Software Exploitation of a Fault-Tolerant Computer with a Large Memory

Frank Eskesen, Michel Hack, Arun Iyengar, Richard P. King, and Nagui Halim
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

The DM/6000 hardware (a prototype, fault-tolerant RS/6000 built at the TJ Watson Research Center) provides fault tolerance and a large, non-volatile main memory. Running a commercial, general-purpose operating system on it, of itself, does nothing to increase software availability. In fact, the time to rebuild the contents of a large memory may decrease availability.

We describe our techniques for hiding most of the main memory, which requires the operating system to access it only by way of services separate from the operating system. This can allow the memory and those access services to achieve much higher availability, which, in turn, increases the availability of the system as a whole. We also performed simulation studies to determine those conditions where this system organization can lead to improved performance for recoverable database applications.

1 Introduction

The DM/6000 [1] is a prototype, fault-tolerant 4-way multiprocessor RS/6000 with a large main memory built at the TJ Watson Research Center. When provided with sufficient backup-battery power, this memory is as reliable as mirrored disks. The intention is to run a commercial operating system with whatever applications a customer might choose to use. Being fault-tolerant, the DM/6000 would offer very high hardware availability, but only moderately higher overall system availability than typical RS/6000 systems, since hardware failures are not the majority of system failures.

Although the memory is non-volatile, it cannot survive the ravages of an application run amok. Thus, when a large amount of memory is installed in a DM/6000, the time to recover from a failure may be dominated by the time to rebuild the contents of memory. For example, the performance of a database system is dependant on the state of its buffer pool. Following a crash, performance is reduced until that buffer pool has been repopulated, which, for a large buffer pool, may take quite a while.

Various workers in the field have explored ways to exploit a non-volatile memory, for example eNVy [19], Sprite [3], and Rio [7, 16]. Their approach is to place the file system cache in non-volatile memory and to modify the operating system and file system to preserve this memory across reboots. Although the level of protection that can be provided to such a cache may be adequate for most purposes [19], this approach does not allow the memory to be completely isolated from failures in the operating system kernel, especially in the paging subsystem, or from failures in add-on extensions to the kernel. At the same time, the memory is completely bound to that one file system running in that one operating system. One possible use of the DM/6000 involves having different operating systems running on its processors at the same time, and the sharing of data in the non-volatile memory seemed a worthy goal.

We therefore decided to hide most of the memory in the DM/6000 from the operating system and the applications running on it. Instead of allowing an operating system to have access to that memory directly, a small server program is loaded in part of that hidden memory, and only that server program accesses any of the hidden memory. The operating system communicates with the server through an intercommunication queue (ICQ) using, for example, a device driver that the rest of the operating system regards as giving access to something like a disk, and the server operates on the hidden memory on behalf of the operating system's users.

Since the server is small and offers a limited set of operations, it can be made much more reliable than a general-purpose operating system. A general-purpose operating system must run run arbitrary kernel extensions (e.g. device drivers), applications, and user programs. With only the server accessing the hidden memory, a failure of the operating system leaves the data in the hidden memory intact. Thus, overall system availability is higher, since less of the state of the system is lost due to software failures.

As noted by Copeland, et al [8], the use of non-volatile main memory is particularly well-suited to applications such as recoverable databases. They typically maintain state in stable storage, allowing a consistent database state to be recovered in the event of a system failure. In order to reduce recovery time after a failure, information is periodically checkpointed to stable storage. Checkpointing to non-volatile main memory is considerably faster than to disk. In addition, a system can reach a steady state after a fail-

ure more quickly if some of the hot pages are checkpointed to non-volatile main memory than if all pages are checkpointed to disk. Although non-volatile main memory can also hurt database paging performance, the advantages often outweigh the disadvantages.

The remainder of this paper is organized as follows. Section 2 is where the general concepts and overall structure of the system are explored. In section 3, an exploitation of this system structure, specifically VDISK, will be introduced to motivate things. In section 4, we present the details of the structure and the implementation of ICQ and the server. In section 5, some performance measures taken of our prototype are given. Section 6 analyzes the effect of non-volatile main memory on recoverable databases. A discussion of work related to ours comes in section 7, which is followed by our concluding remarks, in section 8.

2 General Concepts

The main requirements for the server are that it be reliable and fast. It also has to be maintainable, with the ability to install new versions of the server without affecting the operating system.

The reason for requiring reliability is simply to justify giving the server sole control over a large portion of memory. If the server were no more reliable than, for example, AIX, there would be no reason to take the memory away from AIX. This doesn't just mean that the server should be well written and, therefore, reliable; the server also needs to be sufficiently isolated from AIX that any misbehavior of AIX cannot cause the server to fail or malfunction.

Speed is also an essential part of justifying taking away from the operating system a resource that would otherwise be used to improve its performance. Hidden memory, with only some kind of client/server interface for getting to it, is bound to be slower than regular memory. This was observed by Li and Petersen [12]. They found that segregating a large, slow portion of memory and using the small, fast memory as a cache gave poorer performance than using the slower memory directly. Since we are segregating memory of equal performance, we expect an outright performance loss on many applications. On the other hand, regular memory is not durable. Programs such as recoverable databases that must record data in stable storage therefore benefit from having part of main memory be durable, but only if the server provides access to it fast enough, compared to other forms of durable memory (e.g. duplexed disks), to pay for the loss of the direct use of that memory. As we shall see in Section 6, non-volatile main memory can improve recoverable database performance in certain circumstances.

