The VLDB Journal (1998) 7: 194–204

The VLDB Journal
© Springer-Verlag 1998

Integrating reliable memory in databases

Wee Teck Ng, Peter M. Chen

Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, 1301 Beal Avenue, University of Michigan, Ann Arbor, MI 48109-2122, USA

Edited by M. Jarke. Received January 1, 1998 / Accepted June 20, 1998

Abstract. Recent results in the Rio project at the University of Michigan show that it is possible to create an area of main memory that is as safe as disk from operating system crashes. This paper explores how to integrate the reliable memory provided by the Rio file cache into a database system. Prior studies have analyzed the performance benefits of reliable memory; we focus instead on how different designs affect reliability. We propose three designs for integrating reliable memory into databases: non-persistent database buffer cache, persistent database buffer cache, and persistent database buffer cache with protection. Non-persistent buffer caches use an I/O interface to reliable memory and require the fewest modifications to existing databases. However, they waste memory capacity and bandwidth due to double buffering. Persistent buffer caches use a memory interface to reliable memory by mapping it into the database address space. This places reliable memory under complete database control and eliminates double buffering, but it may expose the buffer cache to database errors. Our third design reduces this exposure by write protecting the buffer pages. Extensive fault tests show that mapping reliable memory into the database address space does not significantly hurt reliability. This is because wild stores rarely touch dirty, committed pages written by previous transactions. As a result, we believe that databases should use a memory interface to reliable memory.

Key words: Reliability – Recovery – Main memory database system (MMDB)

1 Introduction

Current database systems store data on disk and in memory. Disks are considered stable storage; they are assumed to survive system crashes and power outages. On the other hand, database systems traditionally assume that the contents of main memory (RAM) are lost whenever the system crashes [Gray81, Haerder83], an assumption that appears to have its roots in the switch from core memories to volatile DRAM [Gray78].

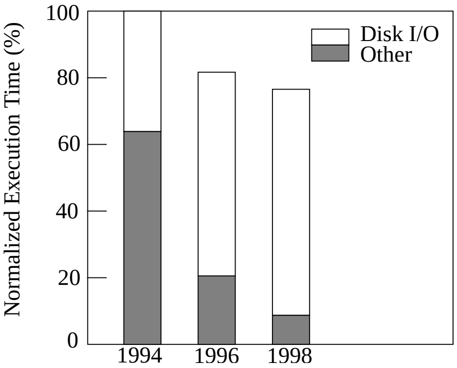


Fig. 1. Database Execution time profile on next-generation machines. This figure is taken from [Rosenblum95] and shows execution time of a database workload (Sybase SQL server running the TPC-B benchmark) on three machine models. The time is normalized to the speed of the 1994 model. Without reliable memory, disk I/Os will be the first-order bottleneck to higher performance

Memory is considered unreliable for two reasons: power outages and software crashes. Memory's vulnerability to power outages is straightforward to understand and fix. A $100 uninterruptible power supply can keep a system running long enough to dump memory to disk in the event of a power outage [APC96], or one can use non-volatile memory such as Flash RAM [Wu94]. Critical database installations often use uninterruptible power supplies to protect against power failure. Memory's vulnerability to software crashes is more challenging to fix; thus database systems assume the contents of buffers in memory are lost when either the operating system or database system crashes.

The assumption that memory is unreliable hurts database performance and complicates database system design. Systems use strategies such as logging and group commit to minimize disk I/O, but these strategies complicate locking and recovery and do not improve commit response time. Even with logging and group commit, disk bandwidth is a significant and growing bottleneck to high performance (Fig. 1) [Rosenblum95].

Recent results in the Rio project at the University of Michigan show that it is possible to create memory that is as safe as disk from operating system crashes [Chen96].

This paper explores how to integrate the reliable memory provided by the Rio file cache into a database system. In particular, we examine how different software designs expose the memory to database crashes. We evaluate the reliability of three designs.

- **I/O interface (non-persistent database buffer cache)**. Hide the reliable memory under the file system interface and use file system operations (read/write) to move data to the reliable memory. Databases see the standard I/O interface to stable storage, and hence this design should be as safe as a standard database from database crashes. This design is attractive, as it requires no database changes.
- **Memory interface (persistent database buffer cache)**. Map the reliable memory into the address space of the database system, and allocate the database's buffer cache (or log) from this region. This increases the exposure of reliable memory to database crashes, but eliminates the double buffering experienced by the I/O interface design. We can also achieve better performance than the first design, as the database manages the buffer cache directly.
- **Memory interface with protection (persistent, protected database buffer cache)**. Map the reliable memory into the address space of the database system, and use virtual memory protection at the user level to protect reliable memory from database crashes. This design offers more protection than the first two designs, but manipulating protections may slow performance.

Our main conclusion is that mapping reliable memory into the database address space does not significantly decrease reliability. To our knowledge, this is the first work that measures how often data is corrupted by database crashes.

2 Benefits of reliable memory

This section summarizes the main benefits of reliable memory, which have been discussed and quantified by many studies [Copeland89, Bhide93, Lowell97].

Reliable memory can be used to store the log (or the tail of the log). Keeping the log in reliable memory removes all synchronous disk writes from the critical path of a transaction [Copeland89]. This decreases transaction commit time and can help to reduce lock contention and increase concurrency [DeWitt84]. Storing the log in reliable memory can also decrease disk bandwidth due to logging, because many log records can be removed before being written to the log disk [DeWitt84, Hagmann86]. For example, undo records may be removed if they belong to transactions that have committed, and redo records may be removed if they belong to transactions that have aborted. Finally, critical information may be stored in the stable memory to help improve recovery time. For example, storing an appropriate pointer in reliable memory can save scanning the log to find the last checkpoint [DeWitt84].

A more aggressive use of reliable memory is to store the database buffer cache, or to store an entire main-memory database [GM92, Bohannon97]. This makes all buffer cache changes permanent without writing to disk. Like the force-at-commit policy, this eliminates the need for checkpoints and a redo log in recovering from system crashes (partial redo) [Haerder83, Akyurek95]. This simplifies and accelerates recovery, because there is no need to redo incomplete operations; each commit is a transaction-consistent checkpoint. Recovering from media failures (global redo) still requires a redo log; however, redundant disk storage makes this scenario less likely [Chen94]. Since undo records can be eliminated after a transaction commits, removing the redo log implies that no log records need be written to disk if memory is large enough to contain the undo records for all transactions in progress [Agrawal89]. In addition, storing the database buffer cache in reliable memory allows the system to begin operation after a crash with the contents present prior to the crash (a warm cache) [Sullivan93, Elhardt84, Bhide93].

