecture of a symmetric distributed main memory database, and compares by simulation the effects of several object replacement policies. [Fran92] describes a four-level memory hierarchy (local client memory, server memory, remote client memory and server's disk) for client-server DBMS architectures, and evaluates by simulation several global memory management techniques. [Felt91] raises policy questions for remote memory paging in networks of workstations, but no evaluation of the policies described is given.

This paper addresses the problem of remote memory paging *policies* in a network of UNIX workstations. In Section 2, we describe our testbed and compare the measured cost of disk paging with the measured cost of remote memory paging. In Section 3, we study the cross impact of remote memory paging and local processing. In Section 4, algorithms for deciding to accept remote paging on a machine and for deciding to get rid of remote paging when a machine is used by his/her owner are proposed and implememtation details are given. In Section 5 we discuss our results and present open questions.

2 A Testbed for Remote Memory Paging

In this section we describe the framework of our study, the components of our testbed and the basic results obtained about remote memory paging in a network of UNIX workstations.

We chose to evaluate remote memory paging policies by measurements on a real network of workstations rather than by simulation or analytical modeling. Whereas results obtained this way have no general value, this approach allows to point out the real problems and to give an idea of what really happens. However, for flexibility reasons, we decided to not modify an existing operating system (UNIX in our environment). Experiments were done on a set of Sun workstations running SunOS 4.1.2, interconnected by a standard 10Mbit/s Ethernet.

2.1 Simulating Remote Memory Paging

The first component of our testbed is *FI*, the Fault Interpretation library designed by Edelson [Edel93]. *FI* allows an application running over UNIX to detect all reads and/or writes to selected pages of its virtual space. An access to a protected page causes a fault, which UNIX passes to the application as a signal. The *FI* signal handler unprotects the page and notifies the application of the access. Then, the faulting instruction is restarted; it succeeds because the page is unprotected. Control returns immediately to the *FI* library, which notifies the application again, re-protects the page, and resumes the application at the next instruction. The two functions involved in *pre-access* and *post-access* notification are written by the programmer. *FI* runs in user space. A number of applications of this technique, including garbage collection and consistency/replication protocols, are discussed in [Edel93].

Using *FI* for simulating remote memory paging was straightforward. The monitored address space mimics the physical memory allocated to the process. On page fault, the *pre-access* function just consists in requesting the missing page to the remote memory server (while sending the page to be replaced if this one is modified). The *post-access* function justs sets up the appropriate page protection. Communication between client process and server process is done with the TCP protocol. On the remote memory server side, a daemon process in user space waits for incoming connection requests, allocates the appropriate amount of memory (as indicated in the first message), and manages this memory as a table indexed by page number.

2.2 Obtaining Fine Time Measurements

In our environment (Sun-4 workstations), the resolution of the system clock is 10ms, which is not small enough for measuring non-reproductable times. For example, timing a loop of, say, 1000 page reads, does not provide the real disk access time because of the clustering strategy of disk access in UNIX.

For fine measurements, we used *utimer*, a small kernel module written for Sun SPARCstations by Silvano Maffeis, University of Zurich. It allows reading the spare counter14 chip providing timer-values with granularity of one microsecond. In fact, statistical tests showed that the standard-deviation of reading the counter using *utimer* is around 5 microseconds with a mean of 60 microseconds; this provides a satisfactory precision for monitoring disk or network accesses.

2.3 Basic Results

By using the *FI* library, we did not really implement a remote memory paging mechanism. However, the behavior of our testbed may be considered as close enough from the behavior of a mechanism fully implemented in the kernel, as will be discussed in Section 5. The following results may thus be considered as good approximations.

As a starting point, we measured the read/write page fault resolution delay provided by standard SunOS paging mechanism (SunOS) and by remote memory paging (RMP)[1]. All machines are

[1]In the following, "read" and 'write" have not exactly their usual meaning. Rather, "read" (resp. "write") means

running SunOS 4.1.2, with a page size of 4 kbytes. In what follows, "ELC" stands for diskless Sun SPARCstations (33 MHz clock rate, 16 Mbytes of main memory). For ELCs, standard paging mechanism is done via Ethernet on a dual-processor Sun SPARCserver 670-MP (40 MHz clock rate, 64 Mbytes of main memory, three 1.3 Gbytes SCSI disk drives with 5 Mbytes/s transfer rate each), and remote memory paging is done on another ELC with the same features, acting as a page server. "SP2" stands for Sun SPARCstation2 (40 MHz clock rate, 16 Mbytes of main memory, one 207 Mbytes SCSI disk drive with 4 Mbytes/s transfer rate). For SP2s, standard paging mechanism is done on the local disk, and remote memory paging is done on another SP2 with the same features, acting as a page server. During the experiments, all machines were idle and there was no traffic on the network other than paging. Caches were flushed before the measurements and the clustering mechanism of standard paging was disabled by scanning memory in decreasing address order. The results are reported in Table 1, where page fault resolution delays are given in millisecond per (4 kbytes) page.

	read page fault			write page fault		
	SunOS	RMP	RMP/SunOS	SunOS	RMP	RMP/SunOS
ELC	8.5 ms	6.5 ms	0.76	19.8 ms	9.2 ms	0.46
SP2	5.7 ms	6.8 ms	1.19	8.9 ms	10.6 ms	1.19

Table 1: *Page fault resolution delay*

It appears that remote memory paging can be a worthwhile alternative to standard paging for diskless workstations (page fault delays are reduced by 24% and 54% for read and write, respectively); this is because standard paging involves not only page transfers on the network (as remote memory paging does), but disk accesses too. For SP2s, by contrast, there is no benefit for remote memory paging (delays are increased by 19%): network delays appear to be larger than disk delays on these machines over Ethernet. In the following, we will focus on ELCs only.