The overall structure of the system we implemented is illustrated in Figure 1. The only means of communication with the server is by placing requests on a queue, which we will call the Intercom Queue (ICQ). By placing this queue in the part of memory visible to all of AIX, we avoid the suggestion that any part of hidden memory is accessible to code running outside of the server. Processing starts when some program running under AIX, e.g. a device driver, puts a request on the queue. The server determines the particular service for which the request is intended and hands it off. That particular service, using whatever information it maintains in the stable memory, processes the request and formulates its response.

There is no specific mechanism for the returning of responses; this is left up to the individual services, whose requirements for interaction with their clients may vary considerably. As an example, however, suppose that some bit in the request data area is designated by a particular service as the *processing complete* bit. Before sending a request, a client would set this to zero. Another word in the request is the location of a response area. The service can build its response, placing it in the response area, and then set the *processing complete* bit to one. The client, meanwhile, can poll the state of this bit, or go off and do other work. A software-interrupt driven scenario would be similar, with the addition of an agreed upon interrupt level.

3 Examples of Possible Services

Most commercial applications tend to use only one kind of stable storage, namely magnetic disks. Therefore, rather than try to modify an application to best exploit a new form of stable storage, we have instead provided applications with very fast versions of stable storage whose interfaces they are already familiar with. We call these services VDISK, which is a stable-storage RAM disk, and PCACHE, which provides stable-storage caching for a disk.

The VDISK service is like a RAM disk, except that it is more isolated from the applications that use it. There is a VDISK device driver installed on AIX that, instead of allocating memory in, say, the kernel space, enqueues a request asking the VDISK service to allocate stable memory. Similarly, read and write requests, instead of resulting in the copying of data by the device driver, result in the construction of corresponding requests to the VDISK service.

PCACHE is used when the data doesn't fit in durable memory. Any disk device can be provided with a main-memory cache. For example, a pseudo-device driver can be installed on AIX that can be configured to read and write to a cache and, when there is a cache miss, pass the request on to the real device driver. There are two differences for PCACHE. First, the cache is only accessible via ICQ requests to the PCACHE service. This may degrade performance, but leads to the other difference: on recovery from a failure, PCACHE service still knows what is in the cache, which benefits performance after a crash.

One particularly interesting application is fast-reboot of the operating system. It is possible to structure the operating system's initialization sequence so that the complete system state is in real memory at the point where most R/W file systems are mounted, using a VDISK for the boot file system. This state (including the boot R/W file system) can then be described to a FASTBOOT service, which takes a consistent snapshot of the relevant main memory and VDISK, and saves it in the durable memory. Once

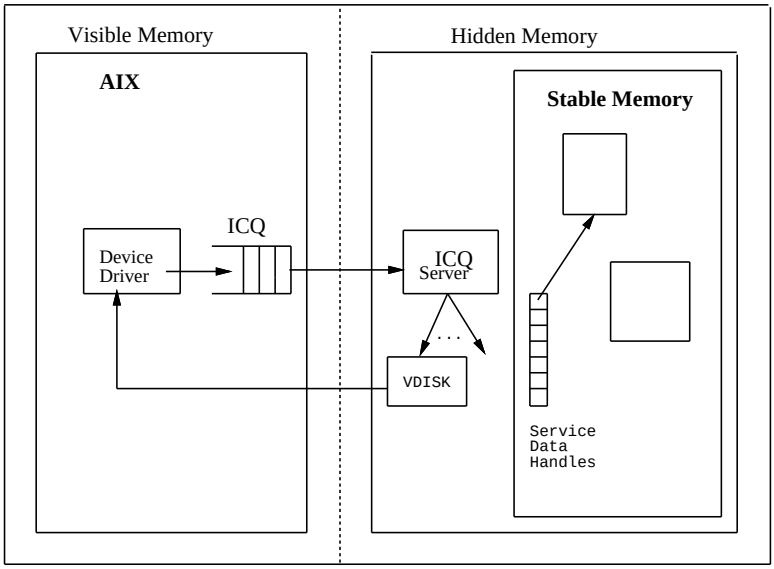

Figure 1: Overall Structure of DM/6000 Software.

this FASTBOOT snapshot exists, the operating system can be rebooted in about 1 second from the time the FASTBOOT server has received a *reboot* request. Since our device drivers are all virtual, in the sense that they communicate with the server and not directly with a physical I/O device, this allows a complete reboot in under 3 seconds.

4 ICQ and the Server

The main problems we faced in providing services outside of AIX were: sharing physical memory; sharing the processor(s); communication between AIX and the server; server installation and restart; and service installation and restart. The last two problem areas arise from the recognition that our intention to make the ICQ server and services free of bugs will not necessarily be fulfilled. Further, even if it is, there will, undoubtedly, be additional services invented after some systems are already in operation. We therefore want to be able to replace the server, and/or services, on a running system without performing a shutdown of AIX.

4.1 Sharing Physical Memory

We want some of the main memory of the DM/6000 prototype to be accessible only to the ICQ server, but the prototype does not support hardware fencing of memory. Thus, there is in principle nothing to prevent the operating system from trespassing onto the ICQ server's storage. In practice, operating systems avoid accessing non-existent real memory, because this could lead to unrecoverable machine-check interruptions: they expect the ROS boot code that loads the operating system to provide a map of available real memory.

