Chunk: A Framework for Modular Distributed Shared Memory Systems

Alexandra Fedorova, Eben Scanlon
fedorova@eecs.harvard.edu, eben@mit.edu

Abstract

We present Chunk, a framework for building modular distributed shared memory systems for UNIX. Chunk allows applications that are designed to share local memory through the UNIX memory mapped file mechanism (`mmap`) to be able to share memory across different physical hosts without modifications. Chunk's modular architecture enables the use of a variety of memory-sharing policies. We present a DSM system implemented using the Chunk framework and evaluate its performance. We demonstrate the viability of our framework for real-world applications by presenting experiments with Berkeley DB.

1. Introduction

In this paper we present Chunk, a framework for building modular distributed shared memory (DSM) systems for UNIX. We describe the architecture of Chunk and the DSM system that we implemented using the Chunk framework for FreeBSD. Chunk allows applications that share local memory through memory-mapped files to share memory across different physical hosts. Applications do not need to be modified or re-linked to use Chunk.

To share local memory using memory-mapped files, the applications must map the same file into their address spaces using the `mmap` system call. They can share memory by having each application read and write the part of its address space corresponding to the mapped file. The Chunk framework emulates the semantics of this memory sharing mechanism. It achieves this by using a kernel *file system module* and a user-level *manager module.*

The Chunk kernel module creates the illusion that the mapped file is located on the local host. In reality it redirects requests to read and write data residing in the shared file to the manager component. It also provides functionality to remove shared memory pages from one host in order to grant exclusive ownership rights on that memory to another host.

The Chunk manager component is a user-level process that implements page-sharing policies and is responsible for moving the shared data between machines and ensuring memory consistency. Because the kernel and manager components communicate through a well-defined interface, the manager component of Chunk is *pluggable*: it can implement any memory consistency algorithm, and can use any network transport protocol.

DSM systems could be implemented either entirely at user-level, or entirely in the kernel. The advantage of the user-level implementation is simplicity, extensibility and isolation of possible bugs from the kernel. The advantage of the kernel implementation is transparency and efficiency. The Chunk DSM system captures the best of both worlds. It achieves flexibility and extensibility through a user-level page sharing and message passing module, and it achieves transparency through a kernel-level module.

We envision using applications built on top of Chunk in a secure cluster environment. High-performance application servers can achieve performance improvements due to parallelism and resource scalability of the cluster. Because we envision using Chunk in this environment, we do not discuss issues of network security.

The rest of the paper is organized as follows: In section 2 we describe the Chunk architecture and the implementation of its kernel component. In section 3 we present our implementation of a Chunk manager. In section 4 we present experimental results. In section 5 we contrast Chunk with related work. We conclude in section 6.

2. The Chunk framework

To achieve transparent memory sharing through the UNIX memory mapped file mechanism, we embed a special file system in the kernel – the *Wrapper File System* (WFS). This file system implements a limited subset of the Virtual File System (VFS) / VNODE interface [6] to provide enough functionality to allow mapping a file into an application's address space. WFS represents the kernel component of Chunk. WFS can be mounted on UNIX as a regular file system. A program that desires to share memory through Chunk must access shared files whose path begins with that of the WFS mount directory. When a file has been accessed in this way, the kernel redirects all file operations to the WFS file system module. The WFS module, in turn, forwards those requests to the user-level component of Chunk, the Chunk *manager*. The basic operation of the Chunk architecture is shown in Figure 1. WFS also provides functionality for invalidating memory pages in an application's address space in order to grant ownership rights on that memory to an application running on another physical host.

The Chunk manager communicates with Chunk managers on other hosts involved in memory sharing in order to maintain memory consistency and satisfy the WFS file lookup and I/O requests.

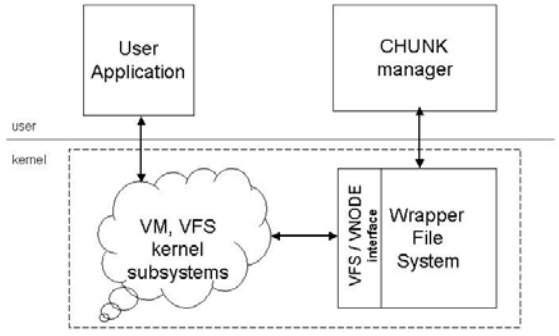

Figure 1. The Chunk architecture.

