The Utility of Exploiting Idle Memory for Data-Intensive Computations

Anurag Acharya
Dept. of Computer Science
University of California
Santa Barbara, CA 93106

Sanjeev Setia
Dept. of Computer Science
George Mason University
Fairfax, VA 22030

Abstract

In this paper, we examine the utility of exploiting idle memory in workstation pools. We attempt to answer the following questions. First, given a workstation pool, what fraction of the memory can be expected to be idle? This provides an estimate of the opportunity for hosting guest data. Second, what fraction of a individual host's memory can be expected to be idle? This helps determine the recruitment policy – what is the maximum amount of memory that should be recruited on a single host? Third, what is the distribution of memory idle-times? That is, what is the probability that a chunk of memory that is currently idle will be idle for longer than time t? This information indicates how long guest data can be expected to survive; applications that access their data-sets frequently within the expected life-time of guest data are more likely to benefit from exploiting idle memory. Fourth, how much benefit can a user expect? We use two metrics for the benefit of exploiting idle memory: (1) if I have a pool with w workstations, how much memory should I expect to get for free by harvesting idle memory; (2) how much improvement can be achieved in end-to-end execution time? Finally, how long and how frequently might a user have to wait to reclaim her machine if she volunteers to host guest pages on her machine? This helps answer the question of social acceptability. To answer the questions relating to the availability of idle memory, we have analyzed two-week long traces from five workstation pools with different sizes, locations, and patterns of use. To evaluate the expected benefits and costs, we have simulated three data-intensive applications (0.5 GB-5 GB) on these workstation pools.

1 Introduction

Idle workstations have traditionally been harvested for their cycles. Exploiting idle workstations for hosting guest computation has been a popular research area. Systems that utilize idle workstations for running sequential jobs have been in production use for many years (e.g. Condor [15], which manages about 300 workstations at the University of Wisconsin); there has also been significant amount of research on exploiting idle workstations for hosting parallel computations [1, 3, 5, 6, 16, 19].

With the current growth in the number and the size of data-intensive tasks, exploiting idle workstations for their *memory* is an attractive option. This would be particularly attractive for programs with transient peaks in their memory requirements or for data-intensive programs that are run infrequently. Harvesting idle memory is potentially easier than harvesting idle processors since eviction of guest data usually does not have the same urgency as the eviction of guest computation. Memory can potentially be shared for long periods without significant impact on the interactive performance, particularly on today's machines with their large primary memories.

In this paper, we examine the utility of exploiting idle memory in workstation pools. We attempt to answer the following questions. First, given a workstation pool, what fraction of the memory can be expected to be idle? This provides an estimate of the opportunity for hosting guest data. Second, what fraction of a individual host's memory can be expected to be idle? This helps determine the recruitment policy – what is the maximum amount of memory that should be recruited on a single host? Third, what is the distribution of memory idle-times? This indicates how long guest data can be expected to survive; applications that access their data-sets frequently within the expected life-time of guest data are more likely to benefit from exploiting idle memory. Fourth, how much benefit can a user expect? We use two metrics for the benefit of exploiting idle memory: (1) if I have a pool with w workstations, how much memory should I expect to get for free by harvesting idle memory; (2) how much improvement can be achieved in end-to-end execution time? Finally, how long and how frequently might a user have to wait to reclaim her machine if she volunteers to host guest data on her machine? This helps answer the question of social acceptability – whether users will feel comfortable enough with the performance of the system to agree to participate.

To answer the questions relating to the availability of idle memory, we have analyzed two-week long traces (Sep 14-27, 1997) from five workstation pools with different sizes (14-50 machines), different locations (Comsearch Inc., University of Maryland, George Mason University) and patterns of use (shared cluster, private workstations, compute cluster). To evaluate the expected benefits and costs, we have simulated the execution of three data-intensive applications (out-of-core LU decomposition, decision support queries on a sales database and data-mining for buying patterns in retail transactions) on these workstation pools. To drive these simulations, we used traces captured by running these applications on realistic workloads [22]. The datasets for these applications vary from 0.5 GB to about 5 GB.

Previous research into using idle memory for hosting guest data has taken one of three approaches. Leff et al [14] use an analytical and simulation model based approach to study the feasibility of remote caching architectures. Their study uses synthetic models of both memory availability and program behavior. It is difficult to draw conclusions about behavior of real data-intensive programs on real workstation pools from their work. Dahlin et al [8] and Sarkar et al [20] propose schemes to use idle memory to increase the effective file cache size. Both these studies use the Sprite distributed file system traces [4] to determine both memory availability and program behavior. The Sprite traces were acquired from a cluster of workstations used for computer science research and as such are not focused towards data intensive applications. In fact, there is no information about the identity or the memory footprints of individual applications. Feely et al [9] describe a low-level global memory management system that uses idle memory to back up not just file pages but all of virtual memory as well. They show that this scheme is able to use idle memory to improve the performance of a suite of data-intensive tasks. While they study the performance of real data-intensive applications they have used synthetic workloads to model memory availability. While their results showed that it is possible to achieve significant performance improvement by exploiting idle memory in a networked workstation pool, their study provides no information either about the availability of idle memory in typical workstation pools or about the performance of data intensive applications on such pools. In our study, we have used detailed memory usage and user activity traces from several workstation pools to determine memory availability. We have used actual traces from three data-intensive applications running realistic workloads; the size of the datasets of these applications are significantly larger than those of the applications used by Feely et al [9].

We first examine the memory availability questions. Section 2 describes the methodology used to monitor the memory usage in a workstation pool, and the metrics computed to estimate the opportunity for hosting guest memory. Next, we examine the benefits and costs of exploiting idle memory for data-intensive computations. In Section 3, we describe the framework assumed by our simulation experiments, the applications used in the study, and the system and workload models used in our simulations. Section 4 presents our results. In addition to studying the utility of exploiting idle memory, we also study the impact of technological improvements (network and disk bandwidth) and recruitment policies for idle memory. Finally, we present a summary of our conclusions.

2 Memory availability

To determine the availability of idle memory for hosting guest data, we have analyzed two-week traces from five workstation pools. For each of these traces, we have computed two metrics. First, how much of the total memory in the pool is idle. We refer to this as the *aggregate-availability* metric. This provides an estimate of the opportunity for hosting guest data. Second, we computed what fraction of the workstations have at least $x\%$ of their memory idle. We refer to this as the *fractional-availability(x)* metric. This helps determine the recruitment policy – what is the maximum amount of memory that should be recruited on a single host? We computed this metric for several values of x ranging from 33% to 66%. In addition, we have computed the distribution of memory idle-times for all the pools. We first describe the traces. We then present the memory-availability metrics and the distribution of memory idle-times for these traces.

2.1 Traces

For each workstation pool, we collected two kinds of traces – workstation activity traces and memory usage traces. We collected user activity traces by checking the keyboard and mouse device files every 90 seconds. In addition, we collected load information every 90 seconds. For the purpose of these experiments, a workstation was considered idle if there was no user activity *and* the load average was below 0.3 for the previous five minutes.

We collected the memory usage traces by running the `top` utility at 90 second intervals. We computed a conservative estimate of the amount of idle memory on a workstation by computing an upper bound on the amount of physical memory used by *active* processes and subtracting this from the total physical memory available on the machine. We used the value reported in the `RSS` field as the amount of physical memory used by the process. We considered the following the processes to always be *active*: the operating system, the X server, and all periodic daemons. In addition to these eternally active processes, processes that were reported by `top` to consume more than 0.01% of the CPU were also considered to be *active*. Furthermore, every time a workstation became idle, we *looked ahead* in the trace to find the processes that were *active* when the workstation became busy again. These processes were considered to be *active* for the entire period while the workstation itself was marked idle. Our goal was to estimate how much memory could be harvested without causing a pause when the workstation becomes busy. This is a conservative estimate of idle memory for two reasons: (1) it assumes that every *active* process uses all its physical memory in every interval it is *active*; and (2) the resident set size (RSS) information does not account for shared libraries. Thus the shared libraries are counted for each active process resulting in an overestimate of the actual resident memory.

