The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

A Transactional Memory Service in an Extensible Operating System

Yasushi Saito and Brian Bershad
University of Washington

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Transactional Memory Service in an Extensible Operating System

Yasushi Saito and Brian Bershad
{yasushi,bershad}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

This paper describes *Rhino*, a transactional memory service implemented on top of the *SPIN* operating system. *Rhino* is implemented as an extension that runs in *SPIN* kernel's address space. We discuss how the extension structure of *Rhino* can solve performance problems previously unavoidable in traditional systems, and we quantify its benefits. We also introduce three alternative buffer management schemes and study their performance under various workloads.

1 Introduction

This paper describes *Rhino*, a transactional memory service that lets applications perform transactions on a virtual memory through ordinary loads and stores.

Rhino runs on the *SPIN* operating system, which lets user applications modify and augment the kernel by safely downloading code (*extensions*) into the kernel address space [2]. *Rhino* is implemented as an extension. Extensions can efficiently manipulate kernel resources such as virtual memory pages and files, and can thus avoid the performance bottlenecks that plague traditional systems.

We have developed several versions of *Rhino* as our understanding of *SPIN* and its target applications grew. These versions share the same transaction mechanisms, but they differ in the way they detect updates to memory and manage buffers. In this paper, we explore the tradeoffs among these systems under different workloads. We also show how *SPIN*'s extension architecture enhances the performance of *Rhino* by comparing *Rhino* with its user-space, UNIX-based implementation.

1.1 Transactional Memory Requirements

Transactional memory service allows applications to access a database file mapped onto virtual memory regions in an atomic, isolated, and durable (also known as *ACID*) manner [8]. It is used as a runtime engine for managing persistent, pointer-rich data structures, such as graphic design databases, project management databases, and directory services.

Transactional memory is similar to a memory-mapped file, but the ACID property requires additional operating system support. To ensure atomicity, wherein a set of database accesses must be performed in an all-or-nothing manner, transactional memory records all updates to the database; it then replays or unwinds them after the system crashes or when the application aborts an operation. To isolate applications, transactional memory must lock virtual memory regions that the applications have accessed.

1.2 Limitations of Existing Systems

The functions provided by conventional operating systems do not meet the requirements of transactional memory systems. The following sections highlight three areas where existing operating systems respond poorly to the needs of transactional memory.

1.2.1 Write Detection

Transactional memory service must keep track of the database changes to ensure ACID property. In user-space transactional memory implementations, writes to the database are usually detected by the MMU protection and upcalls from the operating system (`SIGSEGV` signals in UNIX). However, such

implementations incur high overhead [21], as shown in Figure 1 (a). When a page fault occurs, the kernel performs a full context switch into the signal handler. The signal handler calls the server to bring the faulted page into client memory. It then issues a system call, such as `mprotect`, to change the MMU protection and makes a context switch back to the faulted context. The whole process requires at least eight user-kernel boundary crossings and four context switches.

1.2.2 Inter-process Communication

Most transactional memories are organized as client-server systems, since they can easily handle concurrent updates by multiple applications. These systems require frequent inter-process communication (IPC) for data fetching, database locking, and transaction control. IPC performance becomes more critical in a transactional memory services where clients and the server exchange data pages, than in relational database systems where only queries and answers are exchanged. Although fast IPC mechanisms have been studied extensively [1, 11], they have not made their way into mainstream operating systems. Thus, IPC in existing operating systems is slow, and it often becomes the bottleneck in transactional memory implementations.

1.2.3 Buffer Management

A transactional memory service must often handle databases whose size exceeds that of main memory. When the demand for memory exceeds a limit, the operating system evicts pages from applications. When a database buffer page held on ordinary virtual memory is evicted, extra disk accesses are required to swap out and later swap in the page. Straightforward implementations have been unable to address this problem, called *double paging* [13].

1.2.4 Organization of the Paper

Section 2 introduces the *SPIN* operating system. Section 3 reviews SPIN's *Rhino* extension. In Section 4, we discuss implementation issues and describe the advantages and disadvantages of three alternative buffer management schemes. *Rhino*'s performance is contrasted with UNIX implementations in Section 5. Section 6 examines works related to ours. We summarize our findings in Section 7.

2 Overview of *SPIN*

SPIN is an extensible operating system. Applications can safely extend the kernel functionality by downloading code into the kernel address space [2]. The *SPIN* operating system consists of the *kernel*, which provides basic services such as CPU scheduling and device management, and *extensions*, which are downloaded into the kernel address space after the kernel boots. Extensions efficiently communicate and share resources with the kernel and other extensions; from a performance viewpoint, they resemble dynamically linked modules.

The *SPIN* kernel and extensions are written in Modula-3 [14], a general purpose, typesafe language. The Modula-3 compiler, the *SPIN* runtime environment and the dynamic linker ensure that extensions cannot arbitrarily access memory or other critical resources, such as I/O ports and interrupt masks.