2.1 The Wrapper File System interface

WFS implements a subset of the VFS / VNODE interface that the operating system uses when an application memory-maps a file. We describe the functions that the WFS implements below. The functions marked with an asterisk involve interaction with the Chunk manager:

The VFS interface:

`wfs_mount` – Performs basic initialization.

`wfs_getroot` – Returns a root vnode.

`wfs_unmount` – Performs clean-up.

The VNODE interface:

`wfs_lookup*` – Called by the operating system when doing a name lookup on a file located in the WFS mount directory (usually /wfs). The WFS module communicates with the Chunk manager to check whether the file exists.

`wfs_access` – Called to check access permission on a file before it is opened. WFS does not currently perform any access checks.

`wfs_open` – Called when a file is opened.

`wfs_getattr` – Called to request the attributes of a file. The only attribute that is required to support file-mapping functionality is *size*.

`wfs_bmap` – Called by the kernel to map a file's logical block to the corresponding physical block on the disk. WFS simply returns the logical mapping.

`wfs_strategy*` – Called by the kernel VFS layer to perform file I/O. When a file is memory-mapped, this function is called following a VM fault on a virtual address representing the mapped file. WFS interacts with the Chunk manager to request the missing data.

`wfs_putpages*` – Called by the VFS layer when the operating system decides to write a memory page associated with the memory-mapped file.

WFS interacts with the Chunk manager to write out the data.

Whenever the WFS module has to interact with the Chunk manager, it needs to perform the equivalent of a function call to the user process running the manager. In Unix there is no mechanism to perform function calls from the kernel to a user process (*upcall*s), so we needed to implement one.

We implemented upcalls by sending a signal to the user process. The process must catch and handle the signal. For each case when the WFS needs to upcall to the Chunk manager we have introduced a new signal to the kernel: SIGLOOKUP, SIGREAD, and SIGWRITE. Since the Unix signaling facility does not allow passing arguments along with the signals, we set up a piece of shared memory that is used to pass arguments from the kernel to the manager during upcalls. We call this piece of memory the *upcall communication buffer*. Before sending a signal to the manager, WFS places arguments into the upcall communication buffer. Upon catching a signal, the manager reads its arguments from this buffer. The events happening during an upcall are represented in Figure 2.

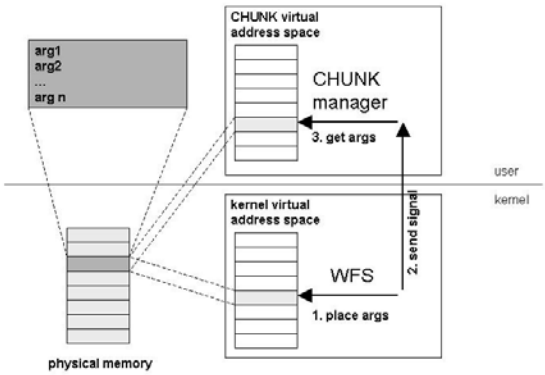

Figure 2. Mechanism for implementing upcalls

While upcalls give the ability to invoke an event handler in a user process, they do not give the ability for the kernel to block until the user process completes the event handling code. Therefore, in order to wait for the user process, the kernel process that performed the upcall goes to sleep on some known address (usually a registration structure associated with the manager). When the manager finishes handling the event, it makes a system call into the kernel, which awakens the sleeping kernel process. We have added the new system calls associated with completion of different events to the FreeBSD kernel.

2.2 WFS core functionality

In this section, we describe the interaction between WFS and the Chunk manager.

1. Registration.

A Chunk manager needs to register itself with WFS, so that WFS can later redirect the lookup and I/O requests to it. In the current implementation, we support only one Chunk manager per system, and one execution thread per Chunk manager. However, WFS could be extended to support multiple managers and multiple execution threads. Registration is performed through the following system call:

```
dsm_register(char *addr, char *pbuffer,
             int pbsize),
```

where `addr` is the address of the upcall communication buffer (see section 2.1). In addition to the upcall communication buffer, the Chunk manager also allocates a scratch buffer that it will use to satisfy read requests from the kernel. The address of this buffer is passed in the variable `pbuffer`, and its size in `pbsize`.