However, memory sizes of workstations are physically limited, whereas swap spaces provided by disks can be extended well over the usual needs of regular processes. With remote memory paging, if the size of the memory that is to be paged happened to be larger than the main memory size of the workstation acting as a page server, this machine itself would have to paginate in some way, such that the benefits of remote memory paging would be lost (we will discuss later possible solutions to this problem). Measured read page fault resolution delay versus allocated swap space for remote memory paging with ELCs (the page server ELC itself using standard paging) is given in Figure 1. As could be expected, read page fault resolution time for memory sizes over 16 Mbytes is roughly the sum of remote paging cost (6.5 ms) and standard paging cost (8.5 ms). However, standard paging on the page server ELC occurs before main memory is exhausted. In fact, the maximum memory size that can be used without paging is limited in SunOS 4.1.2 to [$physical\_memory\_size - kernel\_size -$ `lotsfree`], where the UNIX constant `lotsfree` is 256 kbytes . In the following, we will name this maximum value *"available memory"*, which equals roughly 14 Mbytes on our ELCs. Analog results were obtained with write page faults.

3 Cross Impact of Remote Memory Paging and Local Processing

The results just described show that remote memory paging looks very attractive (at least for some hardware configurations) in optimal conditions, namely when the size of the allocated memory is not too large, and when the machine acting as a remote memory server is really idle. In this section we study the cross impact of remote memory paging and local processing ("local processing" means

that the page to be replaced is not (resp. is) modified, thus does not need (resp. needs) to be paged out in swap area before paging in the new page.

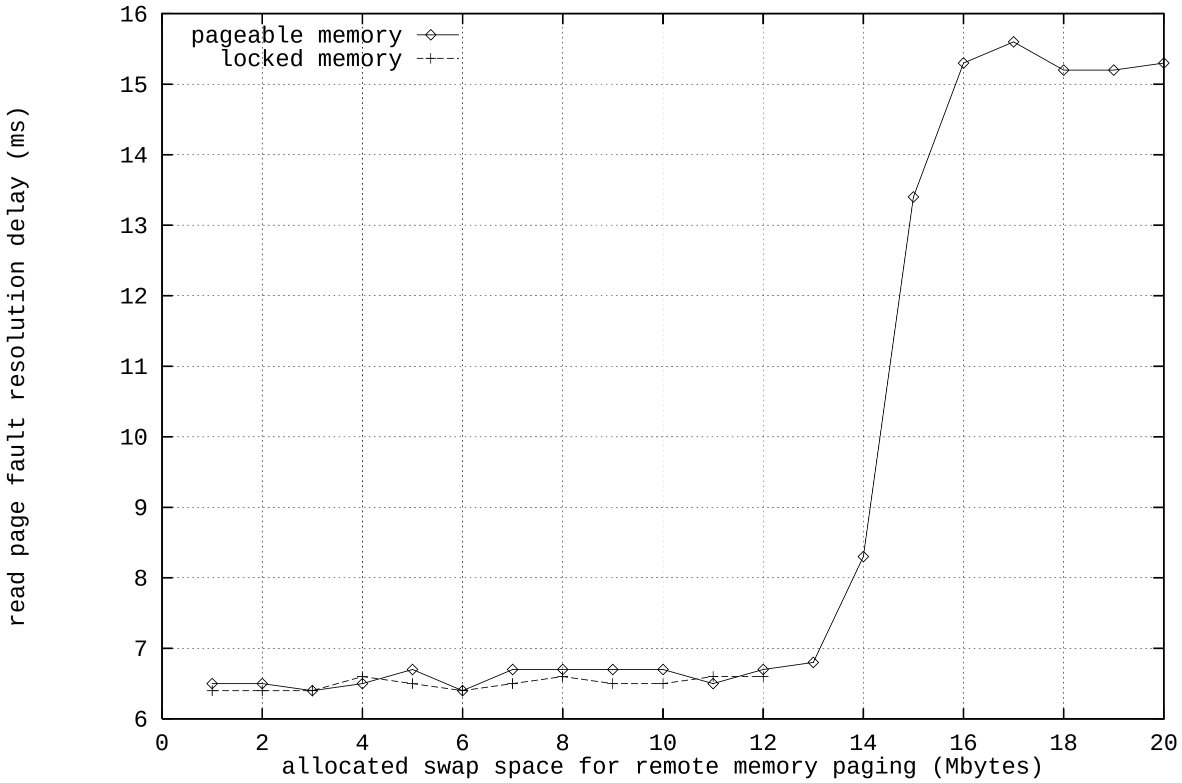

Figure 1: *Read page fault resolution delay with remote memory paging*

computing activity, on the workstation that is used as a remote memory server, for the benefit of the owner of this workstation). We have to answer two questions before being able to define a policy for remote memory paging:

1. what is the impact of remote memory paging on local processing?
2. what is the impact of local processing on remote memory paging?

These impacts should be expressed in terms of influence over process response time, since this criteria is the one directly perceived by users. Answering these questions is quite important. On one hand, if local processing slows down remote memory paging to a value below standard paging performance, then paging should return to standard mode, for the client's benefit. On the other hand, if remote memory paging slows down local processing by an unacceptable factor, then remote memory paging should be abandoned on the server machine, for the owner's comfort.

Three kinds of activities are involved in remote memory paging: *(i)* user programs that use remote memory as swap area, *(ii)* user programs running on a machine whose memory is used as swap area, and *(iii)* system task(s) providing page service on a machine whose memory is used as swap area. In our testbed, the last activity is performed by a standard UNIX daemon process (so called *"server"* in the following). We designed our experiments with one process of type *(i)* (so called *"client"* in the following) running on an ELC and doing remote memory paging on another ELC, and one process of type *(ii)* (so called *"local"* in the following) running on this other ELC.