2.1 Use of Extensions in *SPIN*

Figure 2 shows a typical configuration of *SPIN*. Some *SPIN* extensions provide basic services used by other extensions. For example, file system extensions provide common directory and file operations like those found in UNIX. The virtual memory extension provides address spaces and memory objects similar to those found in Mach [23].

SPIN also supports user-space applications. Although the *SPIN* kernel does not support native system calls, it provides mechanisms that allow extensions to catch events from user-space applications, such as system calls and page faults. The UNIX emulation extension uses these mechanisms to provide the UNIX API for user-space applications[16]. *Rhino* is another extension that implements a transactional memory service for user-space applications. Note that unlike extensions, user-space applications can be written in any language. They are protected from other components by hardware mechanisms, as they are in other operating systems.

2.2 *SPIN*'s Approach to Transactional Memory

Rhino is implemented as an in-kernel extension. This design makes it possible to overcome the operating system deficiencies described in Section 1.1. For example, Figure 1 (b) shows page fault handling in *Rhino*. By placing the page fault handler inside the kernel space, *Rhino* can process a write detection event with two user-kernel crossings and zero

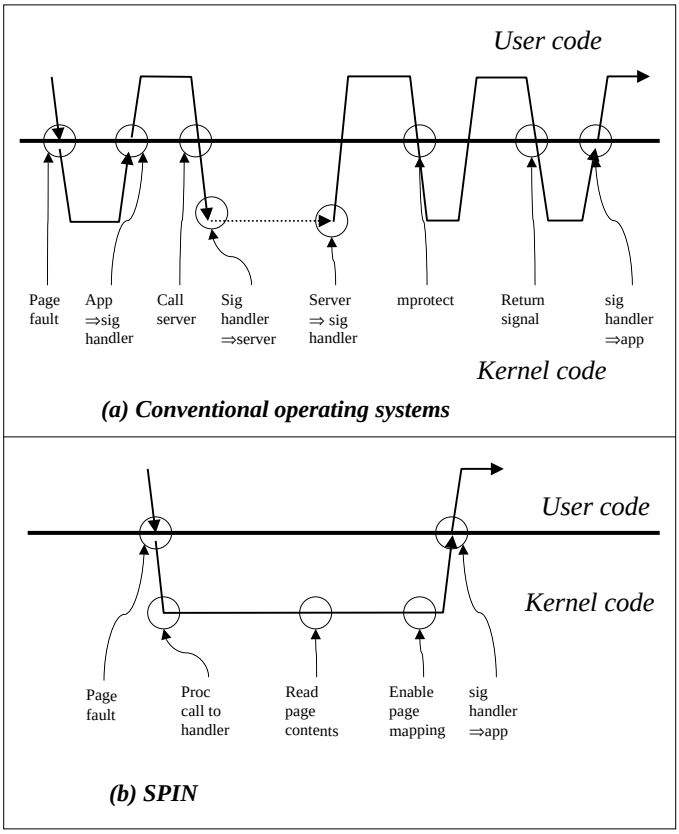

Figure 1: Page fault handling comparison. Conventional operating systems require at least eight user-kernel boundary crossings and four context switches to read the page contents in response to a page fault. In contrast, *SPIN* requires two user-kernel boundary crossings and no context switches.

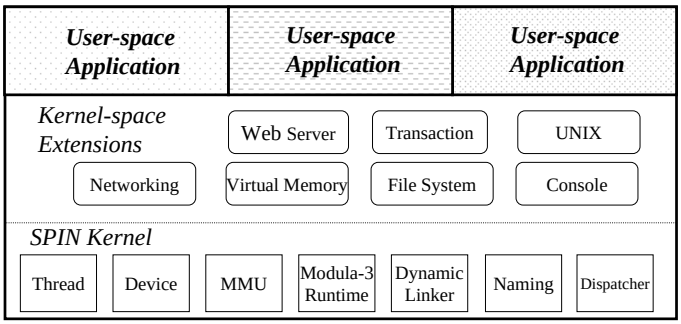

Figure 2: A typical *SPIN* configuration. The *SPIN* kernel and extensions are in the kernel address space. Each user-space application runs in its own address space (shown by different stipple patterns).

context switches.

Extensions also reduce the IPC overhead that exists in client-server systems. Typical IPC involves argument copying, context switching to the server (assuming the server runs on the client machine), writing back of results, and context switching back to the client. In *Rhino*, context switches are eliminated, since the extension is in the kernel address space and shared by all user-space applications.

Another example of the benefits of *SPIN*'s extension approach is buffer management. *Rhino* maps the database buffer directly onto an application's address space. It cooperates with the virtual memory extension to swap buffer pages directly to a database file rather than to a disk, thus solving the double paging problem.

3 Overview of *Rhino*

Rhino is structured as an extension that communicate with user-space applications via system calls. This section reviews the structure of the *Rhino* extension and its usage.