The first trace, `alpha`, is from a cluster of 4-processor DEC Sables at UMIACS in the University of Maryland. This trace contains data for 10 nodes. This cluster is used by several departments for large/long-running programs. This cluster is used both interactively and in batch-mode. In batch-mode, up to 8 nodes can be allocated for exclusive use. When a node is in batch-mode, its resources are not available for the other users. The total physical memory in this cluster was 2560 MB. The second trace, `umdres`, is from the workstation cluster of a research group at the Department of Computer Science, University of Maryland. This trace contains data for 14 workstations. These machines are used by faculty and graduate students for research and personal purposes. The total physical memory in this cluster was 864 MB. The third trace, `gmu`, is from the group of personal and public workstations at the Department of Computer Science, George Mason University. This trace contains data for 32 workstations. These machines are used by faculty and graduate students for research and personal purposes. The total physical memory in this cluster was 1248 MB. The fourth trace, `comsearch`, is from the workstation cluster of Comsearch, a Reston VA company specializing in wireless communications. This trace contains data for 37 workstations. These machines are used by engineers and software developers. The total physical memory in this cluster was 3200 MB. The fifth trace, `umdcomm`, is from the public workstation cluster of the Department of Computer Science, University of Maryland. This trace contains data for 50 workstations. These machines are used by junior computer science graduate students for course assignments and personal purposes. The total physical memory in this cluster was 2608 MB.

2.2 Memory availability metrics

Figure 1 presents the *aggregate-availability* metric for all traces. Each graph shows how the metric varies over the two-week period. We note that, while 70%-85% of the total memory is available on the average, 60% of total memory is (almost) always available. All graphs show the expected daily and weekly patterns – weekdays are indicated by the dips; nights by the humps and weekends by the shorter-than-usual dips. In each graph, the horizontal line labeled `avg` shows the average fraction of the total memory that is idle. The graph for `alpha` shows irregular patterns due to the impact of batch-mode reservations. The sharp dip in the graph *comsearch* was caused by a failure that resulted in all the workstations on the cluster becoming temporarily unavailable.

Table 1 and Figure 2 present *fractional-availability* metrics for all traces. The table presents the median values for *fractional-availability(x)* metrics for several values of x. These medians were computed over the entire two-week period. Note that, except for `umdres`, more than 90% of the hosts have at least 50% memory idle. Also, there appears to be a relatively sharp change in the slope at 50%. This suggests that the recruitment policy for idle memory should not recruit more than 50% of the memory on a single host. The graphs in Figure 2 present the variation of *fractional-availability(50)* (the fraction of hosts with at least 50% idle memory) for the two-week period. The graphs show that about 70% of the hosts always have at least 50% of their memory available; for `gmu` and `umdcomm`, over 85% of the hosts have at least 50% of their memory idle. These graphs provide support the conclusion that 50% of the physical memory on a host is a good operating point for the recruitment policy for idle memory.

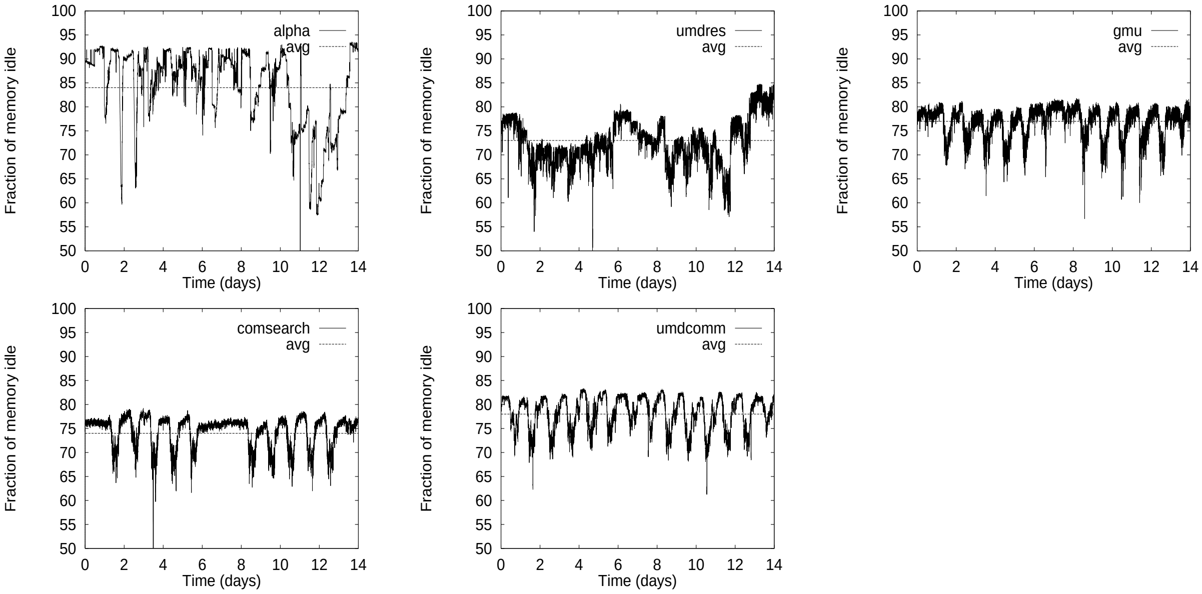

Figure 1: Variation of aggregate-availability over two weeks. Note that, on the average, 70%-85% of the total memory is available and 60% of the total memory is always available. These graphs have a resolution of 90 seconds.

Idle memory $\geq$	`alpha`	`umdres`	`gmu`	`comsearch`	`umdcomm`
33%	94%	90%	99%	96%	99%
40%	94%	86%	98%	95%	98%
50%	92%	84%	97%	92%	97%
60%	88%	77%	93%	86%	93%
66%	85%	72%	88%	82%	89%

Table 1: Median fraction of hosts with $\geq x\%$ memory idle, for x = {33,40,50,60,66}. The median is taken over all 90 second intervals over the two weeks.

2.3 Distribution of memory availability periods

In this section, we present the distributions $D(x)$ of lengths of the intervals for which memory chunks of size x are idle. We have normalized the size of memory chunks by expressing as a fraction of the physical memory available on the host – that is, $D(50)$ is the distribution of idle intervals for memory chunks that are greater or equal to 50% of the physical memory on the respective host. To eliminate very short intervals, we considered only those idle periods that were at least five minutes long.

Figure 3 presents $D(50)$. We note that apart from `alpha`, all other traces, which correspond to clusters of user workstations, have similar distributions. The median length of the idle intervals is 18 to 30 minutes (the median interval length for alpha is 99 minutes).

Table 2 presents the median values of the distributions $D(33), D(40), D(50), D(60)$ and $D(66)$. As expected, for every value of x (memory size in terms of a fraction of host memory), the median value for `alpha` is significantly higher than the values for all the other traces. Another point to note is that even very large memory chunks ($\geq 66.6\%$ of host memory) are likely to survive for more than 15 minutes. This suggests that applications that access their datasets several times in the 15-20 minute range are more likely to benefit from exploiting idle memory.