3.1 Impact of Remote Memory Paging on Local Processing

Experiment 1 studies the impact of the *size* of memory devoted to remote memory paging on the performance of a *local* memory-intensive process. There is no real remote memory paging activity.

Rather, the *server* process just locks some amount of main memory to be used for remote memory paging (via the `mlock()` system call). The *local* process allocates some amount of memory and scans (i.e., just reads the first byte of each page) this memory space in decreasing address order five times. Figure 2 shows the response time of the *local* process (in seconds) versus the size of locked memory for remote memory paging, for various amounts of local memory allocated by the *local* process. Clearly, the memory-intensive *local* process suffers from remote memory paging (even

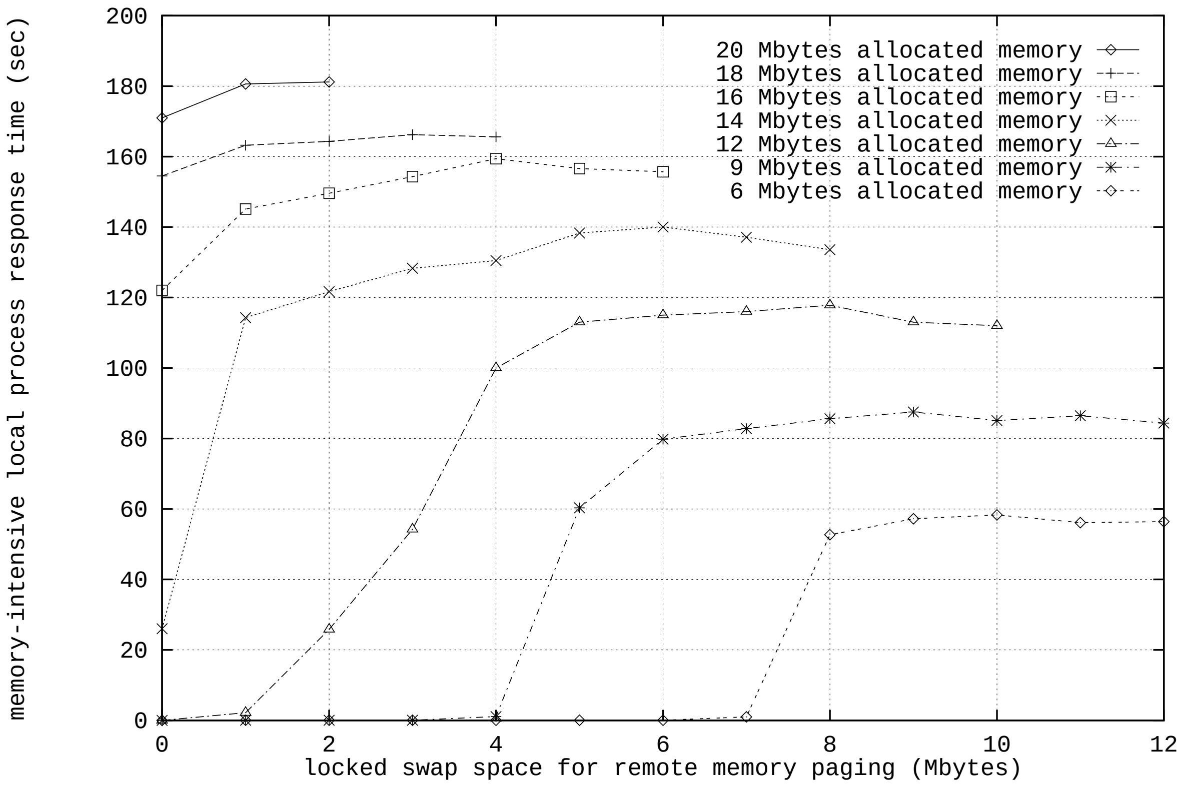

Figure 2: *Impact of the size of locked memory for remote memory paging over a memory-intensive local process*

when the *server* process is inactive), when it would nor paginate if it were alone, but does paginate because the total memory allocated (both for remote memory paging and for local processing) exceeds the *"available memory"* as defined earlier (eg., 9 Mbytes of memory for local processing together with 8 Mbytes of locked memory for remote paging). By contrast, when the memory-intensive *local* process would paginate if it were alone, it does not suffer much from the lack of memory due to remote memory paging (eg., the maximum penalty is 7% for 18 Mbytes of memory allocated for local processing).

Experiment 2 studies the impact of remote memory paging *activity* on the performance of a *local* CPU-intensive process. The *local* process justs executes a large number of times an empty loop. The *client* process makes remote memory accesses (read or write page faults) at a variable rate (page faults interleaved with an empty loop of variable size). The memory used by the *server* process is locked. As can be seen on Figure 1 (dotted line), this does not affect the response time of the *client* process. Figure 3 shows the normalized response time of the *local* process versus the throughput (page faults served per second) of the *server* process. Clearly, the CPU-intensive *local* process suffers from remote memory paging activity, with a somewhat smooth degradation. For instance, a remote memory paging activity of 40 pf/s (read) or 17 pf/s (write) leads to a 10% increase in response time for the *local* process.