3.1 *Rhino* Structure

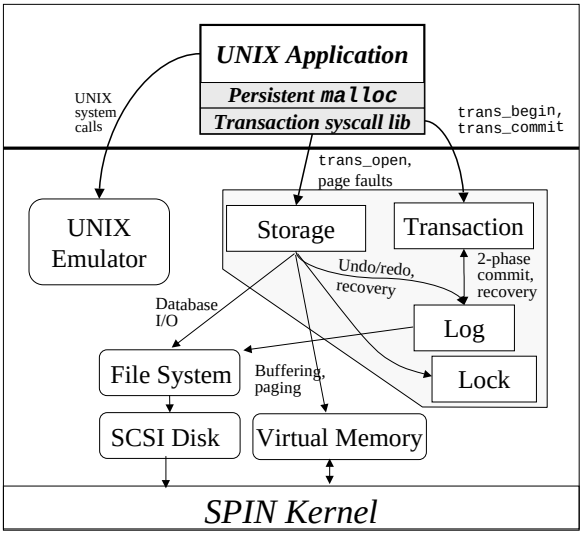

Figure 3: The *Rhino* extension structure. The *Rhino* system consists of shaded regions. It uses the file system extensions for database I/O and logging, and the virtual memory extensions for database buffering. *Rhino* is used by UNIX applications linked with the transaction library.

The shaded regions of Figure 3 show the structure of *Rhino*. Persistent `malloc` and system call stubs are linked into applications as a library. The following additional components reside in the kernel address space as an extension.

- The *transaction manager* starts and terminates transactions.
- The *storage manager* is the core of *Rhino*.[1] It detects applications' reads and writes and logs them to ensure the ACID property. It also manages buffers using memory objects.
- The *log manager* manages the log device, which is a sequential-write, random-read persistent device [8]. It is used by the storage manager to guarantee database durability and atomicity. The log manager also coordinates crash recovery when *Rhino* is installed.
- The *lock manager* manages locks on regions. It also detects deadlocks.

Rhino uses several standard extensions to carry out operations. It stores databases and log records in files managed by file system extensions. Files are usually stored in the extent-based file system, which allocates files on contiguous blocks and does not cache blocks in memory. *Rhino* cooperates with the virtual memory extension to manage database buffers. Its buffer management is described in detail in Section 4.

3.2 Application Programming in *Rhino*

Database files managed by *Rhino* are accessed either by user-space applications or by other in-kernel extensions. For ease of use and understanding, we use C nomenclature to present interfaces accessed by user-space applications.

Table 1 shows the system calls supported by *Rhino*. The system call interface resembles those found in other systems [17, 10, 20]. Figure 4 shows a simple application that writes "z"s onto the region that extends from byte 256 to byte 384 in the file "`/efs/test`". To access a database file, the application first calls `trans_open` to get a handle to the file. It then calls `trans_mmap` to map the file onto a virtual address space region (that must be MMU-page aligned). After setup of the transactional memory region, the application need

[1] Note that our use of the term "storage manager" differs from that in other database literature [8]. "Storage manager" commonly refers to the code that manages raw disk I/O. In our system, the storage manager is a resource manager specialized for file manipulation. Raw disk I/O is not part of the transaction service, since *Rhino* directly uses the raw I/O facility provided by *SPIN*'s file system.

only demarcate transactions by using `trans_begin`, `trans_commit`, and `trans_abort`. Accesses to the transactional memory region are detected through page faults.[2]

```
main()
{
    tid_t tid;
    sid_t sid;
    void *base = (void*)0x10000;

    /* Open the file /efs/test. */
    sid = trans_open("/efs/test");
    /* Map the storage from 0x10000 .. 0x90000. */
    trans_mmap(base, 0x80000, sid);

    tid = trans_begin();
    /* Fill the region with "z". */
    memset(base + 256, 'z', 128);
    if (something_wrong()) trans_abort(tid);
    else trans_commit(tid);
}
```

Figure 4: Sample application in user space.

4 Implementation Issues

This section describes the *Rhino* functions needed to implement a transaction. Buffering is the key to high performance. Locking and write detection are needed to ensure the ACID property [8].

4.1 Buffering

Rhino stores database contents in memory-mapped files; in other words, the database is buffered on pages that are mapped directly onto an application's address space. Memory-mapped files thus avoid the double paging problem even if there is memory competition.

Efficient buffering must comply with the write ahead logging (*WAL*) rule, which dictates that whenever a page is to be evicted, its log records (undo records) must be flushed to the log device [8, 13]. *Rhino* ensures WAL by implementing its own pageout procedure. Whenever the kernel chooses a page for purging, the *Rhino* pageout module is called to flush the log into the log device.

4.2 Locking

Rhino lets multiple applications access a database concurrently. To ensure isolation among transactions, database regions must be locked until a transaction finishes. *Rhino* asserts locks in MMU page grain using page faults.

[2] One version of *Rhino* requires a `trans_setrange` system call instead of page fault detection. In that version, "`trans_setrange(sid, 256, 128);`" is needed just before `memset`.