2. Lookup

When an application opens a file for sharing and `wfs_lookup` is called by the kernel VFS layer, WFS upcalls to the Chunk manager to look up the file name. WFS then goes to sleep, waiting for the manager to complete the lookup. When the Chunk manager completes the lookup, it makes the `dsm_lookup_done` system call, which wakes up the process sleeping in the kernel in the WFS code. Once it wakes up, the process completes the execution of the `wfs_lookup` function, returning the status of the file lookup to the VFS layer.

3. Read

When an application generates a page fault on a page shared through Chunk, the kernel calls `wfs_strategy` (see section 2.1) to read the missing data. WFS must ask the Chunk manager to satisfy the read request for the data. WFS upcalls with arguments of the file name, the offset in the file, and the size of the data[1]. The Chunk manager catches SIGREAD and performs the requested I/O into the scratch buffer that it has allocated at registration. Once the I/O is complete, the Chunk manager invokes `dsm_read_done` system call, which wakes up the process sleeping in WFS. If the I/O has failed, WFS returns with error from `wfs_strategy`. If the I/O has succeeded, WFS must perform some virtual memory mappings manipulations before returning.

Just before calling `wfs_strategy`, the kernel allocates a physical memory page where it expects the WFS module to deposit the data. But the actual I/O is performed by the Chunk manager into its scratch buffer. Therefore, WFS must do the following:

- dispose of the page that the kernel had allocated for I/O;
- unmap the scratch buffer from the Chunk manager's address space;
- map the scratch buffer into the address space of the application that generated the page fault and triggered the I/O.

Inside the FreeBSD kernel, the address space of an application is represented by several *virtual memory objects* (vm_objects). When an application maps a file into its memory, the corresponding part of its address space is represented by a separate vm_object. All physical pages containing the data for the mapped file are mapped into the corresponding vm_object. When several applications share a file by mapping it into their address spaces, they all use the same vm_object. When WFS performs page re-mapping, it maps the physical memory page containing the scratch buffer into the vm_object representing the mapped file. Therefore, this re-mapping is automatically reflected in the address spaces of all applications sharing the file on this host. The page-mapping manipulations performed during `dsm_read_done` are shown in Figure 3.

Instead of re-mapping the pages, we could have simply copied the data from the scratch buffer into the physical page pre-allocated by the kernel. However, this extra data copy would add overhead to the page-fault handling operation.

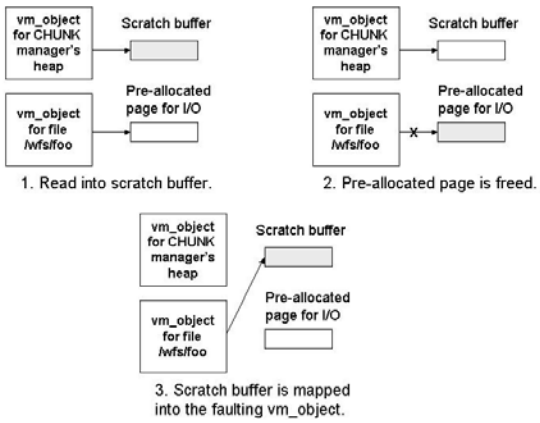

Figure 3. Page mapping manipulations during read operation

4. Write

Shared data is written for two reasons. First, when a dirty shared page is invalidated by the kernel because the system is low on physical memory. Second, when the buffer daemon periodically flushes dirty file data to disk. In each of these cases, WFS delivers requests to write data to the Chunk manager. We describe how the Chunk manager decides where to write the data in section 3.

Write requests are delivered to the manager by sending it a SIGWRITE signal. A write is a very simple operation. The Chunk manager simply maps the piece of file that needs to be written into its address space and gets the data that needs to be written directly from its memory. Once the data has been written, the manager makes a `dsm_write_done` system call into the WFS and wakes up the process that has been waiting for the write to complete.

[1] Right now, all I/O is happening in units of physical page size, which is 4KB on the hardware we are using.

5. Invalidation

Page invalidation needs to occur when some host involved in distributed memory sharing requests exclusive ownership rights on a particular memory page. The following two events need to happen on invalidation:

- the requested page needs to be unmapped from the vm_object representing the shared file
- the page needs to be stored somewhere where the Chunk manager can find it, so that it can send it to the requesting party.