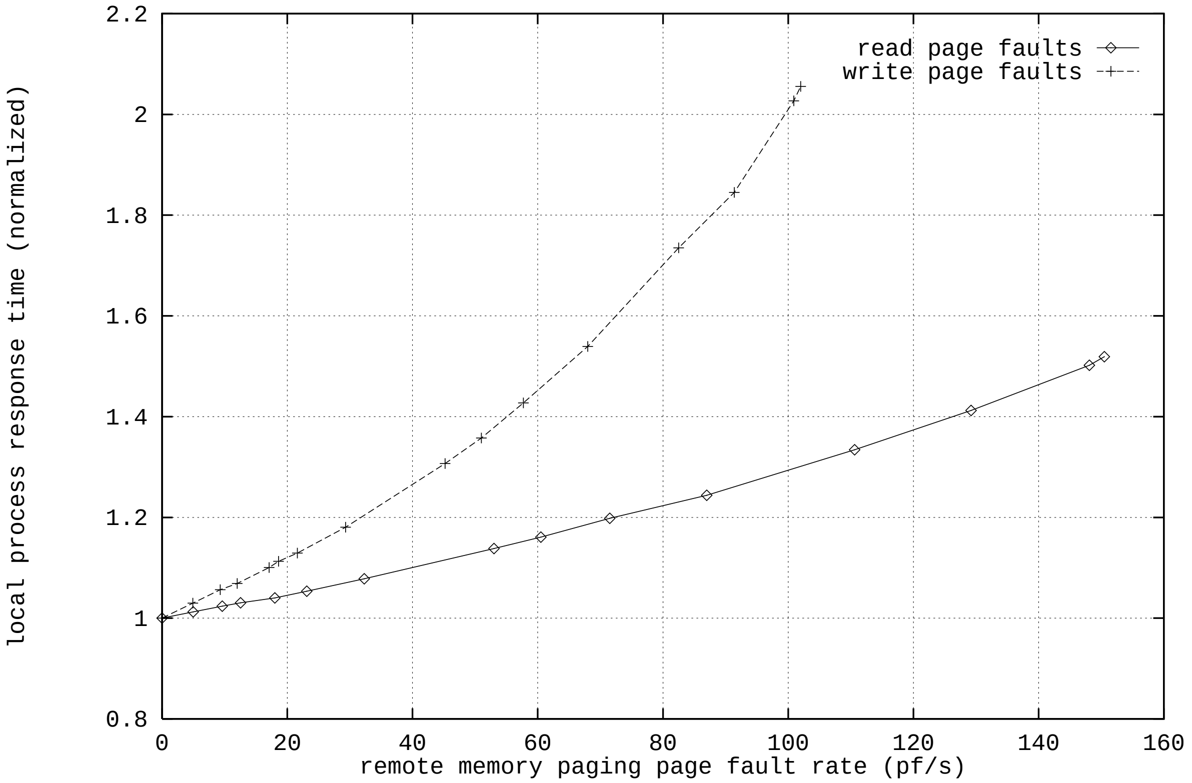

Figure 3: *Impact of remote memory paging activity over a CPU-intensive local process*

3.2 Impact of Local Processing on Remote Memory Paging

Experiment 3 studies the impact of a *local* memory-intensive process on the performance of a *client* process. The *local* process scans a large amount of memory (i.e., does standard paging) at a variable rate (page faults interleaved with an empty loop of variable size). The *client* process makes remote memory accesses (read page faults) as fast as possible (i.e., its response time depends on the throughput of the *server* process). Figure 4 shows the read page fault resolution delay experimented by the *client* process, and Figure 5 shows the throughput of the *server* process, versus the page fault rate of the *local* process. Clearly, the *client* process suffers from (standard) paging of the *local* process: as soon as the local page fault rate exceeds 30 pf/s, the read page fault resolution delay experimented by the *client* process exceeds 8.5 ms, thus the benefit of remote memory paging is lost. This is confirmed by Figure 5: a local page fault rate of 30 pf/s limits the *server* throughput to 117 pf/s, that is 8.5 ms per page fault.

Last, experiment 4 studies the impact of a *local* CPU-intensive process on the performance of a *client* process. The *local* process consumes a variable amount of CPU cycles (empty loop interleaved with sleeping). The *client* process makes remote memory accesses (read page faults) as fast as possible. As before, the memory used by the *server* process is locked. Figure 6 shows the read page fault resolution delay of the *client* process versus the amount of CPU cycles that the *local* process would consume when running without remote memory paging activity. It can be seen that the read page fault resolution delay of the *client* process is not influenced by the amount of local CPU activity; this is because the *server* process itself consumes very few CPU cycles. Notice however that the read page fault resolution delay is now (in average) 6.9 ms, in place of 6.5 ms when there is no local process running. This difference comes from context switching overhead between *local* and *server* processes.