Before a transaction starts, MMU mappings for the database region are invalid. The first access to a page causes a page fault. The page fault handler in the *Rhino* extension obtains either a read or a write lock on the page, depending on whether the access is load or store. This scheme is essentially the same as that used in systems such as ObjectStore [10] and QuickStore [22]. Other locking approaches are possible, such as requiring applications to issue a system call to lock a region [7]. However, we decided on the MMU-based automatic locking approach to make the programming interface as simple as possible.

A shortcoming of this approach is that multiple threads in a single process cannot execute transactions simultaneously on the same database. However, this is not a serious problem since thread is not a unit of protection in most operating systems including *SPIN*; protecting database accesses by threads does not provide much help to programmers.

4.3 Write Detection

Transactional memory service must detect and log all writes to the persistent region, permitting changes to the database to be undone or redone atomically [8]. We implemented three versions of write detection in *Rhino* to study their performance trade-offs under various workloads. They are *setrange*, *page grain logging*, and *page diffing*. Setrange requires applications to issue a `trans_setrange` system call before modifying the database. The other two versions rely on the MMU to detect writes and differ in detection precision. Page grain logging treats the whole page as modified when at least one byte on the page changes. Page diffing tries to compute the exact set of modifications by comparing old and new page contents.

4.3.1 Setrange

The setrange approach creates a memory object for each open database file. It is mapped to the application's address space when `trans_mmap` is called. Before modifying a region in the database, the application must issue the `trans_setrange` system call to notify the *Rhino* extension about the region. In response to the call, *Rhino* pins down all buffer pages in the region so they will not be paged out until the transaction ends. It then records the region in a per-transaction record. Upon commit, *Rhino* scans the per-transaction record and logs the contents of each region as redo records. Thus, it implements a

System Call	Function
storage = trans_open(*path*)	Opens the database file *path*
trans_close(*storage*)	Closes the file
trans_setrange(*storage, from, len*)	Notifies the modification to *Rhino*. This system call exists only in one of the three alternative versions of buffer management.
trans_mmap(*addr, length, storage*)	Maps the file onto caller's address space
trans_munmap(*addr, length*)	Unmaps the file
trans_id = trans_begin()	Begins a transaction
trans_commit(*trans_id*)	Commits the transaction
trans_abort(*trans_id*)	Aborts the transaction and rolls back its effect

Table 1: *Rhino* API

no-steal, no-force policy [6]. Update detection using `setrange` was first implemented in RVM [17].

4.3.2 Page Grain Logging

Instead of relying on system calls from applications, page grain logging version uses MMU protection to detect writes. Database contents are stored in a memory object, as in the setrange version. All virtual memory mappings for the memory object are invalid before a transaction starts.

When the application writes onto the memory object, a page fault occurs, and the *Rhino* storage manager brings page contents in from disk, if necessary. The current contents of the page are then logged immediately as an undo record. Finally, the storage manager maps the page onto the application's address space. Upon commit, contents of all modified pages are logged as redo records.

When a buffer page is chosen as a pageout victim, the storage manager flushes the undo records generated for the page. Next, it writes the contents of the storage page into the database file. Finally, it removes the page from the memory object. Thus, this version implements a steal, no-force buffer management. Variations of page grain logging can be found in many transactional memory systems, including ObjectStore [10].

4.3.3 Page Diffing

Page diffing resembles page grain logging. However, it tries to reduce the size of the log by computing differences between old and new page contents. When a database file is opened, the storage manager creates two memory objects. The *storage object* is mapped onto the application's address space, and it caches the up-to-date contents of the file. The *shadow object* holds old buffer contents. It is not mapped onto address spaces; rather, it is used only to group pages together. Figure 5 shows how the two memory objects are used.

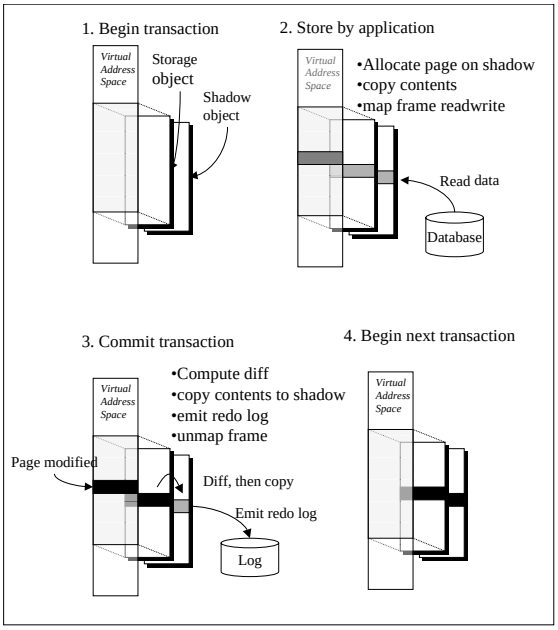

Figure 5: Page diffing algorithm

Before each transaction, all MMU mappings for the database region are invalid. In response to a page fault caused by an application store, the storage manager brings the page into the storage object, if necessary. It then allocates a page on the shadow object and copies the contents of the storage object page onto the new page. Finally, the storage object page is mapped onto the application's address space.