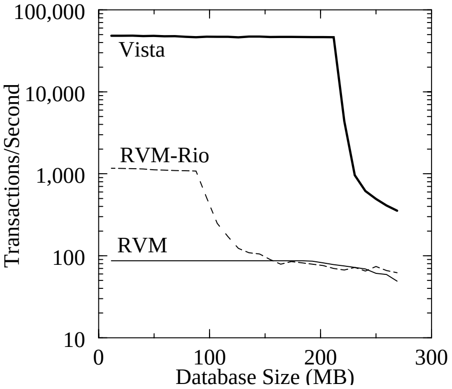


Fig. 2. Performance improvements with reliable memory. This figure shows the performance of three different transaction systems on a DEC 3000/600 with 256 MB memory, running a workload based on TPC-B. RVM is a simple transaction system without reliable memory. Running RVM on Rio (RVM-Rio) provides an I/O interface to reliable memory and speeds RVM up by a factor of 13. Vista uses a memory interface to reliable memory and achieves a factor-of-40 speedup over RVM, even though both run on Rio

Storing the database or the log in reliable memory is a more powerful technique than group commit (an optimization commonly used to improve transaction throughput) [DeWitt84]. Systems that perform group commit wait for a number of committing transactions to accumulate, then synchronously write to disk all commit records for this group in one I/O. While group commit does improve throughput, it suffers from several problems. First, it does not improve response time; waiting for a group of transactions to accumulate lengthens the response time of an individual transaction. Second, throughput for large transactions can be limited by the speed of writing to the redo log. Third, group commit works only if there are many concurrent transactions. Single-threaded applications with dependent transactions cannot use group commit because earlier transactions are delayed; these applications need the fast response times provided by reliable memory.

Storing the log and/or the buffer cache in reliable memory can thus simplify and accelerate database systems. A recent study shows that using a persistent database buffer cache can yield a system 40 times faster than using a non-persistent buffer cache, even when both run on reliable memory [Lowell97]. Figure 2 compares the performance of three systems on a workload based on TPC-B. RVM is a simple transaction system with a redo log and achieves about 100 transactions/second without reliable memory [Satya-

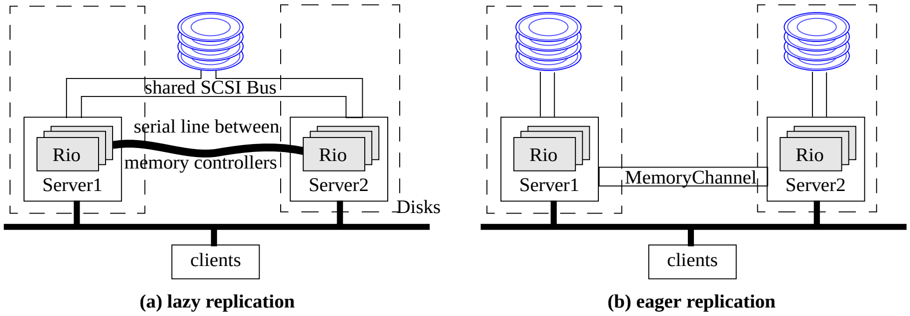


Fig. 3. Increasing the availability of reliable memory systems

narayanan93]. Running RVM on Rio with an I/O interface to reliable memory speeds it up by a factor of 13. Vista is a transaction system tailored to run on Rio. By using a persistent buffer cache, Vista achieves a factor-of-40 improvement over RVM, even though both run on Rio. Vista achieves this remarkable performance by eliminating the redo log, all system calls, and all but one copy. The performance improvement resulting from the simplicity of Vista (Vista is roughly 1/10 the size of RVM) is difficult to quantify but is probably also significant. For example, Vista's small code size leads to much shorter code paths for a transaction than RVM. Vista also avoids the double buffering that causes RVM-Rio performance to drop at 100 MB.

The performance gain provided by reliable memory is most dramatic for applications whose working sets fit in main memory, like main-memory databases. However, reliable memory can also improve database performance when data sets are larger than main memory. Reliable memory can be used in this case to delay writing new data to disk. Delaying the writes allows some data to be overwritten and the remaining data to be written efficiently to disk (for example, by writing new data to disk when the disk head passes over that track).

3 The Rio file cache

The Rio file cache is an area of memory, maintained by the operating system, that buffers file system data [Chen96]. It is protected from operating system crashes by virtual memory protection, and this protection is enhanced by configuring the processor to force all addresses through the translation-lookaside buffer. Protection adds negligible overhead when writing to a file using write system calls and does not affect the performance of mmap'ed files at all, because it changes only the kernel's protection map. Upon reboot, the file cache is restored to the file system on disk, a technique called warm reboot. Six machine months of continuously crashing the operating system (about 2000 crashes) showed that these techniques make the Rio file cache even safer from operating system crashes than a disk-based (write-through) file system. Of the crashes, 1.1% corrupted some data in a disk-based (write-through) file system, and 0.6% of the crashes corrupted some data in a file system using the Rio file cache. See [Chen96] for more details on these experiments and subsequent improvement in performance.

Rio's focus is reliability (not losing data). It achieves this by protecting memory during system crashes and recovering memory using warm reboot. In general, achieving high availability (being able to access data continuously) requires replication of the storage devices and/or computers [Gray93, Cardoza96]. By replicating the memory or computer, Rio can also provide high availability. This replication can be done lazily or eagerly. An inexpensive example of lazy replication is to connect a serial line between the memory controllers of two computers, and to connect disks to both I/O controllers (Fig. 3a). This provides low-bandwidth access to the data in memory while rebooting or during hardware failures. A more expensive solution is to eagerly replicate the file cache data on a remote computer [Papathanasiou97] using a high-speed network such as Memory Channel [Gillett96], Scalable Coherent Interface [Scott92], or a LAN (Fig. 3b). This provides higher availability if one machine is permanently disabled.

The goal of this paper is to explore how to use the Rio file cache to provide reliable memory for databases. Database systems traditionally encounter two problems in trying to use buffer caches managed by the operating system (the file cache) [Stonebraker81].

First, buffer caches managed by the file system make it difficult for the database to order updates to disk. These writes to disk need to be done in order to obey the constraints imposed by write-ahead logging [Gray78]. To order updates to disk, databases either use fsync or bypass the file cache entirely using direct I/O. The Rio file cache solves this problem completely, because data is persistent as soon as it enters the file cache. Thus, databases control the order of persistence by controlling the order that I/O is done to the file cache; no fsync is needed.

Second, databases can manage memory more optimally than a file system can, because databases know more about their access patterns. Our second software design (Sect. 5.2) addresses this problem by mapping the Rio file cache into the database address space. This exposes reliable memory to database crashes, and we quantify the increased risk posed by this design.

4 The Postgres storage system

We use the Postgres95 database management system developed at U. C. Berkeley as the database in our experiments [Stonebraker87]. Postgres has a few unique features which are relevant to this paper, but our results should apply to more conventional databases as well.