Since we are providing the equivalent of the ROS boot code ourselves, we are free to have that code build a map of memory indicating that the vast majority of the installed memory simply does not exist. This is not an ideal solution, but is adequate, given that AIX operates in real mode only until the real addresses of the good memory pages have been loaded into the appropriate tables, when virtual address translation is turned on once and for all (except for a very brief initial sequence in first-level interruption handlers). Since no real addresses are ever generated, there is little danger of an erroneous access to hidden memory.

4.2 Sharing Processors

We could dedicate some of the DM/6000's processors to the ICQ server, leaving the rest to AIX, but possible workload changes make this inefficient. Further, an entire processor is a rather crude granule. And, finally, we wanted to be able to test our system modifications using standard, uniprocessor RS/6000s. Therefore, we needed a way to switch a single processor between running AIX and our server.

This requires a hole in the wall between visible and hidden memory, a call stub in the kernel that transfers control to code in hidden memory. A hardware call gate could be used; it would be unspoofable, though slow. In practice, sufficient protection derives from running the server without address translation, in unmapped storage. The call stub is located in an area of memory that is mapped virtual=real so that translation mode switching makes sense. The real address of the server code is supplied in the operating system's boot data.

When a request is put on the ICQ, the processor making that request is free to spin, waiting for some

other processor to run the server code. Alternatively, it can switch to running the server code itself. That processor may find its own request still on the queue, or some other processor's, or none at all. This provides completely automatic and instantaneous balancing of processing resources between AIX and the server.

We could treat requests as asynchronous. However, a transfer of 4KB using the DMA hardware requires about one thousand cycles, and this is the largest component of processing the request. The cost of a task switch is around a thousand cycles, even ignoring the cost of preparing for an I/O operation to go asynchronous and the performance effects of reloading the cache after the switch, which can be tens of thousands of cycles. Like Tucker [18], therefore, we concluded that it is faster to treat most requests as synchronous.

4.3 Communication Between AIX and the Server

The ICQ provides the communication path between AIX applications and the server. It is composed of a header record and a set of fixed-sized queue entries (request slots). The header record has offsets, relative to the beginning of the page, to the FIFO queue and the *free list.* (By using offsets, the queue makes sense to both AIX, in virtual mode, and the server, in real mode.) The page holds a fixed number of request slots, initially all on the free list. Allocation and deallocation of queue entries result in moving a slot from or to the free list, using lock-free pointer updates. Indeed, all accesses to the ICQ are made using lock-free protocols. All of this makes it possible for AIX and the server to share the ICQ as efficiently as possible.

Since AIX has access to the ICQ, AIX must be responsible for allocating and maintaining it. An object in virtual memory that spans a page boundary will, quite likely, occupy noncontiguous real pages. To avoid this inconvenience to the server, which operates in real mode, the ICQ must fit within a single page. After the virtual page to contain the ICQ has been initialized and pinned into real memory, the word in visible memory that indicates the location of the ICQ is set to the real address of the ICQ.

4.4 Enqueue and Dequeue

We want operations on the ICQ to be very fast, very simple, but also to be completely thread safe and multiprocessing safe. So we use lock-free protocols built on the load-and-reserve and store-conditionally instructions of the PowerPC processors [15]. (For an example of a shared-queue algorithm instead using compare-and-swap, see [17].) These allow multiple threads, whether they run on the same processor or not, to access the queue without interference with each other.

To enqueue a queue entry, first load-and-reserve the word in the queue header that contains the offset of the first entry in the queue (the head offset). Then set the new entry's *next* offset to that value. Finally, store the offset of the new entry in the head offset, on the condition that one's reservation on that word is still present. If it isn't present, start over.

This, of course, gives a queue organized in LIFO fashion. Giving the queue a FIFO flavor is done in the dequeueing process. To dequeue an entry, start by loading-and-reserving the head offset in the queue header. Then steal the whole queue, by storing, conditionally, an End-of-Queue indicator (zero) in place of that value. (As before, if the reservation isn't held anymore, start over.) With the entire queue in hand, the server is free to traverse the queue (if necessary) and to take the oldest request off of the queue. If the queue is not empty, put back what is left.

This putting back requires some care. If something else has been put on the official queue in the meantime, those entries must be merged with the set of entries stolen by this dequeuer. To do that, load-and-reserve the head offset word. If it is zero, there are no new entries, so just store, conditionally, in the head offset the offset to the first entry of the list being returned. Otherwise, store, conditionally, a zero and use the value loaded as the start of a new queue to be merged with the old one, then try again. Either way, a failure to store means try again from the start of this section.

An optimization is performed to avoid running to the end of the LIFO chain every time the oldest entry on the chain is wanted: as a side-effect of traversing the LIFO queue to find the oldest entry, the queue pointers are reversed, so that the queue is in FIFO order by the time the server is ready to re-anchor it. FIFO elements are distinguished from LIFO elements by having a non-null *tail* pointer. The tail of the FIFO queue is the head of the original LIFO queue, i.e. the original anchor value, so the entire operation can be performed in a single pass. The tail pointer (offset, really) allows the tail to be located quickly for insertions at the end. An example of this can be seen in Figure 2. While new entries are being enqueued, there will be some LIFO entries followed by some FIFO entries. The next time a server performs a dequeue operation, it need only scan the new LIFO portion, which gets appended (in reverse, i.e. FIFO, order) to the tail of the FIFO portion.