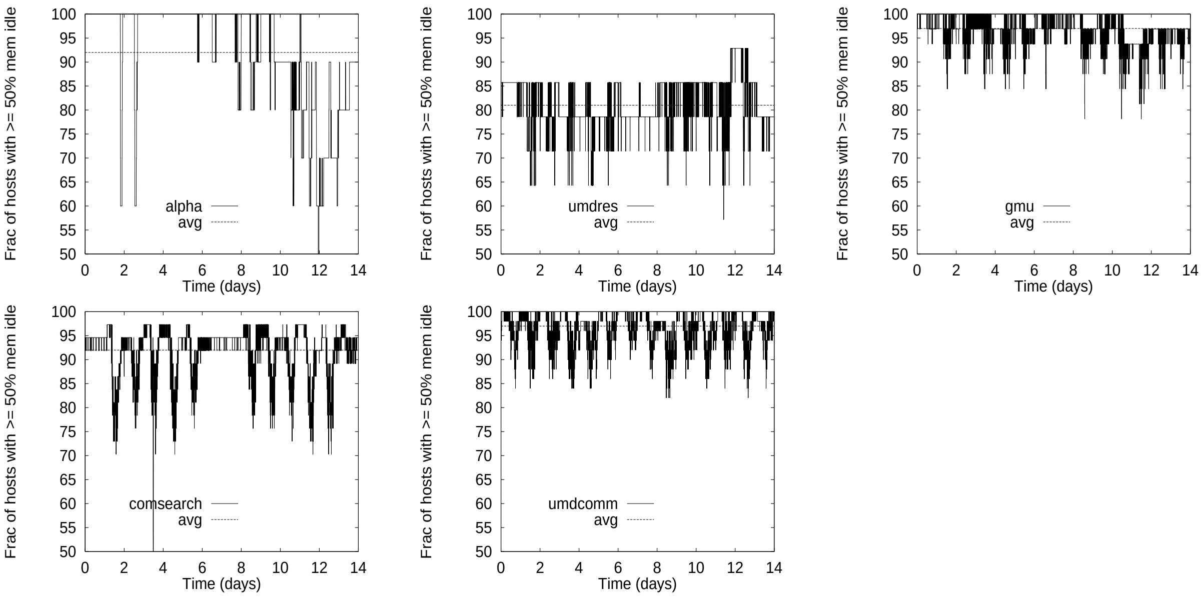

Figure 2: Variation of fractional-availability(50), the fraction of hosts with at least 50% of their memory idle, over two weeks. The graphs have a resolution of 90 seconds.

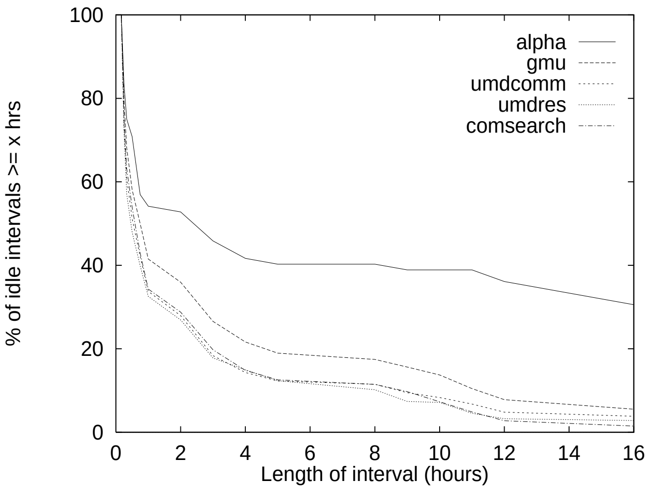

Figure 3: Distribution of idle intervals for memory chunks that are at least 50% the size of the host memory. The median length of the idle intervals for the traces from clusters of user workstations is 18 to 30 minutes; the median interval length for alpha is 99 minutes.

Idle memory $\geq$	alpha	umdres	gmu	comsearch	umdcomm
33%	8370	1620	3690	1800	2340
40%	7340	1530	2430	1440	1620
50%	5940	1080	1800	1350	1260
60%	4770	1080	1260	1350	1080
66%	4320	990	990	1260	1080

Table 2: Median interval length for distributions of memory idle times.

3 Experiments

To estimate the benefits of exploiting idle memory in workstation cluster environments, we simulated the execution of three data-intensive applications on the five workstation clusters described in the previous section. In our experiments, we assume that the application is being executed on a dedicated workstation, and that the memory of idle workstations in a cluster is used to cache copies of disk-resident objects that could be accessed by the application. In other words, the memory of idle workstations is treated as an intermediate layer in the memory hierarchy between the dedicated workstation's local memory and disk.

The goal of our simulations was to assess both the benefits that would accrue to an application from the use of idle memory, as well as the potential costs to the owner of a workstation that is being used in this manner. We first describe the run-time software architecture assumed by our simulations. We then describe the applications that we used as benchmarks, and the metrics used to assess the costs and benefits of exploiting idle memory. Next, we describe our system and workload models used in the simulations.

3.1 Run-time Software for Exploiting Idle Memory

Our simulations assumed the existence of a run-time system for harvesting idle memory (similar in several respects to the well-known Condor system [15] which harvests idle processors). This run-time system, which we refer to as *Dodo*,[1] has four components: (i) resource monitors, (ii) idle memory daemons, (iii) a central memory manager and scheduler, and (iv) a run-time library that is linked in with the application.

Resource monitors always execute on each participating workstation. They notify the central manager when a workstation becomes idle. In our simulations, we assume that a workstation's memory can be recruited for hosting remote objects only if the workstation is idle. We make this assumption for two reasons. First, even though hosting guest memory pages is likely to have a smaller impact on the owner of the workstation than hosting guest processes, many workstation owners will probably be unwilling to permit such usage because of social reasons. Second, as shown by a recent study [23], in some situations, the cpu cycles consumed for handling remote memory requests can degrade the performance of the owner's jobs.

An idle memory daemon is started on a workstation when its memory is recruited for hosting remote objects. This daemon is terminated a fixed period (referred to as the *memory vacate period*) after the workstation becomes busy. It stores remote objects in its address space, and supplies them to applications on request. The maximum amount of memory that can be used by the idle memory daemon for storing remote memory objects is restricted to a fixed fraction of the physical memory available on the workstation. We refer to this fraction as the *memory recruitment fraction*.

When an idle memory daemon is started on a workstation, it preallocates a fixed amount of memory (based on the memory recruitment fraction) in its address space. This memory is used for storing objects sent to the idle memory daemon by applications. The idle memory daemon keeps track of how much space it has available. It does not release memory back to the operating system when remote memory objects are deleted. Instead, this memory is marked as free and reused when a new remote object is created. This is to avoid "walking through" memory since that will have the effect of paging out the memory context previously established by the owner of the workstation.

The central manager keeps track of the idle workstations in the cluster as well as the amount of memory available on each workstation for storing remote objects. Each application is linked to a run-time library that implements the functionality needed by the application in order to create, read, write, and delete remote memory objects. In contrast to global memory systems such as GMS [9] where applications use remote memory pages implicitly, our simulator assumes a system in which applications are written to make use of remote memory *explicitly*. Thus, applications create, read, write, and delete remote memory objects by invoking the corresponding routines in the run-time library. We envision these routines as having a similar API to the routines used for accessing objects on disk (e.g., rm_creat(), rm_read(), rm_write(), and rm_free()).

As mentioned earlier, applications execute on dedicated workstations. The application maps only as many objects into its address space as will fit into the physical memory of the local workstation. When an access to an object results in the application's memory requirements exceeding the physical memory of the workstation, space is created for the new object by replacing an object that is currently mapped into local memory. At this point, the object being replaced can be explicitly stored at a remote idle node by making a library call. Only "clean" objects are stored in remote

[1] Following Condor, many resource harvesting systems are named after birds (eg Finch [1]). All of these systems deal with the CPU and other resources, dodo is just for memory (being an extinct bird, dodo is "just in memory" :))

memory. If an object has been modified while in main memory, the run-time library writes it out to disk before storing it in remote memory. Thus, the application explicitly manages the local memory of the workstation by controlling which objects are mapped into local memory.