One novel aspect of Postgres is that it appends new data at commit. In contrast, conventional databases with write-

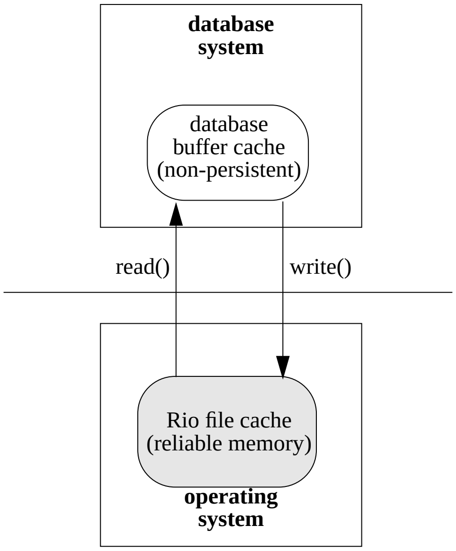

database system
database buffer cache (non-persistent)
read()
write()
Rio file cache (reliable memory)
operating system

Fig. 4. I/O interface to reliable memory. This design hides the reliable memory under the file system interface. The database uses read() and write() system calls to write data to the reliable memory. This design uses a traditional, non-persistent database buffer cache and thus requires change to the database. Because the interface to stable storage has not changed, this design is as safe as a standard database system from database crashes

ahead logs write undo/redo records at commit, then later write new data in place over the old version of the data. Postgres' scheme forces new data to disk at commit, whereas a conventional scheme forces only the log at commit (a no-force policy for the actual data). A force-at-commit policy decreases the amount of time database buffers are vulnerable to database crashes (Sect. 5.2).

As with nearly all database systems, Postgres keeps a database buffer cache in main memory to store frequently used data. Transactions modify the buffer cache data, then force the modified data to disk on commit. Because Postgres appends new data rather than overwriting it, a steal policy may be used without an explicit undo log. If a transaction aborts, the old copy of the data can be recovered from disk. Our second software design (Sect. 5.2) makes the database buffer cache persistent, and hence delays writing data to disk until after commit.

5 Software designs for integrating reliable memory

In this section, we describe three ways databases can include reliable memory and the implication of each design on reliability and performance.

5.1 I/O interface to reliable memory (non-persistent database buffer cache)

Our first design minimizes the changes needed to the database system by hiding the reliable memory under the file system interface (Fig. 4). The Rio file cache is used automatically when accessing the file system, so the database need only write persistent data to the file system instead of to the raw disk (or via direct I/O). No fsync is needed, because all file system writes to Rio are persistent immediately as soon as the data enters the file cache. In fact, Rio implements fsyncs as null operations. This removes all synchronous writes from the critical path of any transaction. This design requires no changes to the database; it needs only run on top of the Rio file system.

Because the interface to stable storage has not changed, this design is as safe as a standard database from database crashes. Recall that the Rio file cache is responsible for protecting and restoring the file system data if the operating system should crash. This transparency and reliability incurs some costs, however.

Using an I/O interface to reliable memory partitions main memory between the Rio file cache and the database buffer cache. The operating system would like the Rio file cache to be large so it can schedule disk writes flexibly and allow the largest number of writes to die in the file cache. But larger file caches reduce the size of memory available to the database. This partitioning creates two copies of data in memory (double buffering), one in the Rio file cache and one in the database buffer cache. Not only does this waste memory capacity, it also causes extra memory-to-memory copies.

One possible solution to these problems is to eliminate the database buffer cache and have the database use the file cache to store frequently used data. This is likely to make memory less effective at buffering database data, however, because a database can manage its buffer cache more effectively than file systems can (databases have more information on usage patterns). Researchers have proposed various ways for applications to control memory [Harty92, Patterson95, Bershad95, Seltzer96], and eventually this may enable the file cache to be as effective as a database buffer cache. At least for now, however, the best performance will be achieved by databases that wire down memory in their buffer caches and control it completely.

5.2 Memory interface to reliable memory (persistent database buffer cache)

Our second design maps the Rio file cache directly into the database system's address space using the mmap system call (Fig. 5). The database system allocates the database buffer cache (or redo log) from this area and wires these pages in memory. This design allows the database to manipulate reliable memory directly, using ordinary load/store instructions.

Using a memory interface to reliable memory has several advantages over the first design. First, management of this area of memory is completely under database control. Hence, no special support is required from the operating system to allow the database to determine replacement and prefetching policies.

Second, this design eliminates double buffering and extra memory-to-memory copies. The database simply manipulates data in its buffer cache, and these changes are automatically and instantly permanent. Hence, this design performs better than the non-persistent buffer cache (Fig. 2).

Third, this design can simplify databases by eliminating the need for redo logs and checkpoints (Sect. 2).

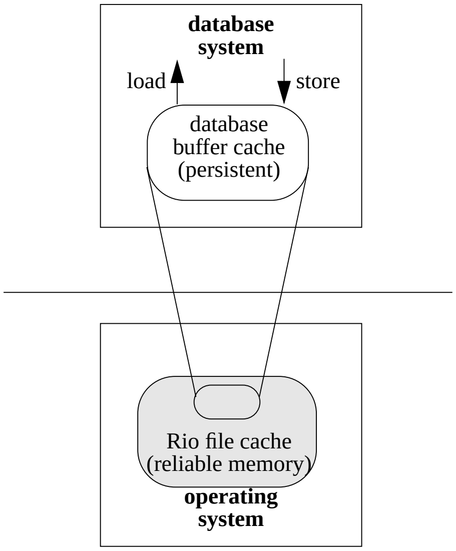

database system
load
store
database buffer cache (persistent)
Rio file cache (reliable memory)
operating system

Fig. 5. Memory interface to reliable memory. This design *maps* the Rio file cache directly into the database system's address space using the mmap system call. The database system can allocate its buffer cache from this region to make a persistent buffer cache. Access to stable storage (the persistent database buffer cache) takes place using load/store instructions to memory. This design eliminates double buffering and can simplify database recovery. However, database crashes can more easily corrupt stable storage than in the I/O interface design

Lastly, this design reduces the system recovery time, as it allows the database to start with a warm cache in the event of database process or operating system failures.

To implement the persistent database buffer cache, we map the buffer cache space into the Rio file cache. In this way, the database buffer cache's contents are preserved if Postgres or the operating system crashes. If Postgres crashes, the latest copy of the database buffer cache will be in the Rio file cache. If the operating system crashes, Rio will restore the contents of the Rio file cache in memory to the memory-mapped file on disk. In both instances, Postgres will restart with a warm cache, because the buffer cache file contains the exact copy of the database buffer cache prior to the crash. We modified Postgres slightly to avoid initializing the buffer cache during startup. The recovery process is idempotent, because the contents of reliable memory survive repeated crashes during system reboot.