Upon commit, it compares the contents of each modified page and its shadow word by word and computes the differences between them. The "diff"s are logged as redo records. After commit, all mappings for the database region are invalidated.

When either a storage page or its shadow is chosen as a pageout victim, and the storage page is modified by some transaction, *Rhino* computes a "reverse diff" of the shadow and the storage page. The reverse diff is logged and flushed as the undo record. Next, the contents of the storage page are written out to the database file. Finally, both the storage page and the shadow page are removed from the memory objects. When the storage page is dirty but is not being modified by a transaction, *Rhino* writes the page to the database file without generating an undo record.

With this design, if the number of pages accessed by the transaction is smaller than the main memory size, no undo records are generated. Undo records are generated only when buffer pages are evicted.

Page diffing was first proposed in QuickStore [22]. Unlike *Rhino*, QuickStore does not allow dirty pages to be flushed before commit (no-steal policy). Thus, QuickStore does not generate reverse diffs.

4.3.4 Trade-offs among Write Detection Versions

We implemented the setrange version first because it was easy to understand and fast for small transactions since it can minimize the amount of redo records. However, we found a number of disadvantages when we ran bigger transactions.

- Setrange requires manual intervention.

  Programmers have to call `trans_setrange` manually before modifying a database region. Forgetting to make this call could result in unrecorded the changes to the database.

- The overhead of calling `trans_setrange` is significant in big transactions.

  When a transaction makes many `trans_setrange` calls, call overhead becomes sizeable. Note that this problem is less serious in systems like RVM [17] and Vista [12], which implement setrange as a library.

- Memory overhead increases in large transactions.

  The biggest problem with setrange is that each setrange region must be recorded until the transaction terminates. Thus, we cannot limit the amount of memory required to keep track of a transaction.

  A related problem is that setrange does not work well with the steal buffer management policy, which is essential to run large transactions. Because ranges must be remembered in memory until a transaction commits, it is difficult to evict pages in the middle of a transaction.

Page grain logging can limit memory consumption regardless of the amount of modifications: all modifications are recorded in memory object pages, and pages can be purged. It is less error prone than setrange, because it does not require programmer cooperation. Another advantage is that it can amortize the write detection cost when many bytes are updated in a page, because detection is required only once[3]. Thus, it is faster than setrange when many bytes are modified per page.

The problem with this approach is the log can grow quite large. For each modified page, log records twice as large as the page size are generated (one for an undo record, one for a redo record). This is wasteful for two reasons. First, it generates undo and redo records for the whole page even if only a single byte is modified on the page. Second, page grain logging blindly generates undo records even for transactions small enough not to require paging, in which case undo records are not needed [8].

Page diffing combines the advantages of setrange and page grain logging. It shares all the advantages of page grain logging. In addition, it can minimize log size by computing page diffs.

However, page diffing introduces overhead that did not exist in earlier versions. One source of overhead is the page diffing itself. Page diffing needs to walk over two pages and write out differences to another memory region. Not only is this procedure slow, but it retards other procedures by contaminating the CPU cache. Another source of overhead is the memory pressure imposed by shadow pages. In the worst case, one in which all accessed pages are modified, the effective memory size is halved. Thus, the system will have more paging activities.

5 Performance

This section evaluates the performance of *Rhino*. All measurements were carried out on a DEC Alphastation 250 with a 21064A CPU running at 266MHz, 47MB of user memory, and a DEC RZ26L 1GB SCSI disk.

We compared five systems: the three versions of

[3] When the page is purged and later brought in again, *Rhino* takes a page fault again.

Rhino running on *SPIN*, the page diffing version of *Rhino* running on Digital UNIX 3.2, and ObjectStore 4.0 running on Digital UNIX 3.2 [10].

Rhino on Digital UNIX uses the same page diffing code to detect modifications, but buffers are on ordinary virtual memory pages instead of a memory-mapped file, and page faults are detected using UNIX signals. Digital UNIX *Rhino* was measured to quantify the benefits of *SPIN*'s extension architecture, which allows low-cost communication between extensions and the kernel.

ObjectStore is a client-server database management system that buffers database contents on a client's virtual memory. It implements no-steal, no-force buffer management and page grain logging. ObjectStore is included to compare *Rhino* against a state-of-the-art, object-oriented database system.

We first present the micro-benchmarks that show the latency of the critical paths. Next, we present results from two benchmarks, RVM [17] and OO7 [3]. The RVM benchmark typifies small update transactions, while OO7 typifies graphical CAD database operations.

5.1 Micro-benchmarks

This section compares the micro-benchmark performance of *SPIN*-based *Rhino* and Digital UNIX-based *Rhino* to show how the extension architecture of *SPIN* improves the performance of critical functions. Table 2 shows the time breakdown of some important events.