The application also explicitly controls which objects are stored in remote memory. For the applications considered in our simulation, we used a "first in" policy for deciding which objects are stored in remote memory. That is, the application attempts to store objects in remote memory as long as there is available space. Once an object is allocated space in remote memory, it will not be replaced from that memory. (However, the object will be dropped from remote memory if the node where it resides becomes busy). This results in a situation in which the objects that are accessed during the earlier stages of an application's execution having a greater chance of being stored in remote memory. The motivation for using a "first in" policy comes from the fact that the applications used in our simulations exhibit scan and triangle-scan I/O access patterns [22]. For such applications, a "first in" policy is a good caching policy.

To allocate remote memory for an object, the application makes a request to the central manager. The central manager allocates space for the object at an idle memory daemon that has enough space for the object. (Note that this requires communication between the central manager and the idle memory daemon.) The remote host at which an object is stored is randomly selected from among the candidate idle nodes. If the memory allocation is successful, the central manager returns a descriptor to the application library, that includes the address of the remote host where space has been allocated. The library then supplies the application with a handle that it uses in all subsequent accesses of the object.

If, however, no space is available, the memory allocation fails. When this happens, the application refrains from making allocation calls for a fixed time period, called the *refraction period*. The motivation behind this is to reduce the cost of allocation attempts in a period when the allocation attempts are not likely to succeed.

Finally, when an attempt to retrieve or write a remote object at a particular node fails (either because the node has crashed, or because it is no longer available for hosting remote objects, or because the node has previously dropped the remote object that was requested), the application library drops all the remote descriptors it possesses that correspond to objects that were stored at that node.

3.2 Benchmarks

We considered three data-intensive applications executing realistic workloads in our study. The I/O characteristics of these applications have previously been studied by Uysal *et al* [22].

OLAP: the workload consisted of five consecutive queries against a simulated database containing records for one year's operation of a large department store (one million customers, 100 thousand products, 10 thousand employees, 200 departments and 100 million transactions). These queries perform complex join, set, and aggregate operations on indexed and non-indexed relations. The data was stored in DB2 (from IBM). The total database size was 5.2 GB (including the indices).

DMINE: this application tries to extract association rules from retail data [17]. The dataset consists of 50 million transactions, with an average transaction size of 10 items and maximal potentially frequent set size of 3 (see [2, 17] for details). The dataset size for this program was 4 GB.

LU: this application computes the dense LU decomposition of an out-of-core matrix [12]. The dataset consisted of an 8192×8192 double precision matrix (total size 536 MB) with a slab size of 64 columns.

We chose these three applications as representatives of three classes of applications. We chose LU to represent the applications whose dataset size straddles the amount of idle memory available in the cluster. We chose DMINE to represent the class of applications whose dataset does not entirely fit into the available idle memory yet a significant portion does. We chose OLAP to represent the applications whose dataset was much larger than the available idle memory.

3.3 Metrics

Benefits: We used two metrics to assess the benefits of exploiting idle memory. The first metric is the percentage reduction in execution time resulting from using idle memory in the workstation cluster as a cache.

The second metric, called *equivalent memory*, is an estimate of the amount of memory that one gets for free by harvesting idle memory. In concrete terms, consider a workstation that has L Mbytes of local memory that can be used by an application executing on it. Now assume that an additional R Mbytes of memory are added to the workstation, but this memory is used to cache disk-resident objects in the same manner in which idle memory on remote workstations is used ,i.e., it is treated as an intermediate layer between the L Mbytes of local memory and disk. Unlike remote idle memory, however, this memory can be accessed at the same cost as local memory. Given this scenario, the *equivalent memory* of a workstation cluster for a particular application is equal to the amount of such memory (R) that has to be added to the dedicated workstation in order to obtain the same improvement in application execution time that is obtained by exploiting idle memory on the workstation cluster.

Note that this metric is application-specific, i.e., the reduction in execution time corresponding to a given amount of equivalent memory will be different for different applications. Figure 4 plots the percentage improvement in execution time for the three benchmark applications as a function of equivalent memory. Note that, in the case of LU, there is no benefit to increasing the equivalent memory beyond 536 MB, since at that point the entire data footprint of the application can fit into memory. In the cases of DMINE and OLAP, the reduction in execution time is a linear function of the equivalent memory; however, DMINE benefits to a much larger extent than OLAP from the increase in equivalent memory. This is because the fraction of total execution time spent on I/O is larger for DMINE than it is for OLAP. Moreover, OLAP has a larger memory footprint (5.2 GB) than DMINE (4 GB) which results in a smaller fraction of I/O requests being eliminated for the same amount of equivalent memory.

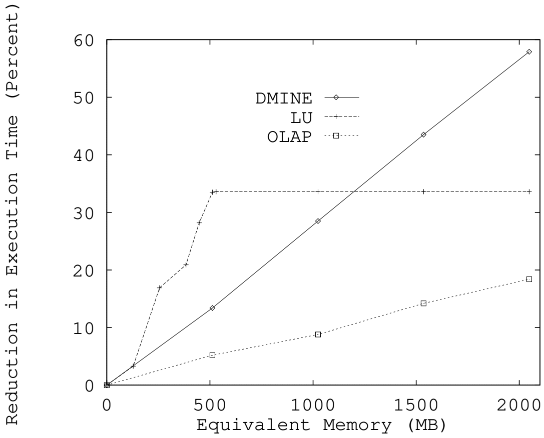

Figure 4: Reduction in Execution Time with increasing Equivalent Memory

Costs: For a workstation owner, the main cost of allowing her workstation's memory to be harvested is the delay experienced when she reclaims the workstation. This delay is a result of the workstation's previous memory contents having been paged out by guest data belonging to the remote application while the workstation was idle.

To assess the cost of exploiting idle memory, we report the average number of reclamations per workstation per day and best-case and worst-case estimates of the average size of the memory context that will need to be reestablished when the user reclaims her workstation. We refer to this metric as the user *memory context size*. The delay experienced by the user is directly proportional to this metric.

As discussed in Section 2, the memory traces used to drive our simulation contain information about the resident set sizes of all the active processes on a particular workstation at a given instant. At the instant when an idle workstation becomes busy, the active process set (according to our definition) will include all periodic daemons, the X server, and any process that has executed on the CPU since the workstation became idle. Assume that the sum of the resident set sizes of these active processes is A, and that the amount of memory occupied by guest data at the time the workstation becomes busy is G. (Since the idle memory daemon never "frees" any memory, this is the maximum amount of memory that has been used for guest data since the workstation became idle.)

In the best-case scenario, when memory is allocated to the idle memory daemon for hosting remote objects, pages belonging to the active processes will not be replaced from local memory since they have been accessed recently and most operating systems use an approximate LRU policy for page replacement. Pages belonging to active processes will only be replaced if the total demand for memory during the idle interval ($G + A$) exceeds the physical memory (say M) on the workstation. In the worst case scenario, however, the G pages acquired by the idle memory daemon will replace active memory pages. This give us the following best-case and worst case estimates for the size of the user memory context (U) that will need to be reestablished:

$$U(\text{best}) = \begin{cases} 0, & \text{when } A + G <= M \\ A + G - M, & \text{when } A + G > M \end{cases}$$

$$U(\text{worst}) = \min(G, A)$$

We note that the worst case estimate for the user memory context size is very conservative for several reasons. First, the resident set sizes (RSS) recorded in our traces for each active process do not provide information about the portion of the RSS that is occupied by shared libraries. Thus the shared libraries are counted for each active process resulting in an overestimate of the actual resident memory. Second, we believe that since most operating systems use a variant of LRU replacement, the best case estimate is more accurate than the worst-case estimate. Finally, we note that the workstation's memory context will be reestablished over a period of time after the owner returns to her workstation, and thus the actual delay experienced by the user will be smaller than the worst-case estimate suggests.