Invalidation is performed through the following new system call:

```
dsm_invalidate(char *path,
          int offset, char *vaddr),
```

where `path` is the name of the file, `offset` is the offset in the file corresponding to the beginning of the page that has been requested for invalidation, and `vaddr` is the virtual address in the Chunk manager's address space where the invalidated page will be stored. The `vaddr` argument must be a valid unused address in the manager's address space. During invalidation, the physical page that contains the needed data is unmapped from the vm_object representing that file and mapped into the manager's address space to back `vaddr`. Note that while we could have copied the data from the invalidated page into the location corresponding to `vaddr`, we do page re-mapping instead, in order to avoid costly data copying. Upon returning from `dsm_invalidate`, the Chunk manager takes the data located at `vaddr` and sends it to the requesting party. `dsm_invalidate` returns to the caller the dirty status of the page.

2.3. Locking

Conventionally, DSM systems provide their own locking facility that can be used by applications to implement critical sections that involve accessing remotely shared memory. Implementing such a facility for Chunk is outside the scope of the current work. Absence of a locking facility is not, however, a limitation of our system. A Chunk application that needs to use locking could simply implement spin-locks whose structures are located in a shared memory segment. In fact, this is one of the lock implementations available in Berkeley DB. If configured to use spin-locks, Berkeley DB can be used on Chunk without limitations.

3. The Chunk manager

We have implemented a simple user-level manager to test the functionality of the Chunk pluggable architecture. Our goal in the design of this user-level manager was consistency and correctness rather than performance because we anticipate that the main value of Chunk will be the future implementation of custom user-level managers.

In our design, there is a single manager that runs on every host within the shared memory set. These managers implement an identical code base and differ only in their startup configuration files. These configuration files specify every other host in the shared memory set by name and communication port.

3.1. Manager overview

In our implementation of the Chunk user-level manager, we allow only one copy of every memory page to be extant in the system at any one time. Although this design precludes read caching, it is easy to attain correct and consistent behavior in this model. Because there is only one copy of each memory page, we do not distinguish between read and write privileges for this page, and we track only the location of each page.

When a manager is first started on a host, it searches an on-disk directory specified in its configuration file. The files contained in this directory will be available for sharing to other hosts in the Chunk system. For example, if the Chunk manager has file "foo" in this directory, an application using Chunk for memory sharing can specify that it wants to memory-map file /wfs/foo. The lookup request for this file will be forwarded by a Chunk manager running on an application's host to the Chunk manager that owns the physical file "foo". Each file in a particular manager's physical directory can exist on only one host within the system, and every

file must be an integral number of memory pages in size. The implications of this particular limitation are obvious and rather broad. Files can be modified in place but cannot be created or extended dynamically.

Each manager is considered the "owner" of all of the files that reside on its local disk. The owner of a file has elevated responsibilities with respect to this particular file. It tracks the location of pages within this file as they move throughout the shared memory system, and it is ultimately responsible for writing dirty pages from this file back to disk.

3.2. Network communication

In our implementation of the Chunk user-level manager, we use UDP as a communication protocol. Because we use UDP, hosts do not establish connections to one another at startup, but rather establish a single listen port and then create the necessary socket-based framework for sending messages to all of the other hosts in the system. Note that we assume that messages are not lost or corrupted. This is a realistic assumption for the cluster environment that we target, where hosts often communicate over reliable physical transport such as FibreChannel.

The manager was designed so that network communication between hosts is asynchronous and non-blocking. This design allows hosts to perform multiple simultaneous tasks without waiting for remote memory operations to complete, but it requires that state be kept between the initiation of a memory task (signified by the delivery of a signal) and the final completion of this task (signified by a system call).

State is kept by passing messages between hosts that contain all of the information necessary to complete a memory procedure. In this way, state is encapsulated within the messages themselves, thereby relieving the managers of the requirement that state be kept on a per-host basis.

3.3. Handling SIGLOOKUP

When the manager receives a SIGLOOKUP signal from WFS, it first creates a structure to keep track of replies from other managers. It then broadcasts a message to all of the other managers, requesting the owner and the size of the file. The file name is used as a unique identifier of the file throughout the system.

As each manager receives the request, it checks whether it is the owner of the requested file, and if so it responds with a message indicating ownership and file size. All other managers simply respond with a negative message.

Once all of the replies have been received, the original manager stores the file owner and size in a manager-global structure and finally completes the lookup call by calling `dsm_lookup_done`.

Note that all communication between the managers is asynchronous, and that the requesting manager is free to service subsequent signals and network messages after broadcasting a lookup message to other managers. In this way, non-blocking event-driven behavior is maintained.