Null call indicates a null system call overhead (on Digital UNIX, we measured the latency of `getpid`). *SPIN* is slower than Digital UNIX, because the implementation of system call in *SPIN* requires the use of additional mechanisms to protect the kernel from the runtime failure of an extension [16].

Begin shows the latency of `trans_begin`. *Commit(ro)* is the time to commit a read-only transaction. *Commit(8byte)* is the time to commit a transaction that modified 8 bytes on a single page. Page diffing is used during commits.

Four numbers are shown for page faults. "Read" faults are caused by load instructions, and "write" faults by store instructions. "Warm" faults occur when database contents are in main memory. Thus, these are times with no disk I/O. "Cold" faults occur when database contents are not in main memory and require pages to be read from the disk.

The *SPIN* version outperforms the UNIX version for all events except the null call. The performance difference is largest for warm page faults. There are two reasons for this: (1) since the page fault handler in *SPIN* runs in the kernel address space, it can eliminate most of the user-kernel crossings, and (2) page table manipulation in *SPIN* is more efficient than `mprotect` used in Digital UNIX. In *SPIN*, MMU can be manipulated by rewriting the MMU page table directly. On the other hand, `mprotect` requires more work, because it must manipulate the memory object map data structure to make its effect persist regardless of paging activity.

5.2 RVM Benchmark

The RVM benchmark is a program developed by the authors of RVM [17]. Each transaction reads and updates three 128-byte blocks and appends one 64-byte block to the end of the database. One 128-byte block is chosen randomly from the entire database; the other two 128-byte blocks are chosen from a narrow region. Thus, this benchmark measures the performance of small transactions.

We varied the database's size, ran 4000 transactions for each size, and calculated the mean time needed to complete one transaction. The number of bytes modified by each transaction does not depend on the database size. However, transactions running on small databases can utilize buffers more efficiently when many transactions are run successively.

Figure 6 shows the results. For small databases, setrange and page diffing perform almost equally well. However, as the database's size grows, the performance of page diffing drops quickly: the page diffing algorithm can utilize only half the amount of main memory available to the other schemes, since all the bytes accessed are modified in this benchmark. Page grain logging does a little worse than setrange for all database sizes because of increased logging activities. The UNIX page diffing version consistently performs about 1.5 to 2 times more slowly than *SPIN*'s page diffing. This difference is due to increased user-kernel crossings and extra data copying during I/O, because buffer pages are in ordinary virtual memory. ObjectStore fares badly in this benchmark. Since it performs page grain logging, whole page contents must be communicated to the server via IPC. Thus, it is not suited to small transactions.

5.3 The OO7 Benchmark

OO7 is the standard benchmark for object-oriented databases [3]. The database consists of objects of

Event	UNIX			SPIN		
	total	trap	other	total	trap	other
null call	2.14	2.14	–	6.11	6.11	–
begin	55.4	–	–	26.4	9.4	14
commit (ro)	152.3	–	–	29.7	13.4	16.3
commit (8byte)	14200	–	–	13328	15.2	13313
page fault (read, warm)	282.4	134	148.4	55.3	13.5	41.8
page fault (write, warm)	234.3	133	101.3	68.8	16.5	52.3
page fault (read, cold)	2881	131	2750	2272	20	2252
page fault (write, cold)	3059	113	2946	3054	19.7	3034

Table 2: Comparison of critical path latencies. The *total* columns show the total microseconds spent in each event. *Trap* columns show the overhead needed to pass control to the signal handler (on Digital UNIX) or to the *Rhino* page fault handler (on *SPIN*). Begin and commit for the UNIX implementation are implemented in the user space, and they issue multiple system calls. Thus, only the total elapsed times are shown for them.

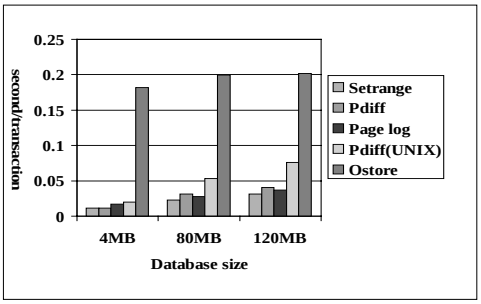

Figure 6: RVM benchmark results. 4000 transactions were run consecutively. The time to complete a single transaction is shown for each system.

various types connected in a tree structure (Figure 7).

We used the small, medium-3, and medium-9 configurations described in [3] (numbers in the medium configurations show the density of internal connections). The total database sizes are about 10MB for the small, 60MB for the medium-3, and 100MB for the medium-9.

We ran three types of traversals, *T1*, *T2A*, and *T2C*. They all traverse the object hierarchy and visit one element within each intermediate node. T1 visits all the elements in read only mode. T2a updates one element for each intermediate node. T2c updates each element four times.

Two types of numbers, *cold* and *warm*, are shown for the small configuration[4]. To obtain the cold numbers, we started the benchmark with an empty buffer cache[5]. An ObjectStore file open call (`objectstore::open`) pre-fetches some of the database contents into memory. The time needed to execute this procedure is also included in the cold numbers. Hot numbers are obtained by running four consecutive transactions after the cold run and computing the mean of the first three. For ObjectStore, the option to retain persistent pointers (`objectstore::retain_persistent_pointers`) was enabled. Thus, hot runs do not include pointer-swizzling overhead.