3.4 System and Workload Model

In our simulations, we assumed a single application was using the idle memory on the workstation cluster. We assumed that this application was being executed on a dedicated workstation with four 250 MHz processors and 256 Mbytes of main memory. Such configurations are becoming increasingly common in desktop machines. The workstations in the cluster were assumed to be interconnected with a switched network that has a bandwidth of 15 MB/sec (the bandwidth that can be expected on an ATM-based LAN). Host-to-host network latency was assumed to be 60 microseconds. The simulator assesses a cost for each communication between the application, idle memory daemons, and the central manager for allocating, reading, writing, or deleting remote memory objects.

We assume that the workstation has a large local disk on which the data set of the application is stored. The disk access times used in our simulation were based on end-to-end disk bandwidth measurements of an IBM Starfire 7200 disk. The measured bandwidth depends upon the size of the data request as well as the nature of the request (i.e., read or write); it ranges from 5.6 MB/sec for 4 KB reads to 6.9 MB/sec for 1 MB reads, and from 6.7 MB/sec for 4 KB writes to 6.8 MB/sec for 1 MB writes. Disk requests were assumed to be serviced in a FCFS fashion. Note that this is a reasonable assumption for the applications under consideration because of two reasons: (i) the applications make large I/O requests (the average request sizes for LU, DMINE, and OLAP are 332 KB, 128 KB, and 32 KB respectively) (ii) the average I/O load imposed on the disk was small resulting in little or no disk queuing. Thus, using more sophisticated disk scheduling policies would yield little benefit.

Three sets of traces were used to drive our simulation: (i) workstation availability traces for each workstation in the cluster under consideration (ii) memory usage traces for each workstation in the cluster under consideration (iii) I/O traces for the three benchmark applications. The workstation availability and memory usage traces for each workstation were obtained as described in Section 2.1.

The I/O traces used to drive our simulation were obtained by running the benchmark applications on eight processors of an IBM SP-2. These traces were acquired by Uysal et al [22] using the AIX `trace` utility. The traces contain information about the compute time between successive I/O requests, as well as the operation performed during (read, write, open, seek, close) an I/O request, the file that is accessed, the size of the request, and the offset of the request within the file. To account for the increase in processor speeds in comparison to the speed of the 66 MHz processors on an IBM SP-2, we divided the computation time between I/O requests by a factor of four.

To emulate the execution of the eight process application on a four processor system, the execution of a process is simulated until it reaches a synchronization point. At that point, the simulator switches to executing another process. Thus all eight processes execute concurrently, with any four executing at a given time on the four processors of the workstation.

We examined two scenarios in our simulation. The first scenario consisted of repeated execution of the LU benchmark without any gaps, i.e., as soon as an instance of LU completes, another instance is started. Further, as would be the case for an out-of-core computation, all remote memory objects created during the execution of the benchmark are deleted at its completion. Thus any performance benefit that LU obtains from remote memory is due to accessing the same object multiple times during a single run.

Benchmark	LU			DMINE			OLAP		
	Equivalent Memory (MB)		Execution Time Reduction	Equivalent Memory (MB)		Execution Time Reduction	Equivalent Memory (MB)		Execution Time Reduction
Cluster	Avg	Median		Avg	Median		Avg	Median	
alpha	211.3	226.2	11.5%	275.9	211.4	7.1%	177.0	134.4	1.8%
umdres	232.2	231.4	14.0%	265.3	269.0	7.0%	167.5	173.9	1.7%
gmu	349.4	360.5	19.3%	391.6	393.6	10.2%	270.7	274.5	2.8%
comsearch	377.4	378.8	20.5%	954.1	1047.8	26.5%	747.8	849.4	6.9%
umdcomm	378.4	378.4	20.6%	753.9	757.9	20.5%	519.9	517.1	5.3%

Table 3: Performance Improvements for the three benchmarks on the five clusters.

The second scenario examined consisted of repeated execution of a benchmark with an inter-execution delay. This scenario was used for the OLAP and DMINE benchmarks. In contrast to the first scenario, under this scenario, remote memory objects are not deleted at the end of an application's execution. Instead, they are retained in remote memory (and will stay there unless dropped as a result of the node becoming busy). There are two motivations for studying this scenario: (i) this scenario models the manner in which remote memory would be used by a database supporting decision support queries or data-mining algorithms (ii) it gives us an idea of how long we can expect an object to be retained in remote memory. In our simulations, we varied the inter-execution delay from half an hour to one day.

4 Results

We ran our experiments for one week of simulated time (Sept 14-20, 1997). This allowed us to understand the effect of time-of-day and day-of-the-week variations in workstation usage. Unless otherwise stated the results presented in this section assume the following baseline parameter settings: *memory recruitment fraction* = 0.5, *memory vacate time* =2 minutes, and *retraction period* = 1 minute. In addition, the length of time a workstation has to be idle before its memory can be harvested, i.e., the workstation *recruitment threshold* was assumed to be 5 minutes. Finally, in the case of the DMINE and OLAP benchmarks, the *inter-execution delay* was assumed to be 1 hour.

4.1 Performance Benefits

Table 3 presents the performance improvements that are obtained for the three benchmarks on the five clusters for the baseline parameter settings. We report the average and median equivalent memory for the simulated week as well as the average improvement in application execution time.

The improvement in performance obtained on a cluster depends on both the amount of memory available on the cluster as well as the characteristics of the application. LU, which has the smallest memory footprint among the three benchmarks, has significant performance benefits on the gmu, comsearch, and umdcomm clusters. The comsearch and umdcomm clusters have the largest amount of physical memory, and our analysis indicates that a large fraction of it is idle. gmu has less physical memory than comsearch or umdcomm, but the amount of idle memory is enough to hold a large fraction of LU's memory footprint. umdres has a relatively small amount of physical memory and this results in LU having smaller performance gains. While alpha has a large amount of physical memory (and a large amount of this memory is idle as indicated in Section 2), we found that the workstations are busy or unavailable for a large fraction of time. We note that the primary use of alpha is for running compute-intensive jobs. As our policy is to not harvest idle memory on busy hosts, this memory cannot be exploited for a large fraction of the time. This reduces the performance gains obtained for LU.

We note that for LU the equivalent memory metric appears to reach a limit around 379 MB. This conclusion is also supported by Figure 5 which plots the temporal variation in equivalent memory for three clusters over the course of the week. The equivalent memory for LU on comsearch and umdcomm has very little variation during the week. This indicates that there is adequate idle memory on these clusters to store the entire memory footprint for LU, and the limit on the equivalent memory reflects the network communication overheads incurred for allocating, reading, and writing remote memory.

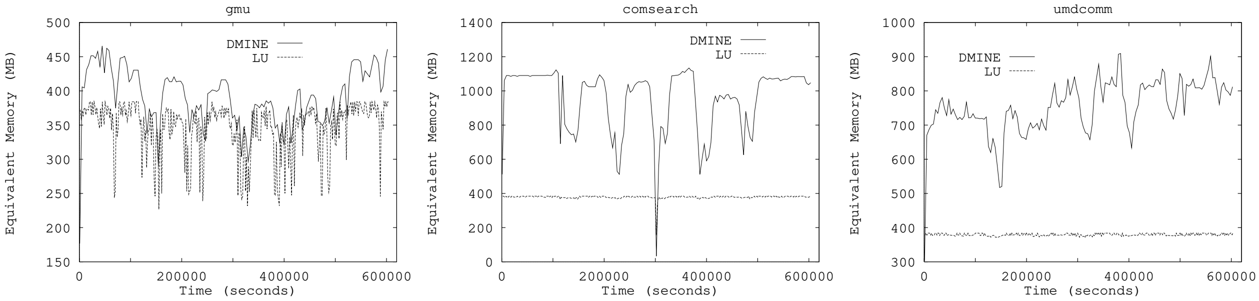

Figure 5: Variation of Equivalent Memory for three clusters during the week.