Making the database buffer cache persistent leads to a few changes to the database. These changes are the same as those needed by a database using a steal policy [Haerder83]. The steal policy allows dirty buffers to be written back to disk (that is, made persistent) at any time. In particular, buffers may be made persistent before the transaction commits. This policy requires an undo log, so that the original values may be restored if the transaction aborts. Persistent database buffer caches require an undo log for the same reason, because all updates to the buffer cache are instantly persistent, just as if they had been stolen immediately. We implement the undo log in Postgres using shadow pages [Gray81]. When the system wants to write to database buffer cache, it first copies the contents to a shadow page and points the database buffer header to the shadow page. When the transaction is ready to commit later, it atomically points the database buffer cache header to the original buffer. Recovering from system crashes is done automatically, as the buffer header will be pointing to the before-image of the buffer data.

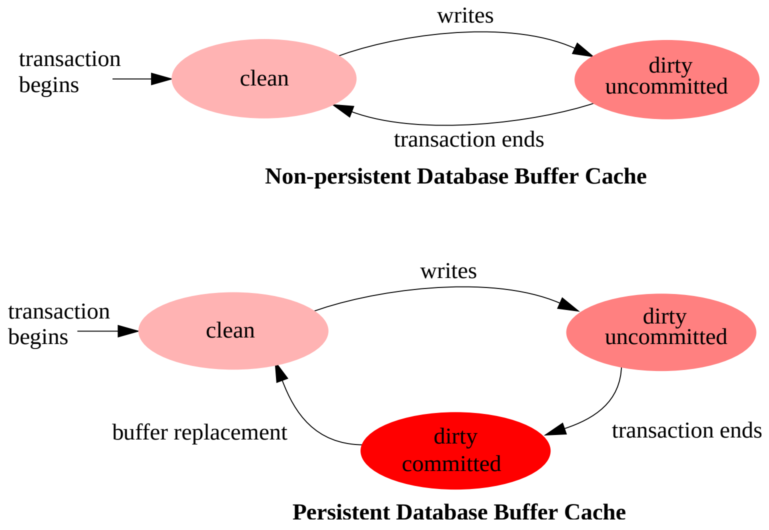
Other designs are possible that map the Rio file cache into the database address space. For example, the database log could be stored in reliable memory. Or an entire database could be mmap'ed, and the database could trust the virtual memory and file system of the operating system to page in and out appropriate pages. This latter approach may be appropriate for situations where the database program is one of several concurrent jobs (perhaps a machine running a client database program). In general, however, we believe that databases prefer to manage their own memory. Because of this, we mmap only the database buffer cache, and we lock these pages in memory to prevent paging.

The main disadvantage to using a memory interface to reliable memory is an increased vulnerability to software errors. The interface to stable storage with this design is now much simpler: load/store instructions instead of read/write system calls. Hence, it is easier for a software bug in the database to accidentally overwrite persistent data [Rahm92, Sullivan91a]. This section discusses the increased vulnerability conceptually, and Sect. 6 compares quantitatively the chances of data corruption among the different designs.

Consider the possible states of a database buffer (see Fig. 6). It may be clean or dirty, where dirty means the memory version of the buffer is different than the disk version. Dirty buffers may contain committed or uncommitted data. In our modification of Postgres95, we keep commit and dirty flags for each database buffer. After a database crash, we restore to disk only those pages that are marked as committed and dirty. Dirty pages that are not yet committed are restored to their before-image using the undo log. The following compares the vulnerabilities of different buffer states for persistent and non-persistent database buffer caches.

- **clean**. This state occurs when a piece of data is read from disk or when a dirty buffer is written back to disk. Buffers in this state are safe from single errors for both designs. To corrupt stable storage with a non-persistent buffer cache, the system would need to corrupt the buffer and later force it to disk (a double error). To corrupt stable storage with a persistent buffer cache, the system would need to corrupt the buffer and mark it as dirty (a double error). With either design, errant stores to buffers in this state may lead to corruption if other transactions read the corrupted data.
- **dirty, uncommitted**. This state occurs when a buffer has been modified by a transaction that is still in progress. Buffers in this state are equally vulnerable for both designs. In either design, stable storage will be corrupted if and only if the buffer is corrupted and the transaction commits.
- **dirty, committed**. This state indicates the buffer has been changed by a transaction and that the transaction has committed, but that the data has not yet been written to disk.

Dirty, committed buffers can exist in a persistent database buffer cache, because data is not written to disk until the buffer is replaced. Buffers in this state are vulnerable to software corruption; any wild store by another transaction



Non-persistent Database Buffer Cache

Persistent Database Buffer Cache

Fig. 6. Possible states of buffers in a persistent database buffer cache. The database buffer is initially clean when a transaction begins. As the transaction writes to the buffer, it becomes dirty but uncommitted. For a non-persistent database buffer cache, this buffer will become clean again when the transaction commits. However, for a persistent database buffer cache, the dirty uncommitted buffers are marked as dirty and committed when the transaction commits. These dirty committed buffers will remain in memory until the buffer is replaced, and are vulnerable to software errors and database crashes

can corrupt these buffers, and any change is instantly made persistent.

With non-persistent buffer caches, dirty, committed buffers can exist if the database uses a no-force policy. Buffers are dirty and committed until being forced to disk, at which time they are marked as clean. However, non-persistent buffer caches keep these buffers safer than persistent buffer caches. This is because the recovery process for non-persistent buffer caches discards memory buffers and uses the redo log to recover the data. Hence, if the database system corrupts a buffer in this state and crashes soon afterwards, the corrupted data will not be made persistent. Corruption occurs only if the system stays up long enough to write the affected buffer to disk.

Figure 6 shows that dirty, committed buffers make systems with persistent buffer caches more vulnerable to software corruption than systems with non-persistent buffer caches. Dirty, committed buffers are vulnerable for a longer period of time in a system with persistent buffer caches, particularly, compared to systems using a force policy (such as Postgres). And systems with non-persistent buffer caches experience corruption due to these buffers only if the system remains up long enough to propagate them to disk.

5.3 Memory interface to reliable memory with protection (persistent, protected database buffer cache)

Our third design also uses a memory interface to reliable memory but adds virtual memory protection to protect against wild stores to dirty, committed buffers (this scheme was suggested in [Sullivan91a]). In this system, clean or committed buffers are kept write protected. When a transaction locks an object, the page containing the object is unprotected; when the transaction commits, the page is reprotected. If multiple transactions use objects on the same page, the system reprotects the page when all transactions release their locks.