4.5 Using ICQ to Make a Request of the Server

The entire ICQ is contained in a single 4KB page. Individual queue entries are therefore of limited size, and are used only to identify which service a request is intended for, and the real address of a page-aligned collection of input parameters and space for output flags and data. It is the responsibility of, for example, the VDISK device driver to learn the real addresses of any buffers the VDISK service must use. The queue entry doesn't even have the full name of the service; that goes in the input parameter area. Instead, only the handle for the service goes in the queue entry. When the server dequeues the request, the handle is used to look up the service unless the check values stored with the handle don't match. In that case, the full name is used to search the table of services. The service code is then free to do whatever it wants to with the request data. For example, the VDISK service operates synchronously, setting a return code and

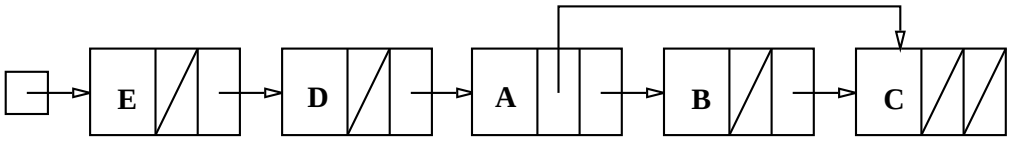

Figure 2: ICQ With LIFO and FIFO Elements.

a completion flag in the output area before returning. Asynchronous services may hold onto input data until they signal completion. These are service-specific aspects of the protocol, and of no concern to the server or the ICQ code.

If a device driver runs out of ICQ request blocks (free list empty), it treats this like a busy device: some I/O is in progress. The server should get time to run (on a uniprocessor), and other threads or processors should get time to complete their requests. Perhaps the driver's own earlier requests need to be completed first: to some extent, a device driver manages ICQ slots like channel paths to a real device.

4.6 Installation and Restart

The server needs to be maintainable without disrupting the operating system. To allow for replacement of the server code, this code must reside in stable memory. Each time the server code starts up, it checks for the presence of a valid version of its persistent data: the list of available services and the stable memory in use by any of the services. This persistent data is used by the server to continue processing as though no change of server had occurred. A change in server has no effect on the ICQ as long as that change takes place while the server is quiescent.

Concurrent maintainability applies to individual services as well. They are the virtual equivalent of hot-swappable drives and hot-swappable control units. Each service is registered in the server at run time. Registration information includes the full name of the service and the location of that services request handling code. The server provides a place in its own persistent data where each service can record one word of information. This is used to point off to whatever persistent data that particular service needs. The contents of this word are passed back to the service every time it is given a request to process.

Restarting a service can therefore be made as invisible to the user as can the restarting of the entire server. As long as the service records all persistent data in stable storage before returning to the server, that service can be restarted, or replaced, whenever it is convenient.

4.7 Multiple Host Operating Systems

All of the above has been discussed in terms of a single host operating system, a single ICQ, and a single server. This has been done solely to simplify the exposition. There can be, in a single DM/6000, different host operating systems using different processors, multiple ICQs, and multiple servers of those queues. Multiple host operating systems might be wanted in the same DM/6000 for functional separation of environments (e.g. development and production work), to provide a hot standby to a critical system, or even to provide multiple operating system platforms (say, AIX and Windows/NT). Sharing of data among these hosts is certainly possible. For example, requests from multiple hosts could target the same VDISKs. Each host could just as well be assigned its own portion of the hidden memory and be the sole user of it, except, perhaps, in the case of a failure of that host operating system. One might then arrange, for example, to have a standby running on another of these host operating systems take over for the failed system and to assume ownership of the data previously managed by the first.

The access protocols for the ICQ are such that multiple operating systems could even share a single ICQ safely. However, this is not really practical, since it exposes each of the host operating systems to possible contamination due to failure of the other. Furthermore, it requires them to agree on the allocation and initialization of the ICQ. It would be better to let each operating system manage its own ICQ, which can be connected separately to the same server, or even to a different one.

5 Some Performance Results

We performed a test reading 64Mbytes from a raw device using various block sizes – a VDISK on the DM/6000, and a real disk on an RS/6000-250. Elapsed time are shown in seconds:

Block size	Vdisk	Hdisk
4K	13.4	25.3
64K	1.6	26.2
1M	1.0	26.6

Both machines used a PowerPC 601. The DM/6000 prototype was clocked at a conservative 40MHz, the RS/6000 at 66MHz. Block size has a dramatic effect on the VDISK timing, because at small block sizes the request handling time dominates the transfer time. The throughput exceeds 60MB per second at large block sizes, over 25 times faster than real I/O to the hard disk. (The actual requests issued were " time dd if=/dev/$disk of=/dev/null bs=$blocksize count=$blocks " such that $blocksize x $blocks = 64M.)

At first glance, these numbers seem to indicate that the DMA hardware is not well used; it can transfer data at a rate of about 300MB per second. However, it cannot transfer data directly from symbol-plane memory to symbol-plane memory, only between

L3 cache and symbol-plane memory [1]. These transfers are, therefore, two-stage transfers by way of the L3 cache. Thus, copying data at 60MB per second requires running the DMA at 120MB per second. Since this load is produced by just one of the four processors, which ran the full code path for file I/O synchronously with the DMA transfer, this starts to look more respectable.

