Dodo: A User-level System for Exploiting Idle Memory in Workstation Clusters

Samir Koussih
Dept. of Computer Science
George Mason University
Fairfax, VA 22030

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

Technical Report TRCS98-35
Dept of Computer Science
University of California, Santa Barbara

December 1998

Abstract

In this paper, we present the design and implementation of *Dodo*, an efficient user-level system for harvesting idle memory in off-the-shelf clusters of workstations. *Dodo* enables data-intensive applications to use remote memory in a cluster as an intermediate cache between local memory and disk. It requires no modifications to the operating system and/or processor firmware and is hence portable to multiple platforms. Further, the memory recruitment policy used by *Dodo* is designed to minimize any delays experienced by the owner of desktop machines whose memory is harvested by *Dodo*.

Our implementation of *Dodo* is operational and currently runs on Linux 2.0.35. For communication, *Dodo* can use either UDP/IP or *U-Net*, the low-latency user-level network architecture developed by von Eicken et al [4]. We evaluated the performance improvements that can be achieved by using *Dodo* for two real applications and three synthetic benchmarks. Our results show that speedups obtained for an application are highly dependent on its I/O access pattern and data set sizes. Significant speedups (between 2 and 3) were obtained for applications whose working sets are larger than the local memory on a workstation but smaller than aggregate memory available on the cluster and for applications that can benefit from the zero-seek nature of remote memory.

1 Introduction

In recent years, there has been an explosion in the number of data-intensive applications. Applications with dataset sizes ranging from several hundred megabytes to several gigabytes can be found in many different domains, e.g., scientific computing, data mining, and multi-media systems. Since their memory footprint is often much larger than the physical memory on most machines, these applications spend a significant fraction of their time waiting for disk I/O.

Given the dramatic fall in the price of DRAM chips in recent years, an obvious way of improving the performance of such applications is to install more memory on the machines used to run them. However, installing large amounts of memory on a *single* machine is still expensive for two reasons. First, memory is cheap only in commodity sizes (32/64/128 MB); there is a large premium for larger sizes. For example, a 512 MB PC100 SDRAM DIMM costs $3900 [12] whereas four 128 MB PC100 SDRAM DIMMs cost $600 [15]. Second, commodity-priced machines have a limit on the amount of memory; server-class machines that can have larger memories come at a substantial premium.

The availability of commodity-priced high speed LANs introduces another alternative: a cluster of commodity-priced workstations. Even though the memory on each such machine is only 64-256 MB, the aggregate memory in the cluster is often several gigabytes. Furthermore, a large fraction of this memory is often idle [2]. Using this memory as an intermediate cache between local memory and disk offers the potential of improving the performance of data-intensive applications, without the expense associated with large-memory machines. This is especially attractive for programs with transient peaks in their memory requirements or for data-intensive programs that are run infrequently.

Several research projects have proposed using memory of idle workstations for hosting guest data [5, 6, 7, 8, 10, 14, 16, 21]. Iftode et al [10] propose extending the memory hierarchy of multicomputers by introducing a remote memory server layer; Felten and Zahorjan [8] examined the idea of using remote client memory instead of disk for virtual memory paging; Schilit and Duchamp [17] investigated the use of remote memory paging for diskless portable machines; Dahlin et al [6] and Sarkar et al [16] propose schemes to use idle memory to increase the effective file cache size; Feeley et al [7] describe a low-level global memory management system that uses idle memory to back up both file pages and virtual memory page.

These efforts have focused on developing efficient kernel-level mechanisms for harvesting idle memory. In this paper, we present a user-level mechanism, called *Dodo*,[1] that requires no modifications to operating system kernel or processor firmware. The history of resource-harvesting systems such as Condor has shown that portability (with respect to both processor architecture and the operating system) is an important requirement for their success and longevity. User-level systems like Condor [11] have flourished for over a decade through several generations of processors and operating systems. Accordingly, portability was an important goal for the design and implementation of Dodo. While the performance improvements obtained via Dodo are potentially smaller than those obtained by customizing the operating system kernel, they are available to much larger community.

The design of *Dodo* is based on that of Condor [11], one of the most successful systems for harvesting idle resources. In contrast to global memory systems such as GMS [7] where applications use remote memory pages implicitly, *Dodo* requires applications to make explicit use of remote memory. Each application is linked to a library that implements the functionality needed by the application in order to create, read, write, and delete remote memory regions.

An explicit interface has the disadvantage that it *requires* the programmer to keep track of memory usage and to coordinate all data transfer to and from the remote memory cache. To lighten this burden, we have developed a coarse-grain memory-management library that can be layered on top of *Dodo*. This library is linked in with the application and tracks coarse-grain memory access-patterns and implements multiple memory-replacement policies. Note that in *Dodo*, remote memory is used for read-only caching. Writes to remote memory are propagated to disk in parallel to being sent to the remote host.

We have implemented *Dodo* on a Beowulf class Linux cluster. We evaluated the performance improvements that can be achieved by using *Dodo* for two real applications and three synthetic benchmarks. Our results show that speedups obtained for an application are highly dependent on its I/O access pattern and data set sizes. Significant speedups (between 2 and 3) were obtained for two kinds of applications: (i) applications whose working sets are larger than the local memory on a workstation but smaller than aggregate memory available on the cluster (ii) applications that can benefit from the zero-seek nature of remote memory.

We first present a brief summary of the study of idle memory in workstation clusters that motivated our work. We then present the design and implementation of the different components of *Dodo*. Finally, we present performance results for a suite of real and synthetic data-intensive applications.

2 Availability of Idle Memory in Workstation Clusters

The design and implementation of *Dodo* was motivated by the results of our study [2] that monitored memory availability on two production clusters for several weeks. For this study, we captured detailed traces of memory and processor usage on a 29 workstation Solaris cluster at the University of California at Santa

[1]Following *Condor*, many resource harvesting systems are named after birds (e.g., Finch [1]). These systems harvest processor cycles and other resources; *Dodo* is just for memory (and being an extinct bird, *Dodo* is “just a memory” :))

Barbara (referred to as `clusterA`) and a 23 workstation Solaris cluster at George Mason University (referred to as `clusterB`) . These workstations are used by faculty and graduate students for personal purposes as well as for running large compute-intensive jobs.