Figure 7 shows the buffer cache state transition diagram for the persistent protected database buffer cache. It is very similar to the persistent database buffer cache design. However, this scheme protects buffers (dirty committed data) in the state that our analysis in Sect. 5.2 showed are more vulnerable for persistent buffer caches. It also protects clean buffers when they are not being used. Dirty and uncommitted data have the same vulnerability in all three designs. Introducing protection thus increases the reliability of persistent database buffer caches, because clean and dirty/committed buffers (clear bubbles with dotted outlines in Fig. 7) are now protected from software errors.

Because the virtual-memory hardware uses a page granularity, this scheme exposes unrelated objects that happen to be on the same page as the locked object. The scheme does not prevent object-level locking, however, since this locking can be accomplished independently from our protection mechanism.

Section 6 measures how effectively the virtual-memory protection scheme protects dirty, committed, pages from wild stores. The disadvantage of this scheme is that the extra protection operations may lower performance.

6 Reliability evaluation

Persistent database buffer caches solve the double-buffering problem by placing stable storage under database control. As discussed in Sect. 5.2, however, this design may be more vulnerable to database crashes. This section compares the reliability of the different designs quantitatively by injecting software bugs into Postgres to crash it, then measuring the amount of corruption in the database. We detect database corruption by running a repeatable set of database commands modeled after TPC-B [TPC90] and comparing the database image after a crash with the image that should exist at the point at which the crash occurred.

6.1 Fault models

This section describes the types of faults we inject. Our primary goal in designing these faults is to generate a wide variety of database crashes. Our models are derived from studies of commercial databases and operating systems [Sullivan92, Sullivan91b, Lee93] and from prior models used in fault-injection studies [Barton90, Kao93, Kanawati95, Chen96].

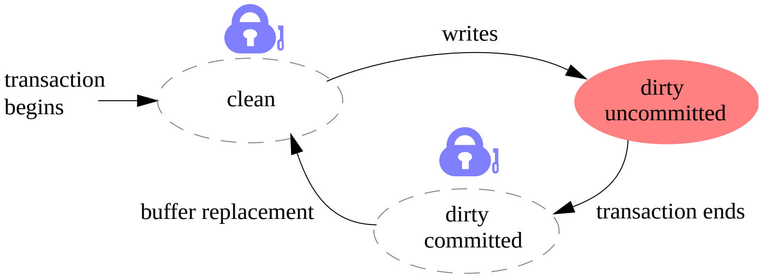


Fig. 7. Possible states of buffers in a protected persistent database buffer cache. The database buffers are protected using virtual-memory protection techniques whenever they are not in use. Dirty uncommitted buffers have the same degree of vulnerability in all three designs. Since clean and dirty committed buffers are now protected from accidental writes due to software errors and system crashes, this makes protected persistent database buffer cache safer than the first two designs

The faults we inject range from low-level hardware faults, such as flipping bits, in memory to high-level software faults, such as memory allocation errors. We classify injected faults into three categories: bit flips, low-level software faults, and high-level software faults. Unless otherwise stated, we inject five faults for each run to increase the chances that a fault will be triggered. Most crashes occurred within 10 s from the time the fault was injected. If a fault did not crash the database after 10 min, we restarted the database (and measured the amount of corruption as usual). This happened about 1/3 of the time and led to one instance of corruption.

The first category of faults flips random bits in the database's address space [Barton90, Kanawati95]. We target three areas of the database's address space: the text, heap, and stack. These faults are easy to inject, and they cause a variety of different crashes. They are the least realistic of our bugs, however. It is difficult to relate a bit flip with a specific error in programming, and most hardware bit flips would be caught by parity on the data or address bus.

The second category of fault changes individual instructions in the database text segment. These faults are intended to approximate the assembly-level manifestation of real C-level programming errors [Kao93]. We corrupt assignment statements by changing the source or destination register. We corrupt conditional constructs by deleting branches. We also delete random instructions (both branch and non-branch).

The last and most extensive category of faults imitate specific programming errors in the database [Sullivan91b]. These are more targeted at specific programming errors than the previous fault category. We inject an initialization fault by deleting instructions responsible for initializing a variable at the start of a procedure [Kao93, Lee93]. We inject pointer corruption by (1) finding a register that is used as a base register of a load or store and (2) deleting the most recent instruction before the load/store that modifies that register [Sullivan91b, Lee93]. We do not corrupt the stack pointer register, as this is used to access local variables instead of as a pointer variable. We inject an allocation management fault by modifying the database's malloc procedure to occasionally free the newly allocated block of memory after a delay of 0–64 ms. Malloc is set to inject this error every 1000–4000 times it is called; this occurs approximately every 10 s. We inject a copy overrun fault by modifying the database's bcopy procedure to occasionally increase the number of bytes it copies. The length of the overrun was distributed as follows: 50% corrupt one byte; 44% corrupt 2–1024 bytes; 6% corrupt 2–4 KB. This distribution was chosen by starting with the data gathered in [Sullivan91b] and modifying it somewhat according to our specific platform and experience. bcopy is set to inject this error every 1000–4000 times it is called; this occurs approximately every 5 s. We inject off-by-one errors by changing conditions such as $>$ to $>=$, $<$ to $<=$, and so on. We mimic common synchronization errors by randomly causing the procedures that acquire/free a lock to return without acquiring/freeing the lock. We inject memory leaks by modifying free() to occasionally return without freeing the block of memory. We inject interface errors by corrupting one of the arguments passed to a procedure.

Fault injection cannot mimic the exact behavior of all real-world database system crashes [Iyer95]. However, the wide variety of faults we inject (15 types), the random nature of the faults, and the sheer number of crashes we performed (2250) give us confidence that our experiments cover a wide range of real-world crashes. Table 1 shows examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments.

6.2 Reliability results

Table 2 presents reliability measurements of our three designs. We conducted 50 tests for each fault category for each of the three systems; this represents 2 machine-months of testing. Individual faults are not directly comparable because of non-determinism in the timing of the faults and differences in code between the different systems; instead we look primarily at trends between the different systems.

Overall, all three designs experienced few corruptions; only 2–3% of the crashes corrupted permanent data over 2250 tests. This argues that, even with a research database such as Postgres, software bugs may crash the system but usually do not corrupt permanent data. There are several factors contributing to Postgres's robustness. First, Postgres has many programmer assertions that stop the system soon after a fault is activated. Detecting the fault quickly and stopping the system prevents erroneous data from being committed to the permanent database image. Second, the operating system provides built-in assertions that check the sanity of various operations. For example, dereferencing a NULL pointer will cause the database to stop with a segmentation violation. In general, faults that left the system running for many transactions (such as off-by-one and interface) tended to corrupt data more frequently than faults that crashed the system right away (such as heap and stack).