6 Database Recovery in the Presence of Durable Memory

Databases utilize stable storage for data pages and transaction logs. In the event of a failure, all pages contained in conventional memory could be lost. It is then necessary to recover a consistent state of the database from the data pages and transaction log stored in stable storage. There are several ways in which a database could benefit from durable memory:

- In the event of a failure, the database would have to be recovered from data stored in stable storage. Backing up hot pages in durable memory instead of disk can reduce the I/O cycles required for recovery. Throughout this section, the term *disk* refers to traditional secondary storage.

- Durable memory can reduce the time to bring the database to a warm state after a failure.

- In order to reduce recovery time in the event of a failure, pages in main memory are often periodically checkpointed by writing them out to stable storage. Pages can be checkpointed more quickly to durable memory than to disk.

- Durable memory can speed up transaction logging. Many databases such as IBM's DB2 use a *write-ahead logging protocol* in which the transaction log must be written to stable storage before the transaction commits and before any page updated by the transaction is written to stable memory. Writing transaction logs to disk can slow down transactions considerably. In order to improve performance, updates to the transaction logs in stable storage can be batched. However, this cannot be done for transactions which must commit immediately. Durable memory can improve the performance of transaction logging. The idea is to reserve space in durable memory for transaction logs. A single transaction record could then be stored in durable memory quickly. Transaction logs could be moved from durable memory to disk opportunistically when such a data transfer would not affect performance. This will prevent durable memory from becoming full.

A drawback to durable memory is that it reduces the size of the buffer pool and can thus hurt performance by causing more paging. The remainder of this section examines the effect of durable memory on I/O resulting from paging, checkpointing, recovery, and bringing the database to a warm state. We distinguish between I/O resulting from three different sources:

1. *Paging I/O:* I/O resulting from pages being brought into the buffer pool from stable storage during normal database processing (i. e. not recovery).

2. *Checkpoint I/O:* I/O resulting from periodically checkpointing database pages in the buffer pool to stable storage in order to reduce database recovery time.

3. *Recovery I/O:* I/O which results from database recovery after a failure.

A DM/6000 which has enough main memory to store the entire database in main memory with room to spare will always benefit by having durable memory. For example, suppose that a DM/6000 has a maximum of 512 mbytes of main memory which can be used for storing database pages without significantly hurting performance. For a database containing 400 mbytes, memory could be partitioned so that all of the database is stored in conventional memory and 112 pages of durable memory are used to back up the hottest pages. The use of durable memory will improve performance of the system over one without durable memory.

Now consider a situation where the database is too large to fit entirely within main memory. The decision of whether or not to partition main memory into durable and conventional memory is less clear. If no durable memory is used for backing up database pages, the buffer pool size is maximized and I/O resulting from paging is minimized. If a fraction of main memory is used for durable memory, checkpointing and recovery I/O can be reduced. In the situations we encountered, durable memory resulted in better steady state performance when short recovery times were desirable while no durable memory resulted in better steady state performance when longer recovery times were tolerable.

6.1 Methodology

We simulated databases in which dirty pages are periodically checkpointed to disk in order to bound recovery time. Databases are checkpointed synchronously after every c cycles where c remains constant throughout each simulation. During each checkpoint, all pages which have been dirty since before the previous checkpoint are backed up in durable memory.

The simulator models a database running on a 100 Mhz machine with 512 mbytes of main memory which can be used for holding data pages. The bandwidth between main memory and disk was 12.5 mbytes/second. The bandwidth between durable and conventional memory was 100 Mbytes/second. Pages are 4096 bytes.

We used a request distribution in which 80% of the transactions were read requests. Hot pages constituted 20% of all pages. 80% of all requests were distributed uniformly to hot pages, while the remaining 20% were distributed uniformly to cold pages. The performance numbers only consider machine cycles consumed by I/O operations and do not take into account other overheads.

During each checkpoint, we attempted to back up as many pages as possible to durable memory. Pages

were only checkpointed to disk after durable memory became full. At the time of each checkpoint, the number of candidates for durable memory was determined. A page is a candidate for durable memory if a copy of the page already exists in durable memory or the page is just about to be checkpointed. When durable memory is not large enough to contain all candidates, pages are given priority based on a formula which considers the number of hits since the last checkpoint. Priority increases with the number of hits since the last checkpoint. Priority is also given to pages in durable memory which don't have to be moved at a checkpoint. For example, suppose that p_1 and p_2 have both been hit seven times since the last checkpoint. Page p_1 is dirty and needs to be backed up to stable storage. Page p_2 does not have to be checkpointed. However, the backup copy for p_2 is contained in stable storage. If we just look at the number of hits since the last checkpoint, both pages would have equal priority. However, leaving the backup copy for p_2 in place and backing up p_1 to disk incurs less I/O than moving the backup copy for p_2 to disk and checkpointing p_1 to durable memory. The system thus leaves the backup copy for p_2 in durable memory during the checkpoint.

The database manages conventional memory using LRU. Two strategies were tested for sending cold pages to stable storage:

1. Always send cold pages to disk. The motivation is to leave space in durable memory for checkpointing hot pages. This strategy worked best when durable memory was small.

2. Send cold pages to durable memory whenever durable memory is not full. This strategy worked best when durable memory was plentiful.