For each workstation in these clusters, we used a suite of tools (including `top`, `lsof`, and the `memtool` kernel package for Solaris) to monitor several kinds of information that allowed us to determine how memory was being used: (i) by the operating system kernel (ii) for caching file pages (iii) for process virtual memory, and (iv) for the free list. We further processed these traces to come up with an estimate of the "idle" memory on each workstation, i.e., the amount of memory that can be recruited for hosting guest data while minimizing any delays experienced by the workstation owner.

We summarize below the main results of our study, while referring the reader to [2] for more details on our methodology and results:

- On average, a large fraction of the total memory installed on a cluster is available: 60-68% if we consider all hosts, and about 53% if we consider only idle hosts. (see Figure 1). We consider a host to be idle if there has been no keyboard or mouse activity for the last five minutes and the average cpu load over the same period is less than 0.3.
- On the average, a substantial fraction of the total memory of desktop machines is not in active use. This fraction grows as the amount of memory on a machine grows – from about 12-14 MB for a 32 MB machine to about 180-192 MB on a 256 MB machine (see Table 1 for details).

While the first result above indicates that there is a significant amount of idle memory available on a cluster, the second result suggests a strategy for exploiting this idle memory without causing any noticeable delays to the owner of the workstation. We discuss this point in more detail in Section 3.

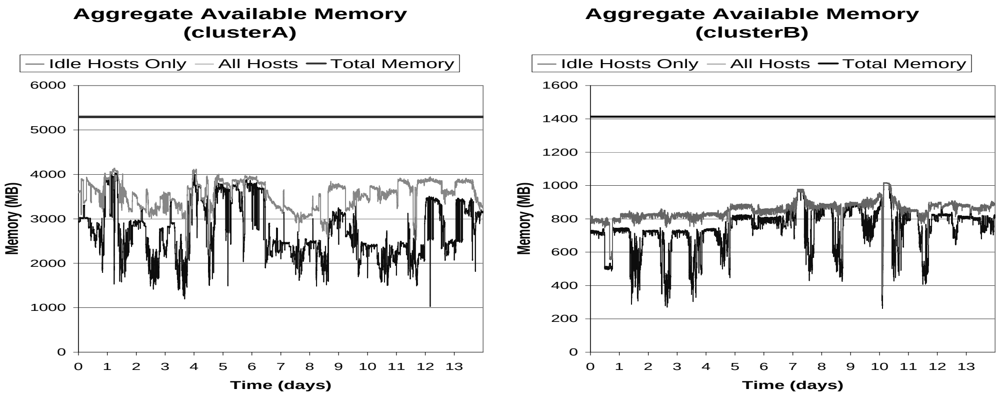
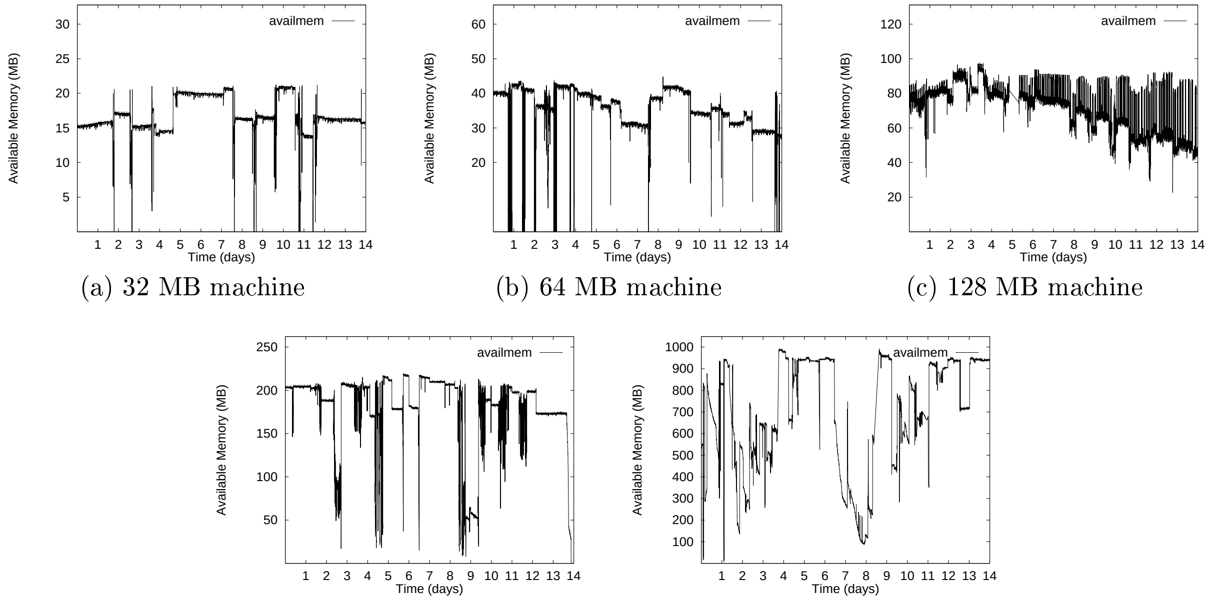


Figure 1: Variation of total memory available in the two clusters. The graphs marked "All Hosts" correspond to memory available on both idle and busy hosts; the graphs marked "Idle Hosts only" correspond to memory available on idle hosts. For `clusterA`, the average available memory for all hosts is 3549 MB and the average available memory for idle hosts is 2747 MB. The corresponding numbers for `clusterB` are 852 MB and 742 MB.

We would like to emphasize that these numbers are *averages* and as such they smooth over sharper variations in memory usage that occur on individual machines. To some extent, they appear to run contrary to a widely held perception that "my workstation pages a lot". To understand memory availability from the point of view of users sitting in front of individual machines, we studied the variation in memory usage for individual workstations. Figure 2 presents the variation in available memory in individual workstations with different amounts of memory. We note that for each workstation, there are several points at which the amount of memory available is very low – these are points at which the workstation is likely to page and the user is likely to perceive the memory of her workstation to be running short. The graphs in Figure 2 also show that even though such dips in memory availability occur frequently enough to be noticed, a substantial fraction of the memory is available on individual workstations most of the time. We conclude the perception of memory being short is based on infrequently occurring worst-case memory requirements and that a substantial fraction of memory on a workstation is available if we look at day-long/week-long intervals.



(a) 32 MB machine (b) 64 MB machine (c) 128 MB machine

(d) 256 MB machine (e) 1 GB machine

Figure 2: Variation in available memory for individual workstations. Note that the range of the y-axis is different for each of the graphs. For each graph, the y-range corresponds to the total memory on the machine. These graphs show that while there are "dips' in memory availability, large fractions of a workstation's memory is available most of the time.