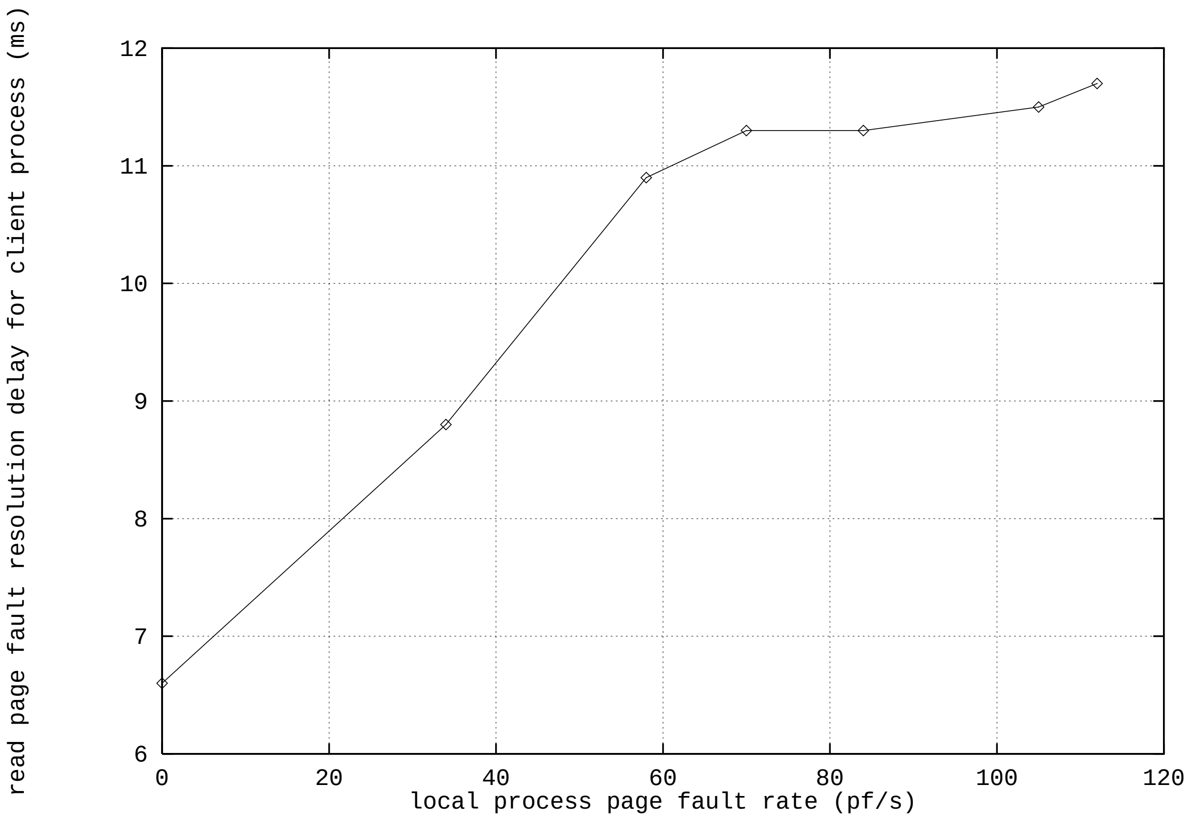

Figure 4: *Impact of a local memory-intensive process over client process read page fault resolution delay*

4 Algorithms and Implementation

In this section we rely on the preceding experimental results for designing a *policy* for using remote memory paging, and we give some implementation details.

4.1 Algorithms for Remote Memory Paging Policy

The goal of the policy is to preserve the interests of both client users (the ones whose processes use remote memory paging) and local users (the ones whose machine's main memory is used for remote memory paging). The criteria for preserving the interest of the client user are straightforward: as soon as the page fault resolution delay experimented with remote memory paging exceeds the page fault resolution delay experimented with standard paging, remote memory paging is no longer worthwhile. Preserving the interest of the local user is more controversial and, in fact, is a "political" choice: how much trouble is the owner of a workstation ready to suffer? This problem has been studied in the load sharing research area. Several ways for taking into account the personal ownership of workstations have been proposed. In Butler [Nich90], a workstation is considered as busy (unavailable for executing foreign processes) as soon as the number of logged in users is greater than some threshold. This criteria is very restrictive, because if users neglect to log out when they do not use the workstation, it will appear busy whereas it is in fact available. In Condor [Litz88], a workstation is declared available when there have been no keyboard or mouse activity for some duration, and the average CPU utilization has been less than some value for a certain amount of time. This criteria is very restrictive too, because if the workstation is used only to run a text editor, it will appear as busy, while running a foreign process would not slower the editing process significantly. In RADIO [Bern91], as in [Alon88], a workstation is declared available

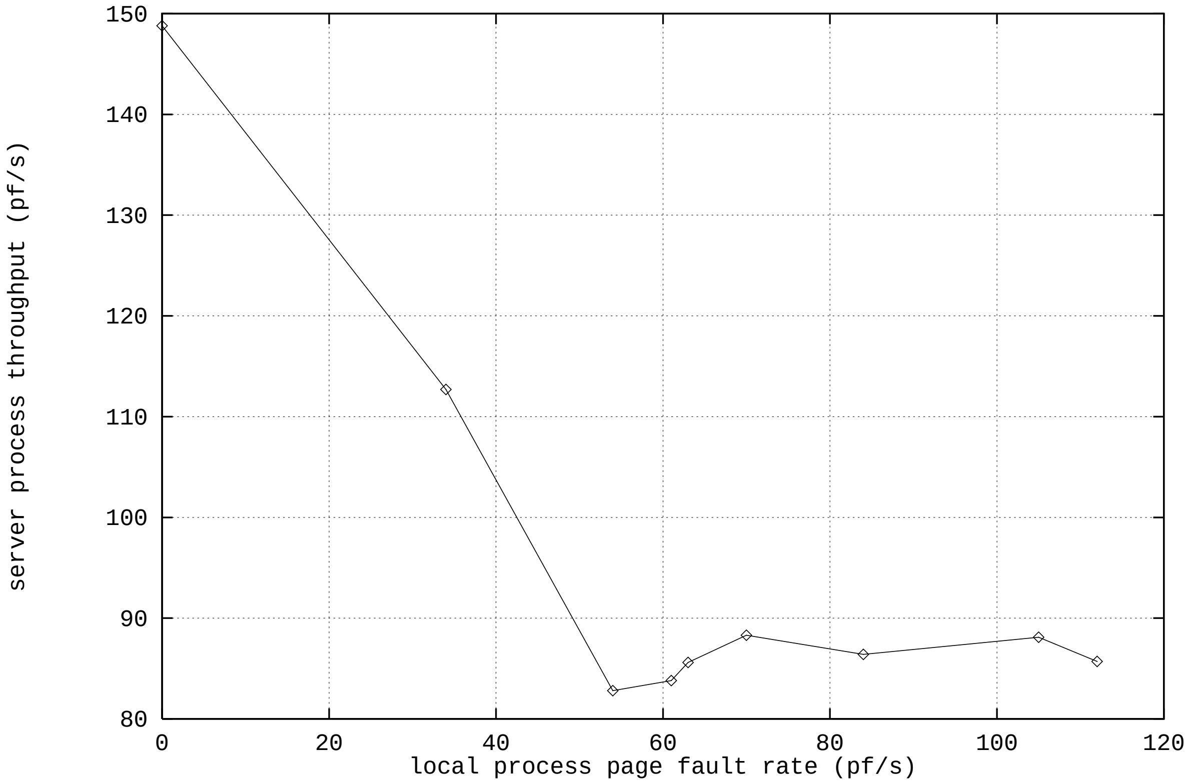

Figure 5: *Impact of a local memory-intensive process over client process maximum page fault rate*

when its current load is less than some threshold on UNIX 4.3bsd load index, the threshold being empirically determined as being the beginning of the perception of foreign activity by workstations' owners.