We do not report the setrange performance, because the setrange algorithm could not run OO7 for larger databases: as described in Section 4.3.4, setrange must retain all the range information in memory until a commit, and it uses up the in-kernel heap.

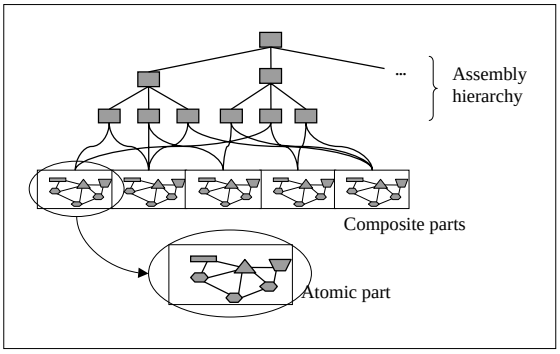

Figure 7: OO7 database structure. The database consists of composite parts, each of which is a web of atomic parts. Composite parts are indexed by a tree.

[4] Warm numbers are not shown for the medium configurations. In fact, they are almost same as their cold counterparts.

[5] We used a raw device for databases to bypass operating system caching.

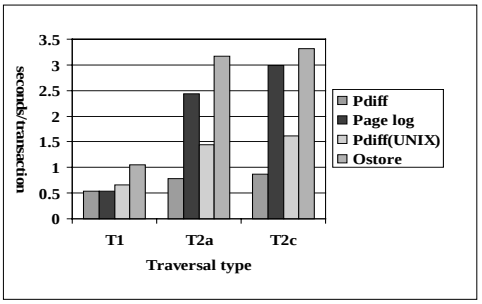

Figure 8: Small configuration results. Buffer cache was warm.

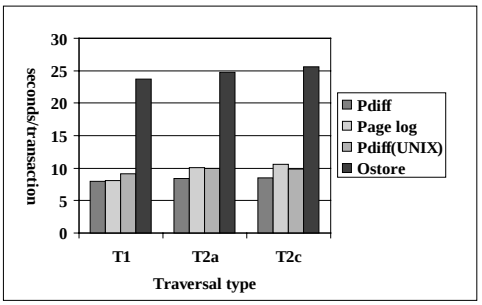

Figure 9: Small configuration results. Buffer cache was cold.

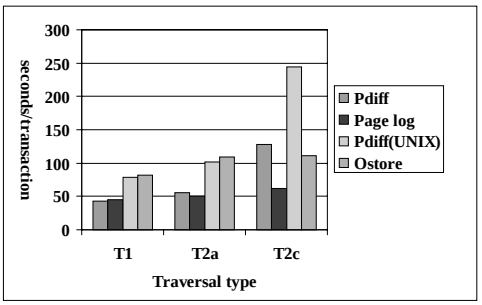

Figure 10: Medium configuration (fanout=3) results. Buffer cache was cold.

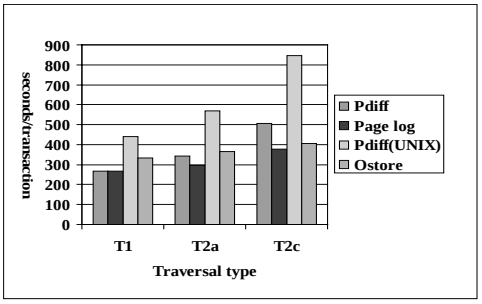

Figure 11: Medium configuration (fanout=9) results. Buffer cache was cold.

Figure 8 shows "warm" results from the small database. First, for T1 traversal, page grain logging and page diffing exhibit the same performance, because they perform the same tasks in read-only transactions. For other traversals, page diffing performs better than page grain logging because of smaller log activity. The UNIX page diffing version is consistently twice as slow as *SPIN*'s page diffing version, for the same reason described in the previous section. ObjectStore consistently performs worse than any of the *Rhino* versions. However, the performance discrepancy is smaller than in the RVM benchmark, because OO7 touches more bytes per page, and thus IPC overhead is amortized.

Figure 9 shows "cold" results for the small database. In the cold runs all *Rhino* versions perform about the same, because disk I/O dominates the time.

Figures 10 and 11 show the "cold" results on a medium database of fanout 3 and fanout 9, respectively. Page diffing does not perform well in these benchmarks, because of the memory pressure caused by shadow pages. Page grain logging is clearly the best choice in medium-3 and medium-9. ObjectStore performs better here, because the IPC cost is amortized over a large amount of updates.