The performance differences for the two strategies was not significant. The statistics presented in this paper use the first approach.

We present the performance statistics obtained from four different configurations:

1. A database containing 256 mbytes, no durable memory. All pages of the database fit in conventional memory.

2. A database containing 256 mbytes for which main memory is partitioned into 256 mbytes of conventional memory and 256 mbytes of durable memory.

3. A database containing 1.024 gbytes, no durable memory. Conventional memory contains space for 512 mbytes of the database.

4. A database containing 1.024 gbytes for which main memory is partitioned into 307.2 mbytes of conventional memory and 204.8 mbytes of durable memory. Partitioning was chosen to make durable memory size equal to the hot page size.

Unless otherwise noted, the performance statistics were taken after the database had reached a steady state.

Performance for a single configuration was varied by varying the checkpoint interval. More frequent checkpointing results in faster recovery after a failure. However, checkpointing consumes significant I/O bandwidth and has an adverse effect on steady state performance. There is thus a trade-off between steady state performance and recovery time. A system optimized for steady state performance would checkpoint infrequently. A system optimized for fast recovery time would checkpoint frequently.

6.2 Results and Discussion

Figures 3 and 4 plot recovery time as a function of the average request time after the database is in a steady state. Recovery time is expressed as the number of megacycles for restoring the database to the most recent consistent state which can be reconstructed after a failure. Average request times were calculated by dividing the cycles for both paging and checkpointing over an interval of several requests which included multiple checkpoints by the total number of requests in the interval. Measurements for average request times were taken after the database was in a steady state.

Performance for a given memory configuration and database was varied by varying the checkpointing interval. Recovery time varies from near zero to arbitrarily large times with or without durable memory depending upon the time between checkpoints. We use the following criterion for comparing the performance of two memory systems m_1 and m_2 on the same database:

> Let max_recov_time be the maximum number of recovery cycles which are tolerable. Let c_1 be the largest number of cycles between checkpoints which allows recovery on m_1 using a maximum of x I/O cycles. Let c_2 be the analogous quantity for m_2. System m_1 outperforms m_2 if and only if the the average request time in a steady state (which is affected by both paging and checkpointing) of m_1 using a checkpoint interval of c_1 is less than that of m_2 using a checkpoint interval of c_2.

If main memory is large enough to store all database pages with room to spare, performance can always be improved by partitioning memory and reserving some space for durable memory. This is the case for the 256 mbyte database, and the performance advantage resulting from durable memory is conveyed by the graph in Figure 3. The configuration with durable memory results in better performance than the configuration without durable memory for all values of max_recov_time.

In Figure 4, the database is too large to fit in main memory. The use of durable memory results in better performance if $max\_recov\_time < 1650$ megacycles (16.5 seconds assuming a 100 Mhz clock rate). Not creating a durable memory partition results in better performance if $max\_recov\_time > 1650$ megacycles. The reason for the crossover point is that

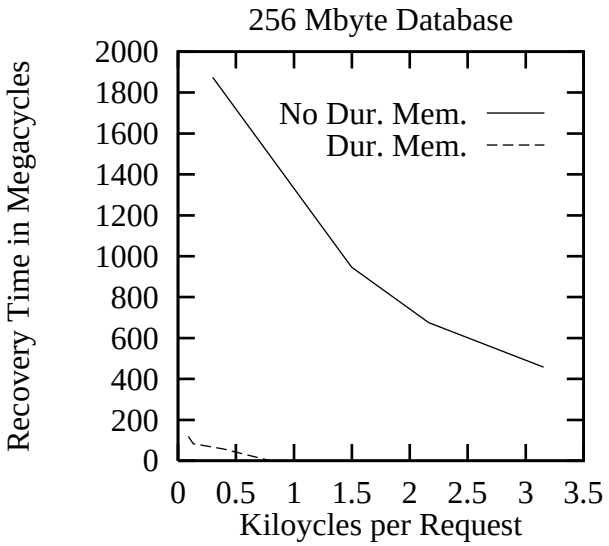

Figure 3: Recovery Time versus Steady-State Request Times with Smaller Database.

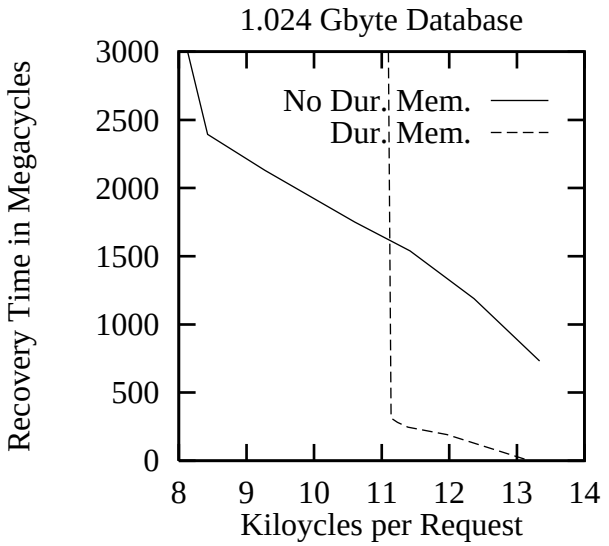

Figure 4: Recovery Time versus Steady-State Request Times with Larger Database.

durable memory reduces checkpointing cycles but increases paging by reducing the buffer pool size. Checkpointing cycles dominate the average request time if checkpointing is done frequently in order to provide fast recovery. If fast recovery is not crucial, however, checkpointing can be done infrequently. In this situation, the average request time is dominated by paging.