In the same way, we argue that local processing and remote memory paging can coexist to some extent. In experiment 1, we have shown that memory-intensive *local* processes that allocate a large amount of memory suffer a small performance degrading because of remote memory paging (eg., the maximum penalty is 7% for 18 Mbytes). Let us call `LOCAL_ALLOC_MEM` the memory size corresponding to, say, 10% degradation (roughly 17 Mbytes on our ELCs). In experiment 2, we have shown that CPU-intensive *local* processes can bear a limited remote memory paging activity with an acceptable performance degrading. Let us call `RMPFR` the threshold corresponding to, say, 10% degradation (40 read page faults or 17 write page faults per second on our ELCs). Finally, in experiment 3, we have shown that there is a threshold for local page fault activity under which remote memory paging provides better performance than standard paging. Let us call `LPFR` the local page fault rate corresponding to, say, 10% degradation (30 read page faults on our ELCs).

We therefore propose the following algorithm (cf. Figure 7). Assuming remote memory paging is in effect on a workstation, the question of whether remote memory paging should be continued is periodically examined. Of course, if no local processes other than the *server* process (and "system" processes) are currently running, remote memory paging is continued. Else, three system facts are examined: local page fault rate (`lpfr`), remote memory paging fault rate (`rmpfr`) and main memory allocated to local processes (`local_alloc_mem`), and compared to their respective threshold (in capital cases)[2]. If there are any local processes running but these do not paginate,

[2]In fact, there should be two remote memory paging fault rates and two local page fault rates, one for read faults and one for write faults. Only one is mentioned here for clarity.

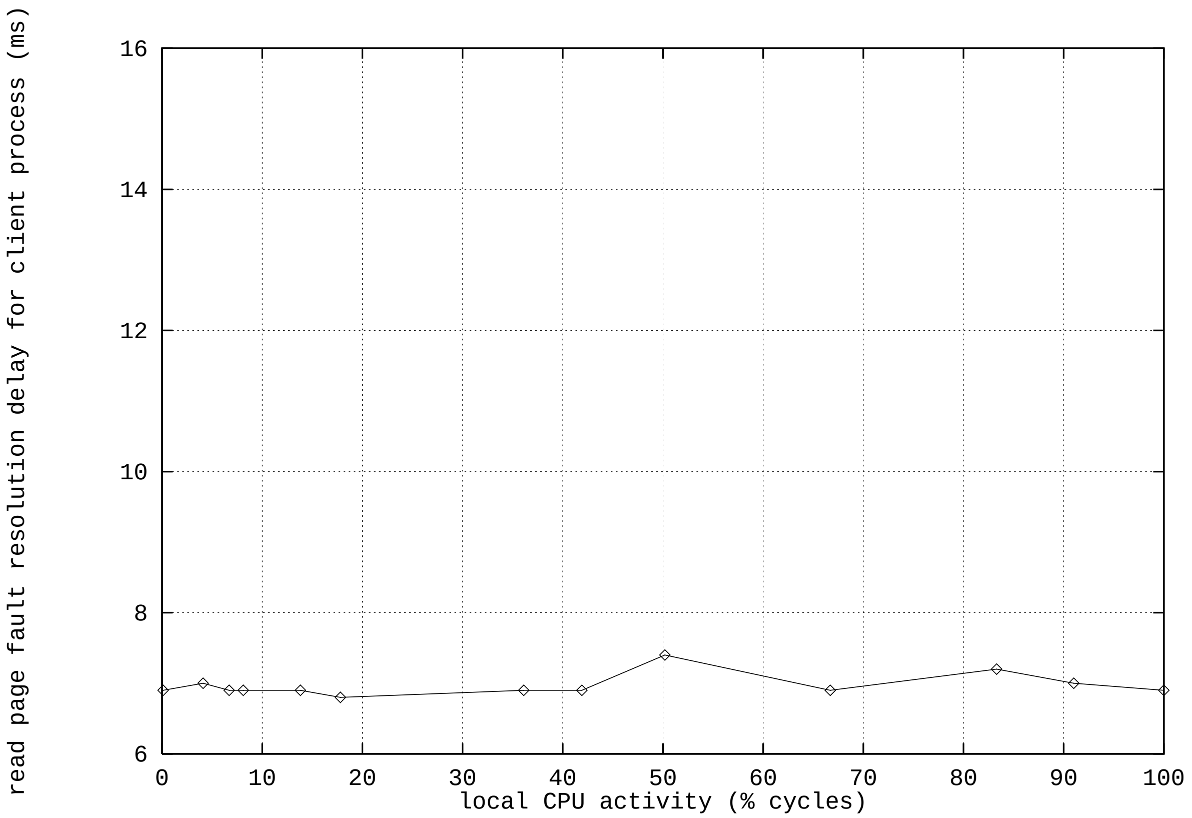

Figure 6: *Impact of local CPU activity over client process read page fault resolution delay*

then the only useful criteria is the remote memory paging fault rate.

For deciding to accept remote memory paging of a new process on a workstation, the algorithm is the same, except that the last test disappears.