We next compare the reliabilities of the three designs. The amount of corruption in all three systems is about the same. The differences are not large enough to make firm conclusions distinguishing the three systems, however we note that, as expected, the traditional I/O interface (non-

Table 1. Relating faults to programming errors. This table shows examples of how real-world programming errors can manifest themselves as the faults we inject in our experiments. None of the errors shown above would be caught during compilation

Fault type	Example of programming error	
	Correct code	Faulty code
destination reg.	**numFreePages** = count(freePageHeadPtr)	**numPages** = count(freePageHeadPtr)
source reg.	numPages = **physicalMemorySize**/page Size	numPages = **virtualMemorySize**/pageSize
delete branch	**while** (flag) {body}	**while** (!flag) {body}
delete random inst.	for (i = 0; $i < 10$; i + +, **j** + +) {body}	for (i = 0; $i < 10$; i + +) {body}
initialization	function () {int i = **0**; . . .}	function () {int i; . . .}
pointer	ptr = ptr → next→ **next**;	ptr = ptr → next;
allocation	ptr = malloc(N); use ptr; use ptr; **free(ptr)**;	ptr = malloc(N); use ptr; **free(ptr)**; use ptr
copy overrun	for (i = 0; $i <$ **sizeUsed**; i + +) {$a[i]$ = $b[i]$};	for (i = 0; $i <$ **sizeTotal**; i + +){$a[i]$ = $b[i]$};
off-by-one	for (i = 0; $i <$ size; i + +)	for (i = 0; $i <=$ size; i + +)
synchronization	**getWriteLock**; write(); **freeWriteLock**;	write();
memory leak	**free(ptr)**;	
interface error	insert(**buf**, **index**);	insert(**buf1**,**index**);

Table 2. Comparing reliability. This table shows how often each type of error corrupted data for three designs. We conducted 50 tests for each fault type for each of three systems using the TPC-B benchmark [TPC90] as the workload. We show the corruption rate for three weightings: equal, DB2, and IMS. Even without protection, the reliability of the persistent database buffer cache is about the same as a traditional, nonpersistent buffer cache. We have observed similar results in an earlier experiment using the Wisconsin benchmark [Bitton83] as the workload

Fault type	I/O interface	Memory Interface	Memory interface with protection
text	1	1	1
heap	0	0	0
stack	0	0	0
destination reg.	4	5	5
source reg.	2	2	2
delete branch	1	1	0
delete random inst.	2	2	2
initialization	0	1	1
pointer	0	0	0
allocation	0	0	0
copy overrun	0	0	0
off-by-one	5	5	3
synchronization	0	0	0
memory leak	0	0	0
interface error	4	3	3
Total	19 of 750 (2.5%)	20 of 750 (2.7%)	17 of 750 (2.3%)
DB2 Weights	1.7%	1.8%	1.6%
IMS Weights	2.1%	2.1%	1.8%

Table 3. Proportional mapping. This table shows how we map between the fault type in our study and those of [Sullivan92], and the corresponding weight assigned to each fault type

Fault Type	Classification	Weight	
		DB2	IMS
kernel text	statement logic	2.8%	3.9%
kernel heap	data error	3.3%	2.1%
kernel stack	data error	3.3%	2.1%
destination reg	data error	3.3%	2.1%
source reg	data error	3.3%	2.1%
delete branch	statement logic	2.8%	3.9%
delete random inst	statement logic	2.8%	3.9%
initialization	initialization	9.7%	11.1%
pointer	pointer	15.9%	20.3%
allocation	allocation	12.4%	9.3%
copy overrun	copy overrun	8.3%	6.5%
off-by-one	statement error	2.8%	3.9%
synchronization	synchronization	13.8%	8.3%
memory leak	memory leak	5.5%	6.5%
interface	interface	10.3%	13.9%

persistent buffer cache) is slightly more reliable than the memory interface (persistent buffer cache) (2.5% corruption rate versus 2.7%). Also as expected, adding protection to the persistent buffer cache improves its reliability (2.3%). Protection can increase reliability over an I/O interface by trapping errant stores to clean buffers and dirty, committed buffers.

As it is difficult to prove that our fault model represents real faults, we present two other interpretations of the data by varying the weights associated with each fault type according to fault distributions published on DB2 and IMS [Sullivan92]. Sullivan's study includes a detailed breakdown of software errors according to the following classification: deadlock and synchronization, pointer management, memory leak, uninitialized data, copy overrun, allocation management, statement logic, data error, interface error, undefined state, and other. Table 3 shows the mapping between our fault categories and Sullivan's studies, together with the resulting weights. Undefined state and other are too vague to be precisely modeled, so we distribute their weights evenly across other categories.

Our main conclusion is that mapping reliable memory directly into the database address space has only a small effect on the overall reliability of the system. This is consistent with the estimates given in [Sullivan91a, Sullivan93]. There are several factors that minimize the reliability impact of persistent buffer caches. First, most stores in Postgres are not to the buffer cache. Using the ATOM program analysis tool [Srivastava94], we found that only 2–3% of stores executed during a run were to the buffer cache. Second, store instructions that are not intended to access the buffer cache have little chance of accidentally wandering into buffer cache space, especially with the vast, 64-bit virtual address space on DEC Alphas. As a result, most corruptions are due to corrupting the current transaction's data. These uncommitted buffers are vulnerable to the same degree in all three systems (Sect. 5.2).

Thus, mapping reliable memory directly into the database address space does not significantly lower reliability. Combined with the advantages of persistent buffer caches (reliable memory under database control, no double buffering, simpler recovery), these results argue strongly for using a memory interface to reliable memory. Stated another way, the high-overhead I/O interface to reliable memory is not needed, because wild stores are unlikely to corrupt nonrelated buffers.

7 Related work

Section 2 summarized the many benefits of using reliable memory. In this section, we describe prior studies that have suggested methods for integrating and protecting reliable memory in databases.

The study most closely related to this paper was done by Mark Sullivan in the context of the Postgres project [Sullivan91a, Sullivan93]. Sullivan implemented two general methods for protecting database buffers from database errors using virtual-memory protection. Expose page unprotects a page before writing to a record on the page and reprotects the page after the write is done. Our protection model in Sect. 5.3 is very similar but does not reprotect the page until the transaction commits. Our method incurs fewer protection operations and is simpler to implement, because reprotection operations are localized to the commit function. Deferred write uses copy-on-write to make a private copy of a page for a transaction, then copies the data back on commit. Sullivan measures the performance overhead of these protection mechanisms for a debit-credit-type workload to be 5–10% when manipulating a database contained in nonvolatile memory (no disk activity) and 2–3% when manipulating a database too large to fit in memory and forcing data to disk.