If the limiting factor is the average request time, a similar analysis applies. The lowest average request time which can be achieved without durable memory on the 1.024 Gbyte database is 8150 cycles and occurs when there is no checkpointing. The lowest average request time with durable memory is 11100 cycles and also occurs when there is no checkpointing. If the average request time must be less than 8150 cycles, 512 mbytes are not sufficient to achieve this level of performance with or without durable memory. If an average request time between 8150 and 11150 I/O cycles must be achieved, then the memory configuration without durable memory should be used. If an average request time of more than 11150 I/O cycles per request is acceptable, then the durable memory configuration should be used in order to minimize recovery time.

Another advantage of durable memory is that it allows databases to be brought to a warm state quickly. For the 256 mbyte database, the configuration without durable memory required 2.8 million requests consuming 2.10 gigacycles for paging I/O to bring the database to a steady state from a state in which no database pages are stored in conventional memory. If all pages are backed up in durable memory, a steady state is reached after 2.8 million requests consuming 262 megacycles for paging I/O. In both cases, a steady state was reached when all database pages were loaded into conventional memory.

For the 1.024 gbyte database, the configuration without durable memory resulted in performance which was substantially inferior to steady state performance from the time conventional memory was empty until 150 kilorequests consuming 2.4 gigacycles for paging I/O had been satisfied. At this point, memory was 58% full. It took a total of 500 kilorequests consuming 4.3 gigacycles for memory to become 100% full. Between the time memory was 58% and 100% full, performance was slightly superior to steady state performance. The reason for this behavior is that a memory system which is not full can read in a page from disk without sending a cold page to disk. Therefore, twice as many pages can be accessed from disk per unit of time compared to a memory system which is full. After memory became 100% full, steady state performance was reached.

The durable memory configuration with the 1.024 gbyte database did not exhibit cold start performance problems when all hot pages were backed up in durable memory. Surprisingly, performance was superior to steady state performance during the time conventional memory was filling up. The reason for this behavior is the same reason why the nondurable memory configuration also results in superior performance between the time memory is 58% and 100% full. However, durable memory provides a significant advantage because hot pages can be read in from durable memory much more quickly than from disk. Therefore, superior performance commences immediately instead of after memory was 58% full. Memory became 100% full after 175 kilorequests consuming 1.13 gigacycles for paging I/O had been satisfied. At this point, steady state performance was reached.

7 Related Work

Other well-known fault-tolerant architectures use redundant storage devices or processors, or some combination of both as in IBM's High Availability Cluster Multi-Processing (HACMP) [2] and Tandem's Non-Stop architecture [4]. In these systems, there is more than one copy of the operating system and applications running on more than one system, with one copy acting as a standby in case the other fails. The hypervisor approach taken by Bressoud and Schneider [6] is similar, but uses virtual machines within a single system. We exploit the DM/6000's hardware resiliency by treating the hardware as fault-free and limiting our task to operating system isolation and fast failure recovery.

We isolate ourselves from operating system software failures by hiding a portion of memory. We place our compact, limited function kernel in the hidden memory. Because our kernel is compact and has extremely limited function, it is possible to be shipped essentially error free. In itself, this is not a new or novel approach. For example, the hypervisor in CP-67 [14] and the later VM/370 [10] also limits the amount of kernel function. Our kernel design goes much further and is similar to that used in the Coupling Facility for the S/390 Parallel Sysplex [5], in that our kernel runs in real mode and does not accept interrupts, instead polling for work.

Unlike the Coupling Facility, however, we share main storage with a standard host operating system. Our prototype design prevents the host operating system from modifying our protected, hidden memory by indicating to the operating system that the storage does not exist. The operating system simply does not use hidden storage, and we are isolated from almost all operating system failures. Our design does not use hardware protection mechanisms such as those used by VM/370 since they were not available on the prototype hardware. This additional protection would eliminate the possibility of an operating system failure in boot or address translation service damaging durable (hidden) memory.

This sharing of main memory was also examined by Li and Petersen [12]. They observed that using a slower, extended memory as a cache (as a stage of memory between fast main memory and disk) gave poorer overall system performance than accessing it directly (as the bulk of a much larger but slower main memory) for applications with large data structures. However, they did not consider the possibility of having the cache survive system crashes, how that might be done, or the consequences of such survival to system design or performance.

IBM's Transaction Processing Facility (TPF) [13] does not recover from faults but instead restarts as quickly as possible, attempting to reuse data surviving in the buffer areas. Our architecture requires a buffer reload when AIX fails, but data are copied from durable memory.

The VDISK service has goals that are similar to those for the Phoenix file system [9]. A Phoenix logical disk resides in memory but is kept safe from certain kinds of failures through a copy-on-write strategy that leaves intact a timestamped version of the logical disk. However, the data is not safe from intrusion from other parts of the kernel, nor extensions to it. In contrast, the contents and structure of a VDISK are safe from all failures other than those of the ICQ server and those few services it runs, and no operating system failure can interrupt a VDISK operation.