In the case of DMINE, we find that performance benefits obtained on each cluster reflect the amount of idle memory available. Unlike LU, DMINE's memory footprint does not fit in the idle memory available on any of the clusters. Thus, in contrast to LU, the equivalent memory for DMINE on a cluster is limited by the amount of idle memory in addition to communication overheads. The performance gains for DMINE are largest on comsearch which has an equivalent memory of more than 1 GB. Performance gains are smallest on umdres and alpha reflecting the lower amounts of idle memory on these clusters. Note from Figure 4 that in order to obtain performance improvements greater than 10% for DMINE, a cluster needs an equivalent memory greater than 512 MB. The modest performance improvements obtained for DMINE on gmu and umdres is consistent with the relatively small ($<$ 1GB) amount of physical memory available on these clusters, and the fact that at most 50% of this memory is recruited in these experiments.

The trends observed for OLAP are similar to those for DMINE, with an important difference – the overall performance improvement obtained is much lower than that obtained for DMINE. This can be explained by two reasons. First, as shown in Figure 4, the performance improvement for OLAP for a given amount of equivalent memory is much smaller than the improvement obtained for DMINE. Second, the average size of an I/O request for DMINE is 128 KB, whereas the average size for OLAP is 32 KB. This results in OLAP making many more requests over the network for allocating and reading remote memory objects than DMINE does. Thus, the larger communication overheads experienced by OLAP in comparison to DMINE result in a smaller equivalent memory for the same cluster, as shown in Table 3.

Figure 5 plots the temporal variation in the equivalent memory of gmu, comsearch, and umdcomm for each benchmark over the course of a week. The plots illustrate the obvious time of day effects, with peaks in the equivalent memory corresponding to weekends and nighttime hours. These trends are particularly pronounced for comsearch reflecting the difference between academic and office environments. As pointed out earlier, on both umdres and comsearch there is *always* enough idle memory that there is no variation in equivalent memory for LU. On the other hand, in the case of gmu, the equivalent memory for LU varies with the time of day since this cluster does not have as much physical memory as the other clusters. Finally, we note that the sharp dip in equivalent memory for DMINE on comsearch was caused by a failure event that resulted in all the workstations on the cluster becoming temporarily unavailable (see Figures 1 and 2).

4.1.1 Normalized Equivalent Memory

These results above clearly show that the equivalent memory for an application depends on both its I/O characteristics as well as the amount of idle memory on the cluster. However, we would like to be able to formulate a rule-of-thumb that helps us determine how much memory we can harvest for free from a cluster, independent of the size of the cluster, and the amount of memory available per host on the cluster.

With this goal, we analyzed the equivalent memory results reported above for the DMINE benchmark in more detail. We selected DMINE because of two reasons. First, its memory footprint exceeds the physical memory available on these clusters; consequently DMINE is likely to utilize all idle memory that is available during its execution. Second, in some respects, DMINE is an ideal application for exploiting idle memory since it is very I/O intensive, and does I/O using large requests. Thus, the equivalent metric reported for DMINE in Table 3 is likely to be close to the highest equivalent memory metric one can obtain given the network bandwidth of the cluster.

Table 4 shows the *normalized equivalent memory* for DMINE on the five clusters for a memory recruitment fraction

Cluster	Equivalent Memory (MB)	Equivalent Memory per host (MB)	Normalized Equivalent Memory per host
alpha	276	27.6	0.10
umdres	265	18.9	0.30
gmu	392	12.3	0.31
comsearch	954	25.8	0.30
umdcomm	754	15.1	0.28

Table 4: Normalized equivalent memory for DMINE on each cluster using a memory recruitment fraction of 0.5.

of 0.5. The second column in Table 4 reports the equivalent memory *per host* in the workstation cluster. This column shows that we can expect to get for free between 12 and 28 MB of memory per host in a typical cluster today for an ideal application like DMINE if we restrict the memory recruitment fraction to 0.5. The third column in Table 4 reports the *normalized equivalent memory per host*. To obtain this metric, we divided the equivalent memory per host by the average amount of memory per host for the cluster. This column shows that for all clusters that consist of user workstations, using a memory recruitment fraction of 0.5, we can expect to harvest for free around 30% of the memory per host in the cluster.

4.2 User Costs

In Table 5, we report (i) the average number of workstation reclamations per day, and (ii) the best-case and worst-case estimates of the average user memory context size observed in our experiments. We also report the median for each metric.

Benchmark	LU			DMINE			OLAP		
	Num. Reclaims	Memory Context Size (MB)		Num. Reclaims	Memory Context Size (MB)		Num. Reclaims	Memory Context Size (MB)	
Cluster	Per Day	Best	Worst	Per Day	Best	Worst	Per Day	Best	Worst
alpha	3.2 (3.7)	0 (0)	4.4 (2.6)	2.0 (2.3)	0 (0)	3.8 (1.5)	3.2 (3.6)	0 (0)	5.1 (4.4)
umdres	7.2 (7)	0.8 (0)	13.9 (14.2)	4.8 (5.7)	1.1 (0)	14.4 (15.3)	5.5 (6.3)	1.3 (0)	14.6 (15.1)
gmu	4.5 (4.6)	0 (0)	7.3 (6.6)	2.9 (2.9)	0 (0)	7.3 (6.7)	3.4 (3.6)	0 (0)	7.3 (7.0)
comsearch	6.9 (7.3)	0 (0)	10.1 (11.4)	3.8 (3.7)	0.6 (0)	16.1 (14.9)	5 (5)	1.5 (0)	17.3 (15.1)
umdcomm	11.1 (5.1)	0 (0)	7.3 (7.0)	8 (4.1)	0.1 (0)	8.7 (7.9)	9.8 (4.7)	0.1 (0)	8.8 (7.4)

Table 5: Impact of exploiting idle memory on workstation owners. The number in parentheses is the median.

Our results show that in most cases a user will not experience any delays when she reclaims her workstation. The median best-case estimate of the memory context size is 0 for all three applications on all five clusters, while the *largest* average best-case estimate of the user memory context size is 1.5 MB. The average worst-case estimate of the user memory context size ranges from 3.8 to 17.3 MB. However, as discussed earlier, we believe the best case estimate is a more accurate measure for the user memory context size than the worst-case estimate. Our results thus support the conclusion that harvesting at most 50% of an idle node's physical memory has a very small impact on the owner of the workstation.

Our results also show that the average number of reclamations per day ranges from 3 to 11. We note that the number of reclamations experienced for the LU workload is larger than it is for OLAP and DMINE. This is explained by the fact that in our simulations, LU is being executed repeatedly without any gaps, whereas OLAP and DMINE are being executed with an inter-execution gap of 1 hour. The number of reclamations is larger for OLAP than it is for DMINE because of OLAP's longer execution time.

Recruitment Fraction	LU			DMINE		
	gmu	comsearch	umdcomm	gmu	comsearch	umdcomm
0.33	0	0	0	0	0	0
0.5	0	0	0	0	0.64	0.1
0.66	0	0.03	0	0.17	2.4	0.65

Table 6: Impact of changing the memory recruitment fraction on the best-case user memory context size.

4.3 Sensitivity Analysis

We now examine the effect of changing the various input parameters that control the simulation. The sensitivity analysis experiments described below simulate the execution of DMINE and LU on the clusters gmu, comsearch, and umdcomm. These clusters were selected because they represent different size clusters (in terms of both number of workstations and amount of physical memory) and different environments (office and academic).

4.3.1 Impact of changing the memory recruitment fraction

In Figure 6, we plot the equivalent memory for LU and DMINE as a function of the memory recruitment fraction (rf). We note that in the case of DMINE, none of the clusters has enough idle memory to accommodate its entire memory footprint. Thus, increasing the memory recruitment fraction results in a proportional increase in the equivalent memory. In the case of LU, however, increasing rf is only beneficial on gmu. comsearch and umdcomm have enough idle memory that even for $rf = 0.33$, the entire memory footprint of LU can be stored in idle memory. On gmu, the benefits of increasing rf from 0.33 to 0.5 are more pronounced than the benefits of increasing the rf from 0.5 to 0.66. This reflects the fact that a large fraction of the memory footprint of LU can be accommodated in memory for $rf = 0.5$.