3.4. Handling SIGREAD

When a manager receives a SIGREAD signal from WFS, it should always be the case that the file whose page is being read has previously been looked up via a call to SIGLOOKUP. Since the file has been previously looked up, the manager knows both the size of the file and the identity of the file owner.

The first step in processing a read request is to send a message to the manager that owns the file requesting the current location of the page. The file owner responds with the identity of the current page owner. The requesting host can then request the page directly from the current page owner.

When the current page owner receives a request for a page, it first invalidates the page with a call to `dsm_invalidate`, and then responds directly to the requesting host with the contents of the page, and an indicator of whether the page is clean or dirty (i.e. whether it has been modified since it was last written to disk.)

When the requesting manager receives the page from the previous page owner, it writes the contents into its scratch buffer (see section 2.2)

and then calls `dsm_read_done`. Finally this manager sends a message to the manager that owns the file to inform it that the read completed and that it is the new owner of the page.

There is one subtlety to this process that deserves mention. It is possible that a page has been silently evicted by the VM system of the current page owner. If this is the case, then when the current page owner receives a request for a particular page and tries to invalidate this page, the WFS will respond that the page is not present. In this event, a message is sent to the requesting manager instructing it to request the page from the manager that owns the file, and has this page on disk.

3.5 Handling SIGWRITE

When a manager receives a signal to write a page from WFS, it first copies the contents of the page to a user level buffer. Our current model for handling writes of a memory page is that writes are always directed to the manager that owns the physical file for the page, and that the owning manager responds to write messages by writing the page back to disk. As was mentioned previously, writes can occur either because a host attempted to swap a page to disk or because the VM system on a host is silently writing a dirty page that it is still in its active set.

Because the manager that owns a physical file can receive a write request even when the current page owner has not ceded page ownership, the manager that owns the physical file does not change the owner of the current page when handling a write message.

3.6 Locking protocol

A lock is maintained for each page in a file by the manager that owns the file. When the file owner receives a request for the owner of a particular page, it first checks to see if the page is currently locked. If the page is not locked, then the manager locks the page and responds to the requesting manager with the identity of the current page owner. If the page is locked, then the location of the page is considered to be changing, and the current request is queued pending the completion of the current page translocation. The queue is simply a list of structures, each of which contains all of the information necessary to restart a request.

Requests to write a page are handled in a similar fashion by the file owner. Before a manager can write a page back to the manager that owns the physical file, it must first request a lock for that page. If the lock request succeeds, then the page is locked at the owner, and the page can be written back to the owner. If the lock fails, then this means that another host is in the process of reading this page. In this case, the manager that is trying to write the page is not granted the lock and indeed does not need to take any action at all. It will soon receive the read request from a remote host, and can fulfill this request from its user-level write buffer. This is a very subtle point in the locking protocol. Note that a requested write-lock can never fail due to another write-lock because there is only one copy of each page in the system at any one time.

When the manager that owns a file receives a “transaction completed” message from the host that requested a lock, then it restarts pending locked requests on this page.

This rather simplistic locking protocol works well for the single-page manager that we implemented for this project, but it would need to be modified for more complex page management algorithms.

4. Experimental results

We have experimented with our Chunk DSM system on a pair of Pentium III 800 MhZ workstations connected by a 100 Mbps Ethernet. Each machine was equipped with 512 MB of physical RAM.

4.1. Microbenchmarks

We begin by presenting the microbenchmarks that demonstrate the performance of basic Chunk primitives. Table 1 compares the performance of Chunk to the local case, whenever appropriate. The numbers reported in the table are averages of 10 measurements. Standard deviation is reported in parentheses.

Null page fault measures the time to handle a page fault discounting the time to fetch the

data. For the local case, this is the time to set up page translations when the file is already cached in memory. For Chunk, it is the time to upcall to the Chunk manager and wait for the manager to execute the `dsm_read_done` system call. For this experiment we modified the Chunk manager to call `dsm_read_done` immediately after receiving an upcall, without fetching the data. Null page fault takes a significantly longer time on Chunk because it involves two context switches: from the faulting application to the Chunk manager, and back.

Full page fault measures the time it takes to fetch a missing page from the remote Chunk host. For the local case, we arranged that the mapped file is located on an NFS server in order to achieve comparable I/O overhead. Full page fault performs comparably on both configurations, because the network communication cost dominates the overhead. The performance is slightly slower for the case when the page has to be retrieved from the NFS server, because the NFS protocol is heavier than the Chunk communication protocol.