3 The design of *Dodo*

The design of *Dodo* is based on that of Condor [11], one of the most successful systems for harvesting idle resources.[2] Each application is linked to a library that implements the functionality needed by the application in order to create, read, write, and delete remote memory regions. In contrast to global memory systems such as GMS [7] where applications use remote memory pages implicitly, *Dodo* requires applications to make explicit use of remote memory. This is an fundamental trade-off in the design-space for remote memory systems. Implicit memory-sharing systems, such as GMS, allow applications to run without being modified. However, implementing such systems requires modification of the virtual memory system (and

[2] Condor has been operational at the University of Wisconsin for over a decade.

Host type	kernel memory (KB)	file-cache memory (KB)	process memory (KB)	available memory (KB)
32MB hosts	10310 (1133)	2402 (2257)	3746 (2686)	16310 (3844)
64MB hosts	16347 (2081)	4093 (3776)	10017 (6982)	35079 (8030)
128MB hosts	25512 (3257)	8216 (10271)	12583 (12621)	84761 (17623)
256MB hosts	50109 (8625)	7384 (7821)	17606 (23335)	187045 (47535)

Table 1: Average amount of memory used for different purposes. The numbers in parentheses are the corresponding standard deviations.

possibly the processor firmware). Explicit memory-sharing systems, such as *Dodo*, require application-level modification. However, they require no modifications to the kernel or the processor firmware. The history of resource-harvesting systems has shown that portability (with respect to both processor architecture and the operating system) is an important requirement for their success and longevity. User-level systems like Condor have flourished for over a decade through several generations of processors and operating systems. Since our goal was to avoid becoming extinct like our namesake, we selected a user-level design for *Dodo*.

We chose an explicit and synchronous interface for portability and simplicity. *Dodo* has been designed to run on Unix platforms. Note that in *Dodo*, remote memory is used for read-only caching. Writes to remote memory are propagated to disk in parallel to being sent to the remote host. An explicit interface has the disadvantage that it *requires* the programmer to keep track of memory usage and to coordinate all data transfer to and from the remote memory cache. To lighten this burden, we have developed a region-management library that can be layered on top of *Dodo*. This library is linked in with the application and tracks region-usage on the machine it is running on. It implements several region-replacement policies including LRU and *first-in*.

Dodo is targeted towards two environments: (i) non-dedicated clusters of desktop machines, and (ii) dedicated Beowulf-class clusters of commodity-priced workstations [18]. Each machine in a desktop cluster has an owner and is recruited only when the console is not in use *and* the machine is lightly loaded. Each machine in a dedicated cluster is recruited whenever it is lightly loaded.

We first describe the architecture of *Dodo*. We then describe its application-level interface. Finally, we describe the region-management library.

3.1 *Dodo* architecture

In addition to the runtime library mentioned above, *Dodo* has three other components: (i) resource monitors, (ii) a central memory manager and scheduler, and (iii) idle memory daemons. Resource monitors execute on each participating workstation and notify the central manager when a workstation becomes idle. *Dodo* recruits the memory of a workstation only if the workstation is idle. We made this decision for two reasons. First, even though hosting guest memory pages while the workstation is in use 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 [22], processor cycles consumed for handling remote memory requests can, in some situations, degrade the performance of the owner's jobs.

Two kinds of information are used by the resource monitors to determine if a workstation is to be considered idle – mouse, keyboard and processor usage information and user preferences specified as a set of rules. A workstation is considered idle by its resource monitor if there was no user activity (keyboard or mouse) *and* the load (as reported by `w`) is less than 0.3 for five minutes or more.[3] User preferences are used to give the owner of the workstation complete control over when her machine is recruited by *Dodo*. We borrowed the user preference rules used by Condor for the design of *Dodo*.

The design of the *Dodo* central memory manager resembles that of the Condor central manager [19]. The central memory manager runs on a dedicated machine and keeps track of the idle workstations in the cluster as well as the amount of memory available on each workstation. This information is provided by the resource monitors on individual machines. It also keeps a directory of memory objects that have been allocated. The information provided by the resource monitors is taken as a hint – the central memory manager verifies the availability of memory before allocating it. To handle faults at the application-end, the central memory manager periodically sends keep-alive echo requests to the runtime library linked in with the application. If it determines that an application is no longer alive, it reclaims all memory-regions allocated to it.

An idle memory daemon is started on a workstation (by the resource monitor) when it is recruited and is terminated (also by the resource monitor) as soon as the workstation becomes busy. It stores remote memory regions in its address space and supplies them on request. 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.

[3]This measure of workstation idleness has been used in several studies of workstation utilization.

In order to limit the impact of hosting guest data on workstation owners, we restrict the maximum amount of memory that is recruited from a workstation. The idle memory daemon uses a suite of inquiry programs to determine the memory in use (on Solaris 5.6, this suite consists of `netstat`, `ps`, `memps` and `pmem`). To this, it adds (i) the memory in the paging free list (*lotsfree*), and (ii) a "headroom" of 15% of the total memory to account for the files in the file cache that are not currently open but might be opened in near future. The figure of 15% comes from our study of memory usage [2] which indicated that 15% of total memory is usually enough to hold the live files in the file-cache.

Applications use the *Dodo* runtime library to explicitly control the contents of the remote cache. To allocate remote memory for an object, the application makes a request to the runtime library which, in turn, makes a request to the central manager. The central manager (randomly) selects a remote host from the list of hosts it believes to have enough memory to satisfy the request. It contacts the idle memory daemon on the selected machine to allocate the requested space. If the selected host is no longer available (e.g., due to a shutdown or a crash or the user reclaiming the workstation), the central manager tries again with another host. If the allocation is successful, a descriptor is returned to the library which includes the identity of the remote host and a memory-region identifier on that host. The allocation fails if sufficient memory is not available. When this happens, the library 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. When an attempt to access a memory region on a particular node fails (either because the node has crashed, or because it is no longer available for hosting remote memory, or because it has previously dropped the remote memory region that was requested), the library drops all the descriptors for memory regions stored on that node.

3.2 *Dodo* programming interface

Dodo manages the idle memory of a workstation cluster and provides a caching service for applications. Applications access *Dodo* using an API similar to the *stdio* API provided by the C runtime library. The main routines provided by *Dodo* are mopen(), mread(), mwrite(), and mclose() (see Figure 3 for the API).