Sullivan evaluates the reliability impact of these protection schemes by examining prior failure studies and estimating which fault categories were most likely to be affected. [Sullivan93] concludes that only 5–7% of errors are the type of error (wild stores) that would be prevented by his protection mechanism, although this ignores secondary effects such as wild stores generated by other errors. [Sullivan91a] mentions as future work the type of fault injection studies performed in this paper. These fault injection studies can provide more detailed data than extrapolating from prior failure studies, because the crashes are conducted under monitored environments.

Other general means to protect memory include using separate processes [Bartlett81], replication [Liskov91, Muller96], and software fault isolation [Wahbe93].

[Copeland89] discusses two organizations for integrating reliable memory (safe RAM) in databases. A separate safe uses an I/O interface to reliable memory, while an integrated safe is similar to our memory interface to reliable memory. [Copeland89] evaluates analytically the performance of the separate safe, but does not evaluate the effect on reliability of either organization.

[Rahm92] examines different technologies that can be used as reliable memory (SSD, disk cache, extended memory) but assumes these are not directly addressable by the processor. Hence, he evaluates only the performance benefits of using an I/O interface to reliable memory.

This paper extends the previous work in the following ways.

- We discuss the performance and reliability tradeoffs of different ways to integrate reliable memory provided by an operating system into a database. We also discuss effects of persistent buffer caches on undo/redo logging and cache policies (force, steal).
- We conduct fault experiments to measure the reliability of the three designs (non-persistent buffer cache, persistent buffer cache, persistent buffer cache with protection).
- Because the memory area we use is provided by the Rio file cache, the reliable memory area survives both database crashes and operating system crashes. Yet we do this while providing persistence to individual stores and without needing any extra disk activity. All other work that has provided a memory-mapped interface to reliable memory requires extra checkpoint/logging to disk.

8 Limitations

This paper describes a controlled experiment to measure database corruption. No prior studies have directly quantified database vulnerability to software crashes. However, it would be dangerous to indiscriminately apply our results, and hence we address in this section some possible limitations of our work.

How can we extrapolate our results to commercial database systems? We modified the Postgres95 [Stonebraker87] database management system for our experiment. Postgres differs from most commercial database systems in several ways. For example, it does not use conventional write-ahead logging (WAL), and it only provides a flat transaction model [Gray93]. Our work is largely independent of these differences; we focus on the reliability implications of persistent database buffer caches, and all databases use a buffer cache. Our results are also consistent with a prior study on a commercial database [Sullivan92]. We hope to conduct further experiments on commercial database systems to further support our hypothesis.

Can state-of-the-art database recovery schemes use reliable memory? As described in Sect. 2, any database can run unchanged on reliable memory. A reliable file cache such as Rio simply makes I/O appear faster. Reliable memory also allows state-of-the-art recovery schemes (e.g., ARIES [Mohan92]) to become significantly simpler. For example, the need for fuzzy checkpoints is reduced, because transactions can commit faster. Also, using a force-at-commit policy (combined with the replication techniques described in Sect. 3) reduces the need for a redo log.

9 Conclusions

We have proposed three designs for integrating reliable memory into databases. Keeping an I/O interface to reliable memory requires the fewest modifications to an existing database, but wastes memory capacity and bandwidth with double buffering. Mapping reliable memory into the database address space allows a persistent database buffer cache. This places reliable memory under complete database control, eliminates double buffering, and simplifies recovery. However, it also exposes the buffer cache to database errors. This exposure can be reduced by write protecting buffer pages.

Extensive fault tests show that mapping reliable memory into the database address space does not significantly hurt reliability. This is because wild stores rarely touch dirty, committed pages written by previous transactions. Combined with the advantages of persistent buffer caches, these results argue strongly for using a memory interface to reliable memory.

Acknowledgements. We thank the anonymous reviewers for helping to improve the quality of this paper. We have also benefited immensely from David Lowell's work on the Vista transaction system. This research was supported in part by NSF grant MIP-9521386 and Digital Equipment Corporation. Peter Chen was also supported by an NSF CAREER and Research Initiation Award (MIP-9624869 and MIP-9409229).

References

[Agrawal89] Agrawal R, Jagadish HV (1989) Recovery algorithms for database machines with nonvolatile main memory. In: Boral H, Faudemay P (eds) Sixth International Workshop, IWDM'89 Proceedings, June 1989, Springer, pp 269–285

[Akyurek95] Akyurek S, Salem K (1995) Management of partially safe buffers. IEEE Trans Comput 44(3):394–407

[APC96] American Power Conversion (1996) The Power Protection Handbook. Technical report. American Power Conversion, West Kingston, R.I.

[Bartlett81] Bartlett J (1981) A NonStop Kernel. In: Proceedings of the 1981 Symposium on Operating System Principles, ACM, pp 22–29

[Barton90] Barton JH, Czeck EW, Segall ZZ, Siewiorek DP (1990) Fault injection experiments using FIAT. IEEE Trans Comput 39(4):575–582

[Bershad95] Bershad BN, Savage S, Pardyak P, Sirer EG, Fiuczynski ME, Becker D, Chambers C, Eggers S (1995) Extensibility, Safety and Performance in the SPIN Operating System. In: Proceedings of the 1995 Symposium on Operating Systems Principles, ACM Press, pp 267–283

[Bhide93] Bhide A, Dias D, Halim N, Smith B, Parr F (1993) A Case for Fault-Tolerant Memory for Transaction Processing. In: Proceedings of the 1993 International Symposium on Fault-Tolerant Computing, June 1993, pp 451–460

[Bitton83] Bitton D, DeWitt DJ, Turbyfill C (1983) Benchmarking Database Systems-A Systematic Approach. In: Schkolnick M, Thanos C (eds) Very Large Database Conference, October 1983, Morgan Kaufmann, pp 8–19

[Bohannon97] Bohannon P, Lieuwen D, Rastogi R, Seshadri S, Silberschatz A, Sudarshan S (1997) The Architecture of the Dali Main-Memory Storage Manager. J Multimedia Tools Appl 4(2):115–151

[Cardoza96] Cardoza WM, Glover FS, WE Snaman Jr (1996) Design of the TruCluster Multicomputer System for the Digital UNIX Environment. Digital Tech J 1:5–17

[Chen94] Chen PM, Lee EK, Gibson GA, Katz RH, Patterson DA (19) RAID: High-Performance, Reliable Secondary Storage. ACM Computing Surveys, 26(2):145–188, June 1994.