4.2 Implementation Details

We describe here how our testbed is implemented. On every workstation, a *monitor* process gathers periodically the number of *local* processes running (if any) and the values of the system facts mentioned above, and a *listener* process waits for new remote memory paging allocation requests. For each accepted request, a *server* process is forked, which is dedicated for the *client* process paging. All these processes run in user space. The *monitor* process signals the *listener* process when no new remote memory paging is possible, and signals the *server(s)* process(es) when current remote memory paging should be abandoned. System facts are obtained by examining kernel structures with filtering upon *local* activity as defined earlier. The *monitor* process awakes once per minute only in order to both minimize overhead and smooth sporadic local activity (if any), since the owner may suffer degraded performance a few tenths of seconds and stopping remote memory paging is a rather heavy decision that should not be taken prematurely. The overhead caused by the *monitor* process is very low (less than 1% CPU cycles).

Communication between a *client* process and its peer *server* process is achieved with TCP. This protocol gives roughly the same response time as UDP for transmitting 4 kbytes pages, but reliability and failure detection are provided. Message headers carry the type of access (read or write) and page addresses. For "write" faults, a single message containing the requested page number followed by the content of the victim page is sent by the *client* process to the *server* process. The latter receives the first part of the message and sends the requested page to the *client*

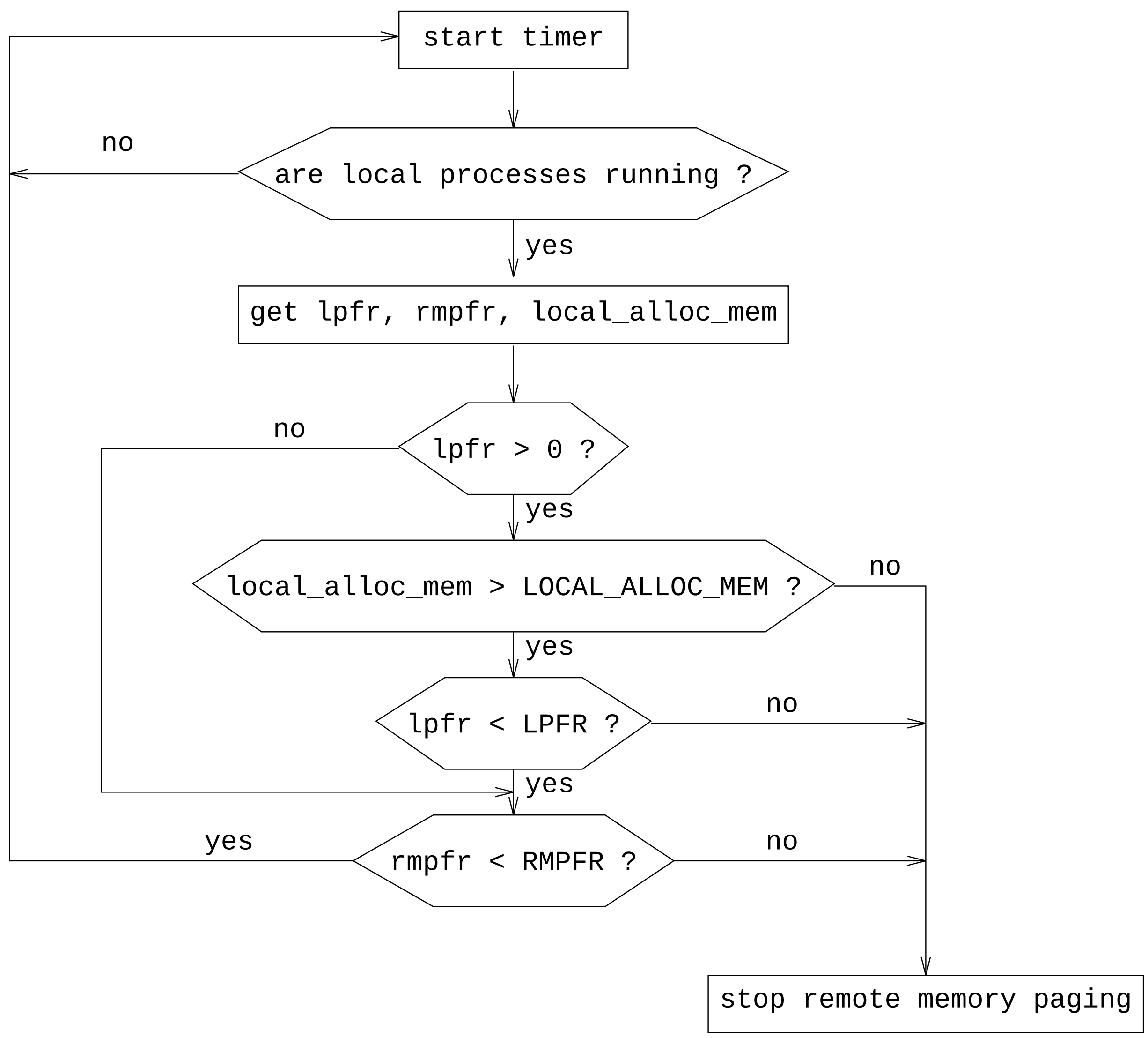

Figure 7: *Algorithm for remote memory paging policy*

before reading the remaining of the message and storing this page at the right place in memory. This way, the page fault resolution delay is minimized.

5 Discussion and Open Questions

Our testbed is not a real implementation of remote memory paging. Rather, it simulates the behavior of a full remote memory paging facility. In this section we discuss the validity of the results obtained with this simulator.

[Minn89] reports that the overhead for access to a page not on the local processor is about equivalent to a page fault on Sun workstations, but communication is via UDP with fragmentation and re-assembly of packets and no figures are given. [Come90] indicates that storing or retrieving an 8 kbytes segment between a Sun 3/50 client and a Sun 3/50 remote memory server requires an average of 39 ms, whereas paging over NFS requires 50 ms. [Felt91] reports that with DEC Firefly's running a modified version of Topaz, the pagein time with remote memory paging is 2.5 times faster than standard paging to disk.