- mopen() allocates a new remote memory region of given length and associates with it a backing region on disk. The backing region is described by an open file descriptor and an offset within that file. This requires the application to open the file a priori but provides the flexibility of in-place updates on the disk(e.g., for out-of-core scientific applications). It contacts the central memory manager to allocate the memory-region. It returns a non-negative integer as a descriptor for the region. This descriptor is to be used for all subsequent operations on that region. If mopen is unsuccessful due to lack of memory, it returns `-1` and sets `errno` to ENOMEM. If mopen is unsuccessful because the backing file descriptor is invalid or the backing file has not been opened in *write* mode or if the length is less than 1 or if the offset is less than 0, it returns `-1` and sets `errno` to EINVAL.

- mread() tries to read `len` bytes from a given memory-region into `buffer`. The `offset` field can be used to read data from the middle of the region. It returns the number of bytes actually read. If the memory-region is not currently active (if the descriptor is not valid or if the remote host/daemon has crashed or if the memory-region has been reclaimed since it was allocated), it returns `-1` and sets `errno` to ENOMEM. If the arguments are not valid (offset beyond the end or invalid buffer pointer), it returns `-1` and sets `errno` to EINVAL. Finally, if `len` bytes are not available at the request offset, it reads as many bytes as are available.

- mwrite() writes the `len` bytes from `buffer` to the backing file as well as to the given memory-region. The `offset` field can be used to write data into the middle of the region. It returns the number of bytes actually written into the region. If the memory-region is not currently active (if the descriptor is not valid or if the remote host/daemon has crashed or if the memory-region has been reclaimed since it was allocated), it returns `-1` and sets `errno` to ENOMEM. If the arguments are not valid (offset beyond the end or invalid buffer pointer), it returns `-1` and sets `errno` to EINVAL. Finally, if `len` bytes cannot be written at the request offset, it writes as many bytes as possible. If the write to the backing file fails for some reason, it returns `-1` and passes the `errno` value set by the erring call to `write()`.

```
int mopen(size_t len, int fileDescriptor, int offset)
/* returns a memory region descriptor */

int mread(int regionDescriptor, int offset, void *buffer, size_t len)
/* returns the number of bytes read */

int mwrite(int regionDescriptor, int offset, void *buffer, size_t len)
/* returns the number of bytes written */

int mclose(int regionDescriptor)
/* deallocates the regionDescriptor */

int msync(int regionDescriptor)
/* returns only when all data in region is on disk */
```

Figure 3: The *Dodo* application programming interface.

- mclose() deallocates a previously allocated memory-region. It contacts the central memory manager to free up the memory-region. It returns `0` if successful. It returns `-1` and sets `errno` to EINVAL if the descriptor is invalid or has already been reclaimed or if it is not able to contact the central memory manager. It does not close the file descriptor associated with the region.
- msync() blocks till all data in the given region has been written to disk.

3.3 Region-management library

The region-management library is layered on top of the *Dodo* runtime library and is meant for use by applications with well-defined memory access patterns. It tracks region-usage on the machine it is running on and implements several region-replacement policies including LRU and *first-in*. It frees the programmer from the responsibility of coordinating data transfers to/from remote memory. Note that the use of this library is optional – applications whose access patterns do not fit the region-management policies provided by this library can directly access the *Dodo* runtime library.

The region-management library provides a wrapper for every call in the *Dodo* API (see Figure 4 for the interface). It allocates and manages a local cache of memory regions. It provides the `csetPolicy()` call to allow the programmer to specify a region-replacement policy for this cache (LRU/MRU/first-in etc). If no policy is specified, it assumes an LRU policy. It ensures that every region allocated by the application is one of four states: (1) cached locally, (2) cached remotely, (3) cached locally and remotely, (4) not cached in memory (only on disk). Every time control is transferred to the library (via a call to one of its routines), it checks if there is enough space to perform the operation. If not, it scans the cache to find regions that can be migrated to remote cache. The library is modularized to make it easy to add modules for region-replacement policies. To add a new policy module, the following procedures need to be specified:

- A pair of state-management procedures which are called whenever the application calls `cread()/cwrite()`. These functions can keep track of the access-pattern.
- A reclamation procedure which is called whenever the library runs of space in the local region-cache. This procedure takes a pointer to the local cache-directory as an argument.

4 Implementation of *Dodo*

Our implementation of *Dodo* is operational and currently runs on Linux 2.0.35. It consists of three daemons and two runtime libraries. The daemons are the *central manager daemon* (`cmd`), the *resource monitor daemon*

```
int copen(size_t len, int fileDescriptor, int offset)
int cread(int regionDescriptor, int offset, void *buffer, size_t len)
int cwrite(int regionDescriptor, int offset, void *buffer, size_t len)
int cclose(int regionDescriptor)
int csync(int regionDescriptor)
int csetPolicy(int policy)
```

Figure 4: The region-management library interface. The `csync()` call can be used to force an immediate write for a region in the local cache. It blocks till the region has been written to remote memory and to disk.

(`rmd`) and the *idle memory daemon* (`imd`); the runtime libraries are `libdodo.a`, the main *Dodo* runtime library and `libmanage.a`, the region management library layered on top of `libdodo`. For communication, it can use either UDP/IP or *U-Net*, the low-latency user-level network architecture developed by von Eicken et al [4]. For programming convenience and portability, we have developed a library, `libusocket.a`, that provides functionality similar to UDP sockets on top of U-Net. The complete distribution is about 10,000 lines of code. In this section, we describe different components of the *Dodo* implementation.

4.1 The resource monitor daemon (`rmd`)

A resource monitor daemon runs on every participating machine and monitors the user activity as well as the load on the machine. Its functions are: (1) determine when the machine becomes busy/idle; (2) notify the central manager daemon about changes in machine status (busy/idle); (3) create the idle memory daemon when the machine is recruited for hosting guest data; and (4) kill the idle memory daemon when the machine is reclaimed.

To monitor the status of the machine, `rmd` checks mouse/keyboard activity as well as process load once every second. It defines the console of a machine to be idle if there is no activity on either the mouse or the keyboard (it uses the `stat` system call to monitor the access times for the corresponding device files). It defines the processor to be idle if the process load is greater than 0.3. It uses two sources of information to determine the process load. It uses information from the file `/proc/uptime` to determine the average load on the machine. To eliminate the impact of the screen saver and the idle memory daemon on the process load number, it determines the processor usage of these processes if present. It defines the processor to be busy if the remaining process load is greater than 0.3. It defines a workstation to be idle when both the console and the processor have been idle for more than five minutes.