5.4 Summary

This section compared the three versions of *Rhino* with a transactional memory service implemented entirely in user-space. The micro-benchmark numbers show that the extension structure of *Rhino* improves the latency of all the events critical to performance, especially the page fault handling. The two application benchmarks, RVM and OO7, compare trade-offs among the three *Rhino* buffer management alternatives. Page diffing version runs efficiently in small transactions. It can run larger transactions, but its performance suffers from memory

under-utilization due to shadow pages. Page-grain logging does worse for small transactions due to large log size, but it scales well up to large transactions. The setrange version is fastest in small transactions, but it does not scale for large transactions. The RVM and OO7 results also show that *SPIN*-based *Rhino* outperforms user-space transactional memory implementations.

6 Related Work

This section reviews systems related to *Rhino*. We include in our review object-oriented databases and persistent languages, since they are closely related to transactional memory.

ObjectStore [10], QuickStore [22], and QuickSilver [18] are client-server systems in which a server process manages transactions, and clients perform IPC to the server to access database contents. The advantage of this approach is that servers can transparently support clients running on different hosts. However, this approach also means that even local clients must communicate through a slow IPC channel, thus creating performance problems.

RVM [17] and Texas [20] are implemented as a library that is linked into user-space applications. Since they have no IPC overhead, unlike client-server systems, they can be fast. However, they are inherently single-user database systems, because there is no single authority that allows safe data sharing.

A problem common to the systems discussed thus far is double paging, since they are implemented as ordinary user-space applications. The only way to solve the double paging problem is to reserve a fixed amount of memory for the database management system and let the system perform its own paging. This approach is effective when the whole machine is dedicated to the database service. However, when there are other applications competing for memory, which is typical in transactional memories, this solution does not work well.

Some systems try to solve the double paging problem by using memory-mapped files and a special system call that lets user programs control the way pages are evicted. RPVM [5] adds a system call that dictates the order of page eviction. By telling the kernel to purge a buffer page after it purges log pages that record updates to that page, write ahead logging (WAL) can be implemented. Camelot [7] and Cricket [19] use the Mach external pager mechanism [23] to implement WAL. One difference between these systems and *Rhino* is that the former are user-space applications. Thus, they cannot avoid overhead due to a large number of user-kernel crossings.

Vista[12] uses Rio, a non-volatile file buffer, to implement transactions. Rio makes all updates to the buffer permanent immediately by recovering buffer contents during the system reboot. Thus, Vista transactions are orders of magnitude faster than transactions based on disk-logging.

Finally, there are systems that implement transactions inside the kernel. IBM CPR [4] and Pilot [15] support transactional updates of memory-mapped files. These systems solve problems found in other systems. However, most applications do not use transactions frequently enough to afford the complexity introduced by embedding transaction support in the kernel, making this approach uneconomical.

Herlihy and Moss proposed a transactional memory that is a CPU instruction set designed to support atomic memory updates[9]. Our use of the term is not related to theirs.

7 Conclusions

This paper described the implementation and performance of *Rhino*, a transactional memory implemented on the *SPIN* operating system. By implementing *Rhino* as an extension dynamically loaded into the kernel address space, we avoid problems associated with traditional systems, such as double paging and user-kernel boundary crossing overhead.

We implemented three write-detection approaches (setrange, page grain logging, and page shadowing) to study their trade-offs in the extension environment. Performance measurement demonstrate that all versions of our system outperform user-space implementations. Also, among the three variations of *Rhino*, it was found that setrange and page diffing perform equally well for small transactions. Page grain logging performs well for large transactions.

The *SPIN* operating system, as well as the *Rhino* transactional memory service described in this paper, can be obtained via the world wide web at http://www.cs.washington.edu/research/projects/spin.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM TOCS*, 8(1):37–55, February 1990.

[2] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, Safety and Performance in the SPIN Operating System. In *ACM SOSP-15*, Copper Mountain, CO, December 1995.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. In *SIGMOD Conference Proceedings*, pages 12–21, Washington D.C., June 1993. ACM.

[4] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM TOCS*, 6(1):28–50, January 1988.

[5] Khien-Mien Chew and Avi Silberschatz. Toward Operating System Support for Recoverable-persistent Main Memory Database Systems. Technical Report CS-TR-92-05, University of Texas at Austin, September 1992.

[6] Wolfgang Effelsberg and Theo Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[7] Jeffrey Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon*. Morgan Kaufmann, San Francisco, CA, 1991.

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[9] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ICSA*. ACM, 1993.

[10] Object Design Inc. home page. http://www.odi.com.

[11] Jochen Liedtke. On Micro-Kernel Construction. In *ACM SOSP-15*, Copper Mountain, CO, December 1995.

[12] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *ACM SOSP-16*, 1997. See also http://www.eecs.umich.edu/pmchen/.

[13] Dylan McNamee. *Virtual Memory Alternatives for Transaction Buffer Management in a Single-Level Store*. PhD thesis, University of Washington, November 1996.

[14] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice Hall, 1991.

[15] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *CACM*, 23(2):81–92, February 1980.

[16] Yasushi Saito and Brian Bershad. System Call Support in an Extensible System. See http://www.cs.washington.edu/-homes/yasushi, September 1997.

[17] M. Satyanarayanan, Henry