Invalidation measures the duration of the `dsm_invalidate` system call.

	Local (µs)	**Chunk (µs)**
Null page fault	4.0 (0.6)	65.7 (2.6)
Full page fault	1184.8 (95.4)	1082.3 (77.6)
Invalidation	-	20.5 (0.9)

Table 1. Microbencmarks

4.2. Simple read/write file sharing benchmark

We now present a benchmark with two hosts sharing a 64KB file. In this experiment each application maps the file into its address space and traverses it sequentially, byte after byte. The application modifies the first byte on each page it traverses. Page size corresponds to the physical page size – 4096 KB. We map the file with MAP_NOSYNC flag to avoid the write requests resulting from the buffer daemon activity, which periodically cleans the file dirty buffers. To simulate computational activity, each page in the file is processed 1000 times. We vary the amount of sharing between the two hosts from 0% (only one host is running at a time) to 100% (all pages are shared among simultaneously running applications). To produce the specific amount of sharing we have one application map the upper part of the file into its address space, and the other to map the lower part of the file. The amount of overlap between the mapped parts determines the percentage of sharing. Regardless of the amount of data mapped, the application always traverses a total of 800 file pages. With this experiment we wanted to show how the system performance degrades as the amount of sharing increases. Figure 4 shows the results.

The numbers reported are averages over 5 runs. Standard deviation is below 5%. The figure displays the running times (in seconds) for the application running on the host that owned the shared file and for the application running on the non-owner host. When the percentage of sharing is high, the application running on the file-owner host finishes marginally faster than the other application, because it does not have to send an extra message to the file owner to inquire about the owner of the missing page (see section 3).

As expected, the system performance degrades as the amount of sharing increases. When the amount of sharing, and therefore the page fault rate, is low, the running time of Chunk is equivalent to that when no sharing occurs and all data is accessed locally. This is the case because this workload is CPU-intensive and the Chunk communication overhead is insignificant, compared to the application CPU overhead. With a workload that is less CPU-intensive, the performance difference between these two cases would be larger. We present such a workload in the next section.

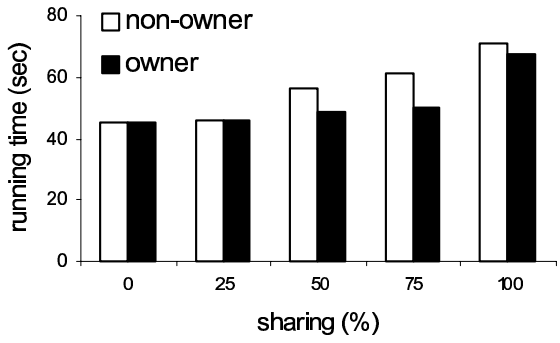

Figure 4. Effect of increased sharing.

4.3. Berkeley DB

A possible target application for Chunk is a program built using Berkeley DB [5]. Berkeley DB is an embedded database library with transactional support for storage and retrieval of persistent data. A typical application that uses Berkeley DB is a server that needs database functionality, but that does not require a full-featured relational database. Such applications can benefit from running on a cluster and transparently sharing memory through a Chunk DSM system.

While Berkeley DB allows sharing its databases among processes running on the same physical host through the `mmap` interface, it limits remote sharing. The only way for processes running on different physical hosts to share a Berkeley DB database is to access it through the Berkeley DB RPC server. This method of access, however, provides only a limited subset of the Berkeley DB's functionality. For example, when using the RPC server, the application is not allowed to create or update secondary indices or to access the database from multiple threads of control.

In this section we present experiments where two applications running on different physical hosts share a Berkeley DB database. The purpose of this experiment is to demonstrate the viability of a Chunk DSM system for real applications.

In this experiment an application retrieves 100,000 records, randomly selected from a 350 KB database filled 64-byte records. It computes an MD5 hash of each record. We run two such programs simultaneously, and compare the performance in three configurations: (i) the database is shared locally, (ii) the database is shared through Chunk, and (iii) the database is shared through the Berkeley DB RPC server. When using the RPC server, the application has to send a message to the server on every database access. In section 4.1 we saw that with Chunk, the host that owned the shared file executed faster than the non-owner host, because it had to send fewer network messages when handling page faults. To avoid this inequity here, we configure a third Chunk host to own the shared database file. Table 2 reports the results.