When a machine switches state from busy to idle, `rmd` notifies the central manager daemon and forks the idle memory daemon. When a machine switches state from idle to busy, `rmd` notifies the central manager daemon and sends a signal to the idle memory daemon. The idle memory daemon handles the signal by completing the ongoing transfers and exiting.

4.2 The idle memory daemon (`imd`)

The idle memory daemon is forked by `rmd`. It allocates a memory pool on startup and manages its usage. The size of the pool is determined by estimating the amount of memory in active use (as described in section 3). It initializes an epoch counter which it uses to timestamp the remote objects it caches. It also spawns a thread that receives and serves memory-region read/write requests. All communication with the central manager daemon is handled by the main thread.

During initialization, `imd` sends two items of information to the central manager daemon: (1) the amount of memory available on the machine, and (2) the current value of the epoch counter. The idle memory daemon receives and serves *alloc* and *free* requests from the central manager. It maintains a local directory to keep track of the memory-regions currently allocated. Once allocated, a memory-region remains active till either a corresponding *free* request arrives from the central manager or if `imd` terminates. The central manager issues a *free* request under two conditions: (1) it receives a deallocation request from the application, and (2) it receives no response to its keep-alive echo request within a threshold time period.

Since memory-regions can be of arbitrary size, there is a potential for fragmentation. The idle memory daemon uses a first-fit policy for allocation and periodically runs a coalescing algorithm to reduce fragmentation. Given the explicit programming interface provided by `Dodo`, we expect that individual memory-regions will be large and, usually, multiples of the pagesize. Furthermore, since applications free memory-regions infrequently (usually just before termination), we don't expect fragmentation to be a major problem. Our experience to date has matched this expectation. If this becomes a problem at a later date, we plan to switch to a buddy-based allocation scheme.

4.3 The central manager daemon (`cmd`)

The central manager daemon runs on a dedicated machine and keeps track of the idle workstations in the cluster as well as the amount of memory available on each workstation. This information is provided by the resource monitor daemons on individual machines. It also keeps a directory of memory objects that have been allocated. It maintains the following data structures:

- **idle-workstation-directory (IWD):** which keeps track of the currently idle workstations. For each idle workstation, it records the last known epoch and the largest block that is known to be free. This information is provided by the idle memory daemons and is taken as a hint – the central memory manager verifies the availability of memory before allocating it. This information is piggybacked on all communication between the individual `imds` and the `cmd`.
- **region-directory (RD):** which keeps track of all allocated regions. For performance reasons, the RD is organized as a hash table. Regions in the RD are keyed by a region-descriptor which consists of a *(inode-number-of-backing-file, offset-in-file)* pair.[4] A region in RD is represented by a structure containing the IP address of the machine that is hosting the region, the offset in the memory pool of the corresponding `imd`, the length of the region and an epoch-based timestamp.

The central manager daemon interacts closely with the *Dodo* runtime library. It exports three operations to the runtime library and, in turn, requires the library to echo the periodic keep-alive messages it sends to the library. The keep-alive messages allow `cmd` to reclaim memory-regions allocated by applications that terminate without freeing these regions. If it determines that an application is no longer alive, it reclaims all memory-regions allocated to it. The operations it exports to the runtime library are:

checkAlloc: this operation checks if a region is currently valid. On receiving this request, the `cmd` looks up the region-descriptor in the region-directory. If the lookup fails, it returns a failure to the client. Else, it checks to make sure that the epoch timestamps of the region-structure and that of the host workstation's entry in the idle-workstation directory match. If they do, it returns the region-structure to the client; else it deletes the region from the RD and returns a failure to the client.

alloc: this operation allocates a new memory-region. On receiving this request, the `cmd` scans the idle-workstation-directory and extracts the list of workstations which it believes have at least one memory block larger than or equal to the size of the request. It then randomly picks a workstation from this list and issues a request to its `imd`. If the allocation is successful, it enters the new region in the region-directory and returns the region-structure to the client. Else, it selects additional hosts at random till either it is able to allocate the requested space or there are no more hosts with adequate free memory.

free: this operation frees a memory-region. On receiving this request, the `cmd` looks it up in the region-directory. If a match is found, it forwards the request to the corresponding `imd` and deletes the entry in the region-directory. Else, it returns a failure to the client.

[4] This assumes that only one client is using *Dodo* at a time. We plan to extend this, in near future, to multi-client configurations by including the IP address of the client in the key.

4.4 The *Dodo* runtime library

In this section, we describe two components of the runtime library which have not been described in other parts of this paper: (1) a protocol to transfer data between the client and the remote-memory servers; and (2) a region-table used to keep track of memory-regions created by the application. A data transfer protocol is needed as memory-regions can be of arbitrary size and cannot be expected to fit within individual packets ($\approx$ 1500 bytes for U-NET and 64 KB for UDP).

Bulk data transfer protocol: If the region does not fit in a single packet, it is partitioned in several packets, and a sequence number is added to keep track of the total ordering. The sender negotiates the amount of space available at the receiver and blasts as many packets as would fit in that space and waits. The receiver waits for the same number of packets to arrive or a timeout to occur. If a timeout occurs, it identifies the missing packets using the sequence numbers and sends a selective NACK which lists the missing sequence numbers. Since we assume all hosts are on the same local-area network, we do not expect duplicate packets.[5] We use `iovec` structures and the *sendmsg* and *recvmsg* operations to avoid copying to and from a temporary buffer: received data is directly copied from the receive socket buffer into the memory location of the block.

Region-table: keeps track of all regions created by the application. It is keyed by the region-handle and stores a structure containing the following information: (1) a local/remote flag; (2) the region-descriptor if the region is remote; (3) size of the region; and (4) a unique identifier for the region.

4.5 Region-management library

The region-management library manages a memory pool similar to that managed by the `imds`. Every time control is transferred to the library (via a call to one of its routines), it checks if there is enough space to perform the operation. If not, it scans the cache to find regions that can be migrated to remote cache (see Figure 5 for the algorithm used). The library is modularized to allow multiple region-replacement policies. To add a new policy module, the following procedures need to be specified:

- A pair of state-management procedures which are called whenever the application calls `cread()/cwrite()`. These functions can keep track of the access-pattern.
- A reclamation procedure which is called whenever the library runs of space in the local region-cache. This procedure takes a pointer to the local cache-directory as an argument.