[Chen96] Chen PM, Ng WT, Chandra S, Aycock CM, Rajamani G, Lowell D (1996) The Rio File Cache: Surviving Operating System Crashes. In: Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM Press, pp 74–83

[Copeland89] Copeland G, Keller T, Krishnamurthy R, Smith M (1989) The Case for Safe RAM. In: Apers PMG, Wiederhold G (eds) Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 1989, Morgan Kaufmann, pp 327–335

[DeWitt84] DeWitt DJ, Katz RH, Olken F, Shapiro LD, MR Stonebraker, Wood D (1984) Implementation Techniques for Main Memory Database Systems. In: Yormark B (eds) Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, June 1984, pp 1–8

[Elhardt84] Elhardt K, Bayer R (1984) A Database Cache for High Performance and Fast Restart in Database Systems. ACM Trans Database Syst 9(4):503–525

[GM92] Garcia-Molina H, Salem K (1992) Main Memory Database Systems: An Overview. IEEE Trans Knowl Data Eng 4(6):509–516

[Gillett96] Gillett RB (1996) Memory Channel Network for PCI. IEEE Micro 16(1):12–18

[Gray78] Gray JN (1978) Operating Systems: An Advanced Course. Notes on Database Operating Systems. Springer, Berlin Heidelberg New York

[Gray81] Gray J, McJones P, Blasgen M, Lindsay B, Lorie R, Price T, Putzolu F, Traiger I (1981) The Recovery Manager of the System R Database Manager. ACM Comput Surv 13(2):223–242

[Gray93] Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, Calif.

[Haerder83] Haerder T, Reuter A (1983) Principles of Transaction-Oriented Database Recovery. ACM Comput Surv 15(4):287–317

[Hagmann86] Hagmann RB (1986) A Crash Recovery Scheme for a Memory-Resident Database System. IEEE Trans Comput C-35(9):839–843

[Harty92] Harty K, Cheriton DR (1992) Application-Controlled Physical Memory using External Page-Cache Management. In: Proceedings of the 1992 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM Press, pp 187–197

[Iyer95] Iyer RK (1995) Experimental Evaluation. In: Proceedings of the 1995 International Symposium on Fault-Tolerant Computing (FTCS), pp 115–132

[Kanawati95] Kanawati GA, Kanawati NA, Abraham JA (1995) FERRARI: A Flexible Software-Based Fault and Error Injection System. IEEE Trans Comput 44(2):248–260

[Kao93] Kao W-L, Iyer RK, Tang D (19 93) FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. IEEE Trans Software Eng 19(11):1105–1118

[Lee93] Lee I, Iyer RK (1993) Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. In: Proceedings of the 1993 International Symposium on Fault-Tolerant Computing (FTCS), pp 20–29

[Liskov91] Liskov B, Ghemawat S, Gruber R, Johnson P, Shrira L, Williams M (1991) Replication in the Harp File System. In: Proceedings of the 1991 Symposium on Operating System Principles, ACM Press, pp 226–238

[Lowell97] Lowell DE, Chen PM (1997) Free Transactions with Rio Vista. In: Proceedings of the 1997 Symposium on Operating Systems Principles, ACM Press, pp 92–101

[Mohan92] Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P (1992) ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans Database Syst 17(1):94–162

[Muller96] Muller G, Banatre M, Peyrouze N, Rochat B (1996) Lessons from FTM: An Experiment in Design and

Implementation of a Low-Cost Fault-Tolerant System. IEEE Trans Reliab 45(2):332–340

[Papathanasiou97] Papathanasiou A, Markatos EP (1997) Lightweight Transactions on Networks of Workstations. Technical Report Technical Report No. 209, ICS-FORTH

[Patterson95] Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J (1995) Informed Prefetching and Caching. In: Proceedings of the 1995 Symposium on Operating Systems Principles, ACM Press, pp 79–95

[Rahm92] Rahm E (1992) Performance Evaluation of Extended Storage Architectures for Transaction Processing. In: Stonebraker M (eds) Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, pp 308–317

[Rosenblum95] Rosenblum M, Bugnion E, Herrod SA, Witchel E, Gupta A (1995) The Impact of Architectural Trends on Operating System Performance. In: Proceedings of the 1995 Symposium on Operating Systems Principles, ACCM Press, pp 285–298

[Satyanarayanan93] Satyanarayanan M, Mashburn HH, Kumar P, Steere DC, Kistler JJ (1993) Lightweight Recoverable Virtual Memory. In: Proceedings of the 1993 Symposium on Operating System Principles, ACM Press, pp 146–160

[Scott92] Scott SL, Goodman JR, Vernon MK (1992) Performance of the SCI Ring. In: Abramson D, Gaudiot J (eds) Proceedings of the 1992 International Symposium on Computer Architecture, ACM Press, pp 403–414

[Seltzer96] Seltzer MI, Endo Y, Small C, Smith KA (1996) Dealing With Disaster: Surviving Misbehaving Kernel Extensions. In: Operating Systems Design and Implementation (OSDI), October 1996, pp 185–199

[Srivastava94] Srivastava A, Eustace A (1994) ATOM: A System for Building Customized Program Analysis Tools. In: Proceedings of the 1994 Conference on Programing Language Design and Implementation (PLDI), pp 196–205

[Stonebraker81] Stonebraker M (1987) Operating system support for database management. Commun ACM 24(7):412–418

[Stonebraker87] Stonebraker M (1987) The design of the POSTGRES storage system. In: Stocker PM, Kent W, Hammersley P (eds) Proceedings of the 1987 International Conference on Very Large Data Bases, Morgan Kaufmann, pp 289–300

[Sullivan91a] Sullivan M, Stonebraker M (1991) Using write protected data structures to improve software fault tolerance in highly available database management systems. In: Camps R (eds) Proceedings of the 1991 International Conference on Very Large Data Bases (VLDB), September, Morgan Kaufmann, pp 171–180

[Sullivan91b] Sullivan M, Chillarege R (1991) Software Defects and Their Impact on System Availability – A Study of Field Failures in Operating Systems. In: Proceedings of the 1991 International Symposium on Fault-Tolerant Computing, pp 2–9

[Sullivan92] Sullivan M, Chillarege R (1992) A Comparison of Software Defects in Database Management Systems and Operating Systems. In: Proceedings of the 1992 International Symposium on Fault-Tolerant Computing, pp 475–484

[Sullivan93] Sullivan MP (1993) System Support for Software Fault Tolerance in Highly Available Database Management Systems. PhD thesis, University of California at Berkeley, Calif.

[TPC90] Transaction Processing Performance Council (1990) TPC Benchmark Standard Specification. Technical report. Transaction Processing Performance Council

[Wahbe93] Wahbe R, Lucco S, Anderson TE, Graham SL (1993) Efficient Software-Based Fault Isolation. In: Proceedings of the 14th ACM Symposium on Operating Systems Principles, December 1993, ACM Press, pp 203–216

[Wu94] Wu M, Zwaenepoel W (1994) eNVy: A Non-Volatile, Main Memory Storage System. In: Proceedings of the 1994 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 1994, ACM Press, pp 86–97