These results are coherent with ours. Thus we believe that our proposed policy would be realistic in a real implementation. On one hand, what is not taken into account in our testbed is the overhead of running the kernel code for selecting a suitable frame and for updating the page table. On the other hand, an implementation of *client* and *server* page fault handling code into the kernel would make page fault resolution delay shorter. We are optimistic about the global bill.

Our numerical results should not be taken with their absolute values. First, they are machine- and system-dependent. Second, setting some of the thresholds indicated in the previous section is a "political" decision. The 10% degradation is just an example. However, our methodology for getting basic results and cross impact of remote memory paging and local processing can be adapted to any configuration in order to obtain the right values.

A number of questions remain opened, which we have not examined yet:

1. What to do when remote memory paging should be abandoned on a workstation? Our results show that the CPU consumption of the *server* process is low, such that this process could duplicate the memory space that it manages on another (mirror) idle workstation without penalizing significantly neither *local* processes nor the *client* process. This would provide both a solution to workstation overloading and an open door for fault tolerance and process migration, but this remains to be evaluated.

2. When several *server* processes run simultaneously on a workstation and when this machine becomes overloaded, which *client* process should be the victim? Drastic remedy (all processes, or at least the one with the heaviest impact – "heaviest" to be defined), or homeopathic medicine (the "lightest" *client* process first)? Clearly, more work is needed in this area.

3. Should any process be candidate for remote memory paging? This question is close from that of "filtering" for load sharing. In this area, several policies have been considered: "manual filtering" (the user has an idea of the behavior of the process which he/she starts), "name filtering" (a number of shell commands are eligible for process migration), or "no filtering" [Bern93]. The pitfall here is the overhead for finding a suitable workstation for remote memory paging.

6 Conclusion and Future Work

In this paper we have shown that remote memory paging can be a worthwhile alternative to standard disk paging. This is certainly the case for diskless workstations, for which page faults involve a network access and maybe a disk access. In the current state of technology (CPU speed compared to network speed compared to disk speed), the advantage is not clear, at least for the machines (Sun SPARCstations 2), the operating system (SunOS 4.1.2) and the network (Ethernet) that were used for our experiments. But technology trends indicate a larger advantage in the future [Felt91].

We have proposed a policy for remote memory paging. We have shown that it is possible to conciliate the interest of the user who needs a fast swap space and the interest of the owners of workstations used for remote memory paging. Our methodology can be used for tuning and evaluating remote memory paging on any configuration of machines and operating systems.

Future work will be undergone in several directions in the context of UNIX:

1. clustering for sequential memory accesses: this strategy in used in Berkeley UNIX in order to minimize disk accesses. It can be easily transposed in network accesses;

2. finding an idle workstation for remote memory paging: we will adapt an algorithm used for load sharing [Bern91] to this goal;

3. handling local processing that is not for the benefit of the owner, such as remote login sessions;

4. evaluating remote memory paging mirroring;
5. server partitioning for very large client allocated memories that do not fit in a single workstation physical memory;
6. adapting Mether-NFS [Minn93] for remote memory paging.

Outside the context of UNIX monolithic kernel, we will investigate remote memory paging for microkernel-based operating systems, where the concept of "external pager" gives a sound base for flexible schemes in this area.

7 Acknowledgments

We would like to thank Daniel Edelson and Ron Minnich for their helpful suggestions and the fruitful discussions.

References

[Alon88] R. Alonso and K. Kyrimis. A Process Migration Implementation for a UNIX System. In *Proc. USENIX Winter '88*, Dallas, Texas, February 1988.

[Bern91] G. Bernard and M. Simatic. A Decentralized and Efficient Algorithm for Networks of Workstations. In *Proc. EurOpen Spring '91*, Tromso, Norway, 20-24 May 1991.

[Bern93] G. Bernard, D. Stève, and M. Simatic. A Survey of Load Sharing in Networks of Workstations. *Distributed Systems Engineering Journal*, 1993. to appear.

[Come90] D. Comer and J. Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proc. USENIX Summer '90 Conference*, Anaheim, California, June 11-15, 1990.

[Edel93] D.R. Edelson. Fault Interpretation: Fine-Grain Monitoring of Page Accesses. In *Proc. USENIX Winter '93 Conference*, San Diego, CA., January 25-29, 1993.

[Felt91] E.W. Felten and J. Zahorjan. Issues in the Implementation of a Remote Memory Paging System. Technical report no. 91-03-09, University of Washington, 1991.

[Fori89] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. In *Proc. USENIX Winter '89 Conference*, San Diego, CA., January-February 1989.

[Fran92] M.J. Franklin, M.J. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, August 1992.

[Litz88] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th Int. Conf. on Distributed Computing Systems*, San Jose, California, June 1988.

[Minn89] R.G. Minnich and D.F. Farber. The Mether System: Distributed Shared Memory for SunOS 4.0. In *Proc. USENIX Summer '89 Conference*, Baltimore, Maryland, June 1989.

[Minn90] R.G. Minnich and D.F. Farber. Reducing Host Load, Network Load and Latency in Distributed Shared Memory. In *Proc. 10th Int. Conf. on Distributed Computing Systems*, Paris, France, June 1990.

[Minn93] R. Minnich. Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory. In *Proc. 4th Symp. on Experiences on Distributed and Multiprocessor Systems (SEDMS-IV)*, San Diego, CA., September 20-22, 1993.

[Nich90] D.A. Nichols. *Multiprocessing in a Network of Workstations*. PhD thesis, Carnegie-Mellon University, February 1990. Available as CMU Report CMU-CS-90-107.

[Pu90] C. Pu, A. Leff, F. Korz, and S. Chen. Redundancy Management in a Symmetric Distributed Main-Memory Database. Technical report no. cucs-014-90, Columbia University, 1990.

[Stum88] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *2nd IEEE Conference on Computer Workstations*, March 1988.

[Thei89] M.M. Theimer and K.A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Trans. on Software Engineering*, 15(11), November 1989.