Currently, the library provides two policies: *LRU* and *first-in*. The *first-in* policy is useful for applications that sequentially scan their entire dataset multiple times. It caches regions in the order they were initially accessed. Once a region is cached, it is not replaced. The decision to implement the *first-in* policy was motivated by Uysal et al's study of the access patterns of a large suite of data-intensive applications [20]. They found that a large fraction of such applications have a sequential-scan or triangle-scan I/O access patterns.

4.6 The U-Net-based socket library

U-Net is a low-latency user-level communication mechanism [4]. U-Net was designed to avoid the overhead of operating systems by allowing the user to have access directly to the network interface. We downloaded the U-Net distribution and ported it to our Network Interface Card, the SMC Etherpower 10/100 (SMC9332BDT). For programming convenience and portability, we implemented a library with a UDP-socket-like interface on top of U-Net (see Figure 6 for the API for this library). We also tuned our ethernet device driver for use with this library.

[5] The protocol can be extended to handle duplicate packets by dropping them.

```
procedure grimReaper()
   static integer lastFailTime ← getCurrTime();
   integer currTime, waitTime;
   region R, M;

   while (freeSpace(cache) < lowWaterMark)
      R ← getRegionToEvict(currPolicy);
      if (dirtyRegion(R))
         writeToDisk(R);
      end
      currTime ← getCurrTime();
      waitTime ← currTime - lastFailTime - refractionPeriod;
      if (waitTime > 0)
         wait(waitTime);
      M ← cloneRemoteRegion(R);
      if (M == null) /* no space in remote cache */
         /* do no allocation for refractionPeriod */
        lastFailTime = currTime;
      else /* region moved to remote cache */
         createRemoteEntry(M);
      end
      removeLocalEntry(R);
   end
end
```

Figure 5: Procedure used by the region-managment library to reclaim space.

```
int       u_socket(int sendbufsize, int recvbufsize);
int       u_close(int usockfd);
macaddr_t u_aton(const char *str_addr);
char *    u_ntoa(const macaddr_t macaddr, char *str_addr);
int       u_bind(int usockfd, macaddr_t *macaddr, int nbaddr);
int       u_connect(int usockfd, const macaddr_t macaddr);
int       u_send(int usockfd, const void *buff, size_t len);
int       u_send_iovec(int usockfd, struct iovec *iov,int iovc);
int       u_recv(int usockfd,void *buff, size_t len,
                 macaddr_t *macaddr, int timeout);
int       u_recv_iovec(int usockfd, struct iovec *iov, int *iovc,
                       macaddr_t *macaddr, int timeout));
```

Figure 6: Programming interface for the *usocket* library.

5 Performance Evaluation

5.1 Experimental Platform

Our experimental platform consisted of a 16 node Beowulf-class cluster running Linux 2.0.35 at George Mason University. Each node in this cluster has one or two 200 MHz Pentium Pro processors, 128 MB of memory and a 3.2 GB Quantum Fireball ST3.2A disk. The nodes are connected by a 16-port BayStack 350 10/100 Autosense Fast Ethernet switch. Each node has a SMC Etherpower 10/100 PCI Network card (SMC9332BDT), which is capable of full-duplex operation and is based on the Dec "Tulip" DS21140 chip.

We used 14 of the 16 nodes on the cluster for our experiments. Of these, one node was used for running the data-intensive application, and another was used for running the central manager daemon (`cmd`), while the remaining 12 nodes were used for running idle memory daemons and hosting application data. Each `imd` allocates a memory pool of 100 MB on startup; thus, the total remote memory available to an application in our experiments is 1200 MB. In addition, we used the region-management library which allocated a local cache of 80 MB on the node being used to run the application. The application's dataset is stored on the local disk attached to this node. This disk (a 3.2 GB Quantum Fireball ST3.2) has an average seek time of 10 (11) ms for reads (writes), maximum seek time of 12 (13) ms for reads (writes) and a rotational speed of 5400 RPM. The application-level bandwidth[6] achievable for this disk is 7.75 MB for sequential 8 KB/32 KB read requests, 0.57 MB/s for random 8 KB requests and 1.56 MB/s for random 32 KB requests. For the experiments involving the use of the U-Net communication library, we loaded a modified version of the U-Net Ethernet driver for the DEC "Tulip" chip.

5.2 Benchmarks

We used two real applications and three synthetic benchmarks to evaluate the performance improvements that can be achieved via *Dodo*.

5.2.1 Real applications

dmine: this application tries to extract association rules from retail data [3, 13] consisting of 10 million transactions, with an average transaction size of 20 items and a maximal potentially frequent set size of 3. This application has a data set size of 1 GB. It has a *multi-scan* data access pattern; accordingly, we used a *first-in* region-replacement policy. Almost all the read requests made by this application are 128 KB each.

lu: this application computes the dense LU decomposition of an out-of-core matrix [9]. The dataset consisted of an 8192×8192 double precision matrix (total size 536 MB) with a slab size of 64 columns. The data is stored in 8 files. This application exhibits a triangle-scan I/O pattern, with most of its I/O requests being reads. Accordingly, we used a *first-in* region-replacement policy. The average I/O request size of this application is large (330 KB); however, I/O requests range in size from 12 KB to 516 KB.

We chose these applications as representatives of two classes of applications. We chose `lu` to represent computationally intensive applications with large data sets whose memory footprint exceeds the amount of memory available on a single machine but is less than the the total memory available in the cluster. We chose `dmine` to represent a class of applications whose dataset size straddles the total available memory on the cluster.

Another difference between these two applications arises from the manner in which remote memory is used by the applications. In the case of `lu`, all remote memory regions created by an application are deleted at its completion. Thus any performance benefit obtained from remote memory is due to accessing the same object multiple times during a single run. This corresponds to the operation of out-of-core applications that use remote memory to hold temporary data. For `dmine`, remote memory regions are not deleted at the end of a run. Instead, they are retained in remote memory. This corresponds to the operation of applications that access persistent data.

[6] Assuming reads are through the filesystem and not to raw disk.

5.2.2 Synthetic Benchmarks

In addition to these applications, we created three synthetic benchmarks to evaluate the impact of access-patterns not exhibited by the real applications. These benchmarks can be parameterized to study the impact of factors such as dataset size and I/O request size on the performance of the system.