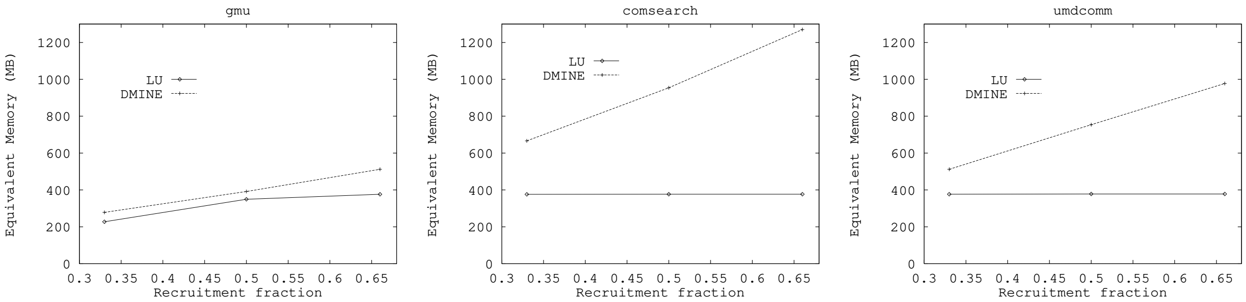

Figure 6: Impact of changing the recruitment fraction on the equivalent memory of a cluster.

Table 6 shows the impact of changing the recruitment fraction on the best-case memory context size. In the case of DMINE, our experiments show that increasing rf from 0.5 to 0.66 results in the best-case average memory context size going up from 0 to a small positive number (< 1 MB for umdcomm and gmu, and 2.5 MB for comsearch). This implies that increasing rf to 0.66 has a definite cost in terms of the delay experienced by the user when the workstation is reclaimed. This result is in agreement with the memory availability results in Section 2 which indicate that the number of nodes per cluster that always have 50% of their memory free is much larger than the number of nodes per cluster that always have 66% of their memory. Although not shown here, our results also showed that changing rf from 0.33 to 0.66 had a negligible impact on the worst-case memory context size.

From these results, we can conclude that while increasing rf beyond 0.5 is beneficial to applications whose memory footprint exceeds the amount of idle memory available on a cluster, this is at the expense of a greater likelihood of a user-visible delay when the workstation is reclaimed. A memory recruitment fraction of 0.5 appears to balance the goals of providing performance benefits to data-intensive applications and minimizing the costs to the owner of the workstation.

4.3.2 Impact of changes in disk and network bandwidth

Our next set of results considers the impact of changing network and disk bandwidths on the equivalent memory of a cluster. In our experiments, we simulated the execution of LU and DMINE on the three clusters assuming network bandwidths of 32 MB/s (the IBM SP-2 interconnect) and 100 MB/s (e.g. Myrinet and Gigabit Ethernet). Network latency was assumed to be 60 microseconds for all cases. As expected, Figure 7 shows that increasing the network bandwidth results in an increase in the equivalent memory of a cluster.

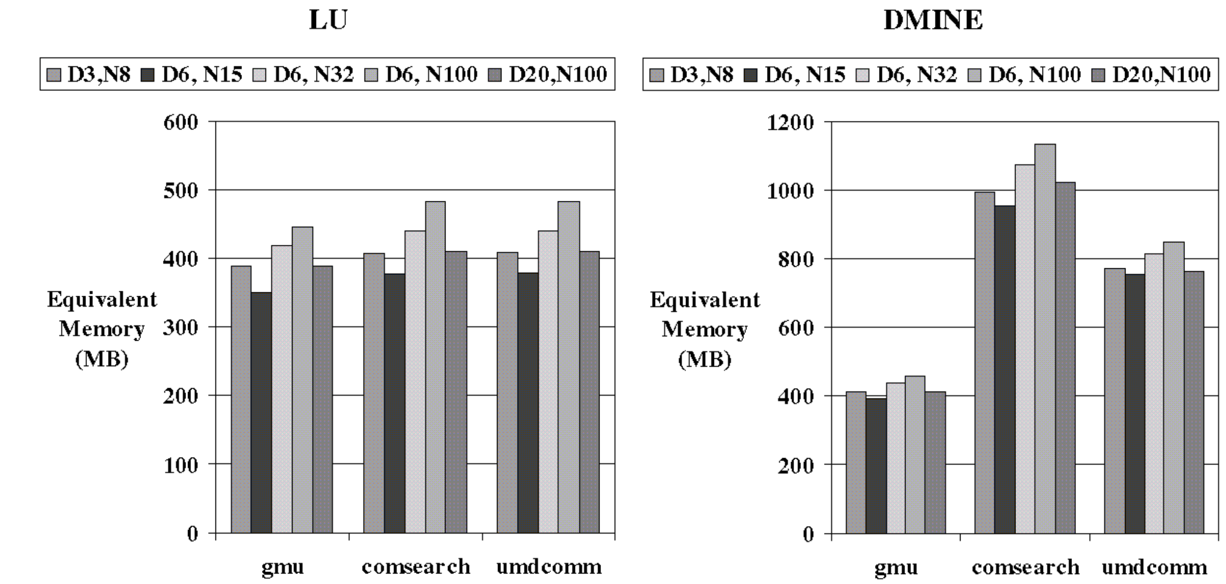

Figure 7: Impact of varying the network and disk bandwidth on the equivalent memory. In the legend, the notation Dx,Ny denotes a disk bandwidth of x MB/sec and a network bandwidth of y MB/sec. The bar corresponding to D6,N15 shows the equivalent memory for the baseline disk and network bandwidths used in our simulations.

We also examined the impact of increasing the disk bandwidth to 20 MB/sec (bandwidth of the latest Seagate disks) and decreasing the bandwidth to 3 MB/sec (the bandwidth of some current generation IDE disks). The equivalent memory for each cluster for the case when disk bandwidth is 20 MB/sec and network bandwidth is 100 MB/sec is slightly higher than the equivalent memory for our baseline experiments reported in Section 4.1 that assume a network bandwidth of 15 MB/sec and disk bandwidth of the IBM Starfire 7200 disks (roughly 6 MB/sec). Interestingly, the equivalent memory assuming disk bandwidth of 3 MB/sec and network bandwidth of 8 MB/sec (achievable on switched Fast Ethernets) is comparable to the equivalent memory achieved when the disk bandwidth is 20 MB/sec and network bandwidth is 100 MB/sec.

From this analysis, we can conclude that the equivalent memory metrics presented in Section 4.1 are robust. Given current disk and network technology trends, we expect that the benefits achieved by exploiting idle memory will continue to be available over the next few years.

4.3.3 Impact of changes in memory vacate time

Our next set of results considers the impact of changing the memory vacate time on the performance of LU and DMINE. Figure 8 plots the change in equivalent memory as the memory vacate time is varied from 0 seconds (immediate vacate) to infinity (never vacate). On all three clusters, there is negligible effect on varying the vacate time on the equivalent machine for LU. For DMINE, there is an increase in the equivalent memory as the memory vacate time is increased. However, in all cases, the reduction in execution time achieved by going from the immediate vacate to the never vacate policy is less than 10%. Given the fact that the immediate vacate policy is the most attractive from the point of view of the workstation's owner, we conclude that a memory vacate time of 0 seconds is the best choice.