The intent of the work done by Chen, et al, for Rio [7] is similar to that behind PCACHE. They put their file cache in a portion of real memory that will not be overwritten when the system crashes and restarts. To protect the data from being overwritten, the write-permission bits in the page table are turned off except while data is being written to the cache. Their performance is better than ICQ, at the expense of slower recovery. With DM/6000, the cache that survives a crash is outside the file system, so file system recovery is not affected. With Rio, a recovery step is needed during reboot, to reconcile the file system on disk with the surviving cache. The use of Rio is also restricted to just that of a cache for files used by a single operating system, rather than a general-purpose server that can be shared by multiple operating system images. Further, since Rio is not a device driver, it cannot be used to store a fast-boot image of the operating system. Rather, Rio depends on the operating system to be rebooted so that the operating system can help bring Rio back up.

For our prototype we assume, like Rio, that the virtual memory manager will not improperly assign real storage addresses. Rio's approach is probably more forgiving of such a failure. Our view is that this exposure should be corrected in hardware inaccessible to the operating system, as in IBM's System/390 PR/SM LPAR [11] product.

8 Conclusions

Our approach to running a general-purpose operating system on a fault-tolerant computer has been to hide the computer's resources from the operating system. Instead, these resources are managed by a limited-functionality, highly reliable server that communicates with the operating system only through the passing of messages on a queue. By hiding the memory resources of the computer from the operating system, the contents of that memory becomes as reliable as the hardware that contains it.

Our design of the communication queue provides high performance and safety. Through the use of checks in the ICQ data structures and a lock-free access protocol, the server is protected from the operating system. There is never a moment when the queue is incorrectly structured. Thus, halting the operating system cannot block the server. And, since all of the structures of the ICQ are checked during access, damage to the structure is detected by the server. Repair of the ICQ is also detected by the server, and once the operating system has restarted, it can immediately start to use the server. At the same time, the server program is decoupled from the rest of the DM/6000 by keeping its persistent state in memory separate from its own volatile storage areas. If the server program needs to be repaired or enhanced, this can be done

at any time, without loss of information and without any apparent interruption of service to the operating system.

The simulations we performed established when this approach actually improves performance. While non-volatile main memory can reduce performance on certain applications by reducing the amount of main memory available to programs, it can also improve performance on applications such as recoverable databases which must periodically store information in stable storage. We found that non-volatile main memory results in improved performance when databases are small, or when databases are large but small recovery times are essential.

One direction for research that we have not yet mentioned is in the area of more intelligent services that move the data farther from the application, but do more per request to the service. One example would be a service that supported a main memory database in stable storage. By providing high-level requests, such as insertion of a record in a table with multiple indexes, the cost of making a request to the server is amortized over more instructions. Determining the right trade-offs between service functionality, performance, and reliability is a problem requiring further study.

Acknowledgments

We gratefully acknowledge the contributions of Basil Smith to the DM/6000 hardware which made the work described in this paper possible.

References

[1] M. Abbott et al. Durable Memory RS/6000 System Design. In *Digest of the 24th International Symposium on Fault-Tolerant Computing Systems*, pages 414–423, June 1994.

[2] G. Ahrens et al. Evaluating HACMP/6000: A Clustering Solution for High Availability Distributed Systems. In *IEEE Conference on Fault-Tolerant Parallel and Distributed Systems*, pages 2–9, June 1994.

[3] M. Baker et al. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of ASPLOS V*, pages 10–22, September 1992.

[4] J. Bartlett. A NonStop Kernel. In *Eighth Symposium on Operating System Principles*, pages 22–29, December 1981.

[5] M. Bradley. Understanding the S/390 Parallel Sysplex: a Technical Introduction. In *CMG94 Proceedings*, pages 1139–1148, December 1994.

[6] T. Bressoud and F. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Trans. Comput. Syst.*, pages 80–107, February 1996.

[7] P. M. Chen and W. T. Ng. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of ASPLOS VII*, pages 74–83, September 1996.

[8] G. Copeland et al. The Case for Safe RAM. In *Proceedings of the Fifteenth VLDB*, August 1989.

[9] J. Gait. Phoenix: A Safe In-Memory File System. *Communications of the ACM*, 9(3):199–218, 1970.

[10] IBM. *GC20-1801 IBM Virtual Machine Facility/370 Planning Guide*, 1972.

[11] IBM. *ZZ81-0334 Using PR/SM LPAR*, 1993.

[12] K. Li and K. Petersen. Evaluation of Memory System Extensions. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 84–93, May 1991.

[13] R. J. Martin. *Transaction Processing Facility: a guide for application programmers.* Yourdon Press, 1990.

[14] P. Meyer and L. Seawright. A virtual machine time-sharing system. *IBM Syst. J.*, 9(3):199–218, 1970.

[15] Motorola. *PowerPC 601 User's Manual*, 1993.

[16] W. T. Ng and P. M. Chen. Integrating Reliable Memory in Databases. In *Proceedings of the 23rd VLDB*, 1997.

[17] J. Stone. A Simple and Correct Shared-Queue Algorithm Using Compare-and-Swap. Technical Report RC 15675, IBM Research Division, Yorktown Heights, NY, April 1990.

[18] S. Tucker. The IBM 3090 system: An overview. *IBM Syst. J.*, 25(1):4–19, 1986.

[19] M. Wu and W. Zwaenepoel. eNVy: A Non-Volatile Main Memory Storage System. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems WWOS-III*, pages 116–118, October 1993.