Each of the benchmarks performs `num_iter` iterations; in each iteration, it reads its entire data set from disk according to the access pattern described below. Each read request is of size `req_size` with a constant compute time of 10 ms between requests.

sequential: this benchmark sequentially reads its data set.

hotcold: this benchmark divides its dataset into a 20% "hot" region and a 80% "cold" region; 80% of the references are to the "hot" region. Within each region, the requests are random.

random: this benchmark makes random read requests from the entire dataset.

In our experiments, we considered data sets of 1 GB and 2 GB and `req_size`s of 8 KB and 32 KB. We set `num_iter` to 4 for all experiments. For these benchmarks, all remote memory regions are created by an application during its first iteration, and deleted at its completion. Thus any performance benefit obtained from remote memory is due to accessing the same object multiple times during a single run.

5.3 Results

Figure 7 shows the speedup obtained for `lu` and `dmine` using *Dodo*. The speedup observed for `lu` is modest – 1.2 while using U-Net and 1.15 using UDP sockets. This is primarily due to the fact that while `lu` does a lot of I/O, it is actually compute-bound. It spends only 9% of its time doing I/O. We note, however, that the running time of `lu` for this dataset is over 6 hours and a speedup of 1.2 represents an appreciable reduction in the execution time.

The speedup obtained for `dmine` is significantly higher – 3.2 using U-Net and 2.6 using UDP. `dmine` spends a larger fraction of its execution time doing I/O than `lu` and has a larger data set (1 GB). Note that the first run of `dmine` on the cluster does not exhibit any speedup. The speedup is obtained on subsequent runs since `dmine` does not delete memory regions at the end of a run. In our experiments, the entire dataset for `dmine` can be stored in the available remote memory during the first run. Subsequent runs can avoid all disk accesses leading to the speedups observed.

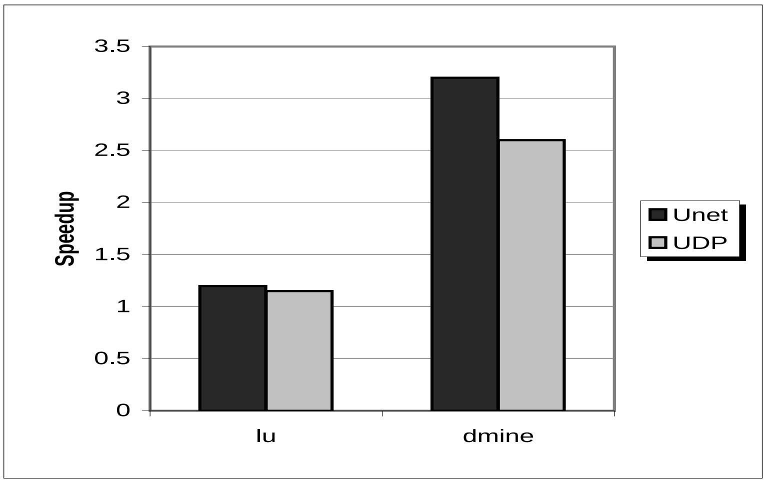


Figure 7: Speedup for `lu` and `dmine` using *Dodo*.

Figure 8 shows the speedups obtained for the three synthetic benchmarks for two dataset sizes (1 GB and 2 GB) and two I/O request sizes (8 KB and 32 KB). We observe that there is virtually no speedup obtained for the sequential benchmark. This is to be expected since the Linux file system is optimized for

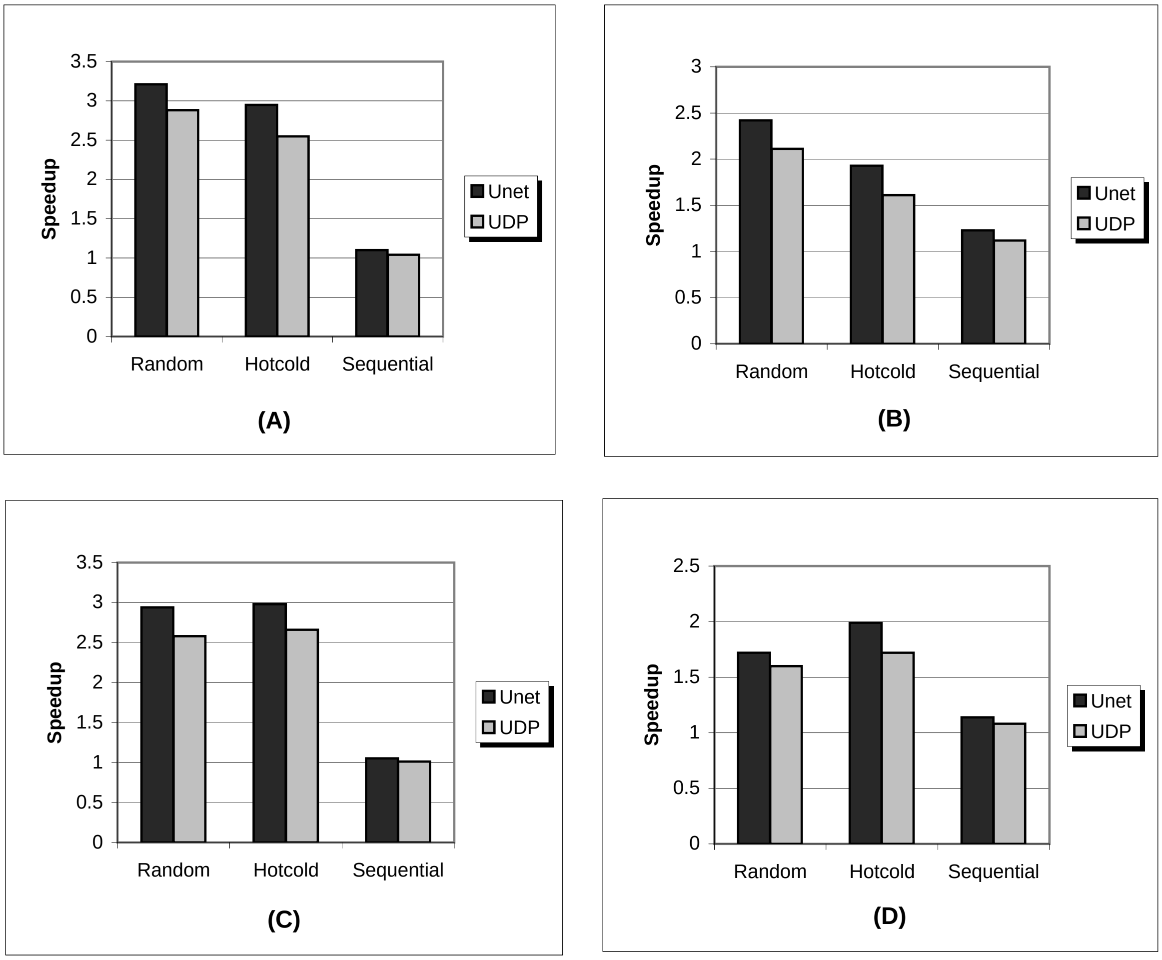


Figure 8: Speedup for the synthetic benchmarks using *Dodo* for (A) 8 KB request size, 1 GB data set (B) 32 K request size, 1 GB dataset (C) 8 KB request size, 2 GB data set (D) 32 KB request size, 2 GB data set.

sequential access patterns and the end-to-end bandwidth attained on the disk for a sequential access pattern is comparable to the bandwidth of the network. For `hotcold` and `random`, the speedup is significantly higher reflecting the random I/O access patterns of these benchmarks.