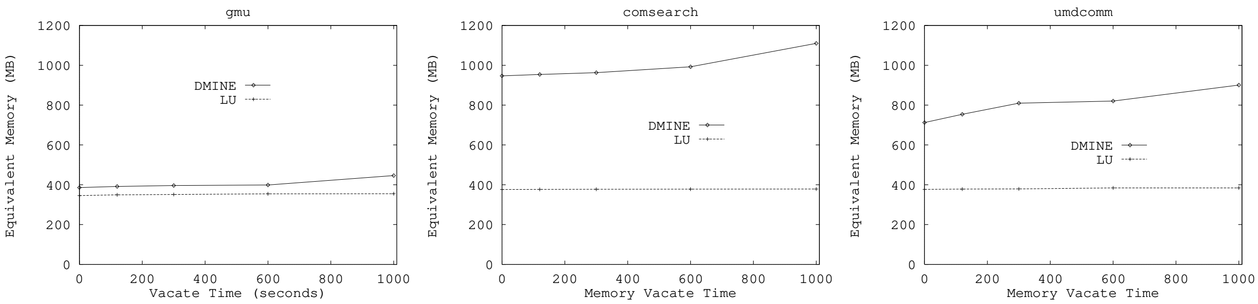

Figure 8: Impact of change in memory vacate time on the equivalent memory of a cluster.

4.3.4 Impact of changing inter-execution delay

As described earlier, the scenario considered in our simulations for the the DMINE and OLAP benchmarks consisted of repeated executions of the application with a delay between consecutive executions. We varied the length of this delay in our simulations to examine its impact on the performance of the benchmarks.

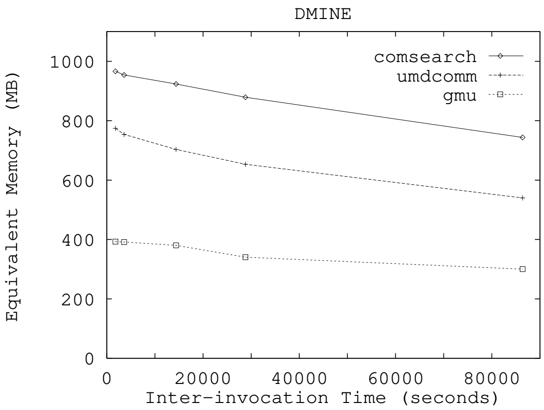

Figure 9: Variation in equivalent memory for different inter-invocation intervals of DMINE.

In Figure 9, we plot the equivalent memory for comsearch, gmu, and umdcomm as a function of the inter-invocation delay. As expected, increasing the inter-execution delay from 30 minutes to 1 day results in a reduction in the equivalent memory of around 25%. However, it is worth noting that on all three clusters, an application with a large memory footprint such as DMINE can expect to get some performance benefit from using remote idle memory even if it is executed a day later. This indicates that in all three clusters, there are some nodes that are always idle. Objects stored in the memory of these nodes do not get dropped resulting in the performance benefits observed in our experiments.

5 Other Related Work

Franklin et al [11] describe a unified memory management scheme for the servers and all the clients in a client-server database system. Their goal was to avoid replication of pages between the buffer pools of all the clients as well as the buffer pools of the servers. Explicit memory servers have been proposed by Narten&Yavatkar [18] and Iftode et al [13]. They describe a memory server similar in spirit to the Dodo central manager. It keeps track of the idle memory available and ships memory objects to the corresponding machines as needed. Iftode et al propose extending the memory hierarchy of multicomputers by introducing a remote memory server layer. Felten and Zahorjan [10] examined the idea of using remote client memory instead of disk for virtual memory paging. Schilit and Duchamp [21] investigated the use of remote memory paging for diskless portable machines. Comer and Griffioen [7] proposed a communication protocol for remote paging.

6 Conclusions

There are three primary conclusions of our study. First, there is significant benefit to harvesting idle memory on workstation clusters for data-intensive applications. As a simple rule of thumb, one can get up to 30% of the memory of a typical cluster of user workstations for free without any adverse impact on the owners of workstations and without making any modifications to the operating system. Our study indicates that for applications that can make good use of this memory, utilizing it as an intermediate cache can result in up to 26% reduction in execution time. Second, restricting the maximum percentage of the workstation's memory that can be recruited to 50% ensures that users experience virtually no delays when reclaiming their workstations. Third, these results are robust over a wide variation in disk and network bandwidth. As long as network and disk technology follow current trends, similar benefits can be expected.

The other conclusions of our study are:

- On the average, 70-85% of total memory in a workstation cluster is idle; 60% of the total memory is always idle.
- At all times, 70-85% of the workstations have more than 50% of their memory idle.
- Memory that has been idle for 5 minutes can be expected to remain idle for at least 15 more minutes.
- The performance impact of vacating memory as soon as a workstation is reclaimed is small. This is significant as an immediate-vacate policy is sociologically preferable.

Acknowledgments

We would like to thank Rudy Zulkarnain of Comsearch Inc. for providing us with the `comsearch` traces. We would like to thank Mustafa Uysal for the application traces used in this study.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. In *Proceedings of SIGMETRICS'97*, 1997.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. of 20th Int'l Conf. on Very Large Databases (VLDB)*, Santiago, Chile, Sept. 1994.

[3] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 267–78, May 1995.

[4] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 198–212, 1991.

[5] N. Carriero, D. Gelernter, M. Jourdenais, and D. Kaminsky. Piranha Scheduling: Strategies and Their Implementation. *International Journal of Parallel Programming*, 23(1):5–33, Feb 1995.

[6] A. Chowdhury, L. Nicklas, S. Setia, and E. Whie. Supporting dynamic space-sharing on clusters of non-dedicated workstations. In *Proceedings of the 17th International conference on distributed computing*, 1997.

[7] D. Comer and J. Griffioen. Efficient order dependent communication in distributed virtual memory environment. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 249–62, 1992.

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 267–80, Nov 1994.

[9] M. Feely, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the fifteenth ACM Symposium on Operating System Principles*, pages 201–12, Dec 1995.

[10] E. Felten and J. Zahorjan. Issues in implementation of a remote memory paging system. Technical Report 91-03-09, Department of Computer Science, University of Washington, 1991.

[11] M. Franklin, M. Carey, and M. Livny. Global Memory Management in Client-Server DBMS Architectures. In *Proceedings of the eighteenth International Conference on Very Large Data Bases*, pages 596–609, Aug 1992.

[12] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Comput.*, 15(5), Sept. 1994.

[13] L. Iftode, K. Li, and K. Petersen. Memory Servers for Multicomputers. In *COMPCON Spring'93 Digest of Papers*, pages 538–47, Feb 1993.

[14] A. Leff, P. Yu, and J. Wolf. Policies for efficient memory utilization in a remote caching architecture. In *Proceedings of First International Conf. on Parallel and distributed Information Systems*, 1991.

[15] M. Litzkow and M. Livny. Experiences with the Condor Distributed Batch System. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pages 97–101, Oct 1990.

[16] J. Moreira, V. Naik, and R. Konuru. A Programming Environment for Dynamic Resource Allocation and Data Distribution. Technical Report RC 20239, IBM Research, May 1996.

[17] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[18] T. Narten and R. Yavatkar. Remote Memory as a Resource in Distributed Systems. In *Proceedings of the third Workshop on Workstation Operating Systems*, pages 132–6, April 1992.

[19] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 259–78, April 1995.

[20] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of The Second Symposium on Operating Systems Design and Implementation*, 1996.

[21] B. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. Technical Report CUCS-004-91, Department of Computer Science, Columbia University, 1991.

[22] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O systems for parallel machines: An application-driven study. Technical Report CS-TR-3802, Department of Computer Science, University of Maryland, 1997.

[23] G. Voelker, H. Jamrozik, M. Vernon, H. Levy, and E. Lazowska. Managing Server Load in Global Memory Systems. In *Proceedings of the 1997 ACM Sigmetrics Conference*, pages 127–138, June 1997.