In this experiment, the RPC configuration outperforms Chunk. This happens because Chunk applications experience a high page fault rate (about 40%), and handling a page fault is more costly than issuing a Berkeley DB transaction over RPC.

Configuration	Running time (sec)
Local	3.6
Chunk	40.2
RPC	24

Table 2. Sharing a Berkeley DB database (1).

We were also interested to show a scenario where the page fault rate is lower. In such a scenario we expected Chunk to outperform RPC, because RPC requires a network trip on every database access, while Chunk would require a network trip only on a page fault. To modify this experiment to have a lower page-fault rate, we had each application, instead of computing an MD5 hash of each record, encrypt 0.01% of all records using DES encryption. DES encryption is a costly computational process, which takes tens of milliseconds on our system using the UNIX `libcrypt` library. Therefore, while one application performs this lengthy computation, another host gets a chance to perform a large number of shared memory accesses without the other host stealing those pages. This configuration reduces the total number of page faults in the system. Table 3 shows the results.

Configuration	Running time (sec)
Local	6.5
Chunk	7.4
RPC	27

Table 3. Sharing a Berkeley DB database (2).

As expected, Chunk does better than RPC. In fact, since the page fault rate is low (only 1%), Chunk performs comparably to the local configuration. This result is consistent with the one in section 4.2, where for low amount of sharing Chunk performed comparably to the local case.

5. Related work

Many DSM implementations have been reported in the literature. Of these, some have been implemented entirely in the kernel [2,7], others entirely in user space [1,3], and a few have taken a hybrid approach [4].

Of the DSM systems reported in the literature the one that is most similar to ours has been implemented by Souto and Stark [4]. Their system has a hybrid design, with a kernel pager component and a user-level server component. Applications, however, need to be modified to use their system. Additionally, our idea of architecting the kernel component in a form of a file system makes our implementation more modular and less intrusive to the kernel code base than theirs. Because our kernel system component does not integrate as closely with the host VM system as theirs, our framework is more portable to different flavors of UNIX.

The goal of our work was not to create yet another DSM system, but to design a framework that allows system programmers to implement DSM systems that can be used by unmodified applications through a familiar memory-mapped file interface.

As an example of the flexibility of our system, we essentially implemented the IVY kernel-level distributed manager algorithm at the user-level using Chunk. In addition, we believe that it would be quite simple for a system like Treadmarks to implement its page replacement algorithms using the Chunk framework. In both cases, we believe that Chunk provides a cleaner system design, leaving page management in the kernel and implementing policy at the user level.

6. Conclusions

In this paper we have presented Chunk, a framework for building transparent DSM systems accessible through the Unix memory-mapped file mechanism. We have implemented a DSM system using the Chunk framework. The applications that we ran on the Chunk DSM system had been written to share local files and they did not need to be modified to run on Chunk. We have demonstrated the viability of Chunk for the cluster environment by presenting an experiment with Berkeley DB – a database library commonly used in high-performance application servers.

From our experimental results we concluded that the bulk of the Chunk page-fault handling overhead results from context switching between the faulting process and the Chunk manager, but that cost is insignificant when compared to network communication cost. We have also concluded that for CPU-intensive workloads with low page-fault rates, sharing distributed memory over Chunk is comparable in terms of performance to local memory sharing.

7. References

[1] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", *In Proc. of the Winter 1994 USENIX Conference*, pages 115-131, January 1994.

[2] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", *In Proceedings of the International Conference on Parallel Processing*, 1988

[3] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. *In Proceedings of the '93 CompCon Conference,* pages 528-537, Februrary 1993.

[4] Souto, P. and Stark, E, "A Distributed Shared Memory Facility for FreeBSD", *In Proceedings of the USENIX 1997 Technical Conference, Anaheim, California,* January 6-10, 1997.

[5] Olson, M., Bostic, K., Seltzer, M. "Berkeley DB", *In Proceedings of USENIX 1999 Annual Technical Conference*, June 1999.

[6] R. Sandberg, "The Sun network file system: Design, implementation and experience", Technical report, Sun Microsystems, 1985.

[7] Fleisch, B and Popek, G., "Mirage: A Coherent Distributed Shared Memory Design", In *Proceedings of the 12th ACM Symposium on Operating Systems Principles,* pages 211-233, December 1989.