Increasing the I/O request size from 8 KB to 32 KB results in a decrease in the speedup obtained from the use of remote memory for `random` and `hotcold`. This reflects the fact that these benchmarks become less I/O bound when the I/O request size is increased to 32 KB. However, the speedup obtained for the `sequential` benchmark is slightly higher for the larger block size. This is because the performance of *Dodo* improves for larger block sizes, while the performance of the file system is unaffected by the larger block size for sequential access patterns.

Increasing the dataset size from 1 GB to 2 GB results in a reduction in speedup for both `sequential` and `random`. This reduction in speedup is more appreciable in the case of `random`, since the speedup of `sequential` is small for both data set sizes. This reduction in speedup is to be expected since the entire data set of 2 GB does not fit in the available remote memory (1.2 GB), whereas a data set of 1 GB can fit in the remote memory.

In the case of `hotcold`, however, increasing the data set size from 1 GB to 2 GB results in an increase in the speedup. This reflects the fact that the hot portion of the data set increases in size from 200 MB to 400 MB when the data set size is increased. This does not have a significant impact on the performance of the application when it can use remote memory. However, the performance of the application without remote memory caching is much worse for the larger data set since local file-cache is less effective in reducing the number of disk accesses.

In all cases, the use of the U-Net communication library results in an appreciable performance improvement over UDP sockets. This is to be expected given the higher overhead of UDP in comparison to U-Net. What is noteworthy, however, is the fact that even while using UDP, we can obtain significant speedups for all our benchmarks. Overall, our results show that *Dodo* results in significant performance benefits for data-intensive applications that can exploit the zero-seek nature of remote memory.

5.3.1 Discussion

The results described above show the performance benefits of using *Dodo* to exploit remote memory on a dedicated Beowulf-class cluster. As discussed in earlier sections, *Dodo* has been designed to exploit idle memory in non-dedicated clusters while minimizing any inconvenience caused to owners of workstations. While we have not yet deployed *Dodo* in such a production environment, we have evaluated its performance in such environments via trace-driven simulation. Our results, reported in [2] show that *Dodo* can result in significant performance speedups on non-dedicated clusters. Further, our results show that using a memory recruitment policy that targets only idle hosts and that does not harvest more memory than is idle on the host ensures that users experience virtually no delays when reclaiming their workstations.

6 Conclusions

In this paper, we have presented the design and implementation of *Dodo*, an efficient user-level system for harvesting idle memory in off-the-shelf clusters of workstations. Dodo enables data-intensive applications to use remote memory in a cluster as an intermediate cache between local memory and disk. It requires no modifications to the operating system and/or processor firmware and is hence portable to multiple platforms. Further, the memory recruitment policy used by Dodo is designed to minimize any delays experienced by the owner of desktop machines whose memory is harvested by Dodo.

We have implemented *Dodo* on a Beowulf class Linux cluster. For communication, *Dodo* can use either UDP/IP or *U-Net*, the low-latency user-level network architecture developed by von Eicken et al [4]. We evaluated the improvements that can be achieved by using *Dodo* for two real applications and three synthetic benchmarks. Our results show that speedups obtained for an application are highly dependent on its I/O access pattern and data set sizes. Significant speedups (between 2 and 3) were obtained for two kinds of applications: (i) applications such as `dmine` and `hotcold` whose working sets are larger than the local memory on a workstation but smaller than aggregate memory available on the cluster (ii) applications such as `random` that can benefit from the zero-seek nature of remote memory.

Finally, our results also show that the use of the U-Net communication library results in an appreciable performance improvement over UDP sockets. This is to be expected given the higher overhead of UDP in comparison to U-Net. What is noteworthy, however, is the fact that even while using UDP, we can obtain significant speedups for all our benchmarks.

References

[1] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computation. In *Proceedings of 1997 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, 1997.

[2] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. Technical Report TRCS98-26, Dept of Computer Science, University of California, Santa Barbara, 1998.

[3] 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.

[4] A. Basu, V. Buch, W. Vogels, and T. von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[5] D. Comer and J. Griffioen. A new design for distributed systems: The remote memory model. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[6] 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.

[7] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 201–12, Dec 1995.

[8] 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.

[9] 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.

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

[11] 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.

[12] 512 MB PC100 SDRAM with ECC[7]. Micro X-press Inc. 5406 west 78th street Indianapolis, IN 46268, Oct 1998.

[13] 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.

[14] T. Narten and R. Yavatkar. Remote Memory as a Resource in Distributed Systems. In *Proceedings of the 3rd Workshop on Workstation Operating Systems*, pages 132–6, April 1992.

[15] 128 MB PC100 SDRAM[8]. Advanced PCBoost, PO Box 80811, Rancho Santa Margarita, CA 92688, Oct 1998.

[16] P. Sarkar and J. Hartman. Efficient cooperative caching using hints. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, 1996.

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

[18] T. Sterling et al. Preliminary report: Findings of the first NASA workshop on Beowulf-class clustered computing,. *http://www.cacr.caltech.edu/ tron/pubs/beowulfprelim.html*, Oct 1997.

[19] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobbs' Journal*, 20(2):42–4, Feb 1995.

[20] 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.

[7] *http://www.microx-press.com/online/* following link from *http://www.pricewatch.com*

[8] *http://www.pcboost.com* following link from *http://www.pricewatch.com*

[21] G. Voelker, E. Anderson, T. Kimbrel, M. Feeley, J. Chase, A. Karlin, and H. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43, 1998.

[22] 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 Joint International Conference on Measurement and Modeling of Computer Systems*, pages 127–138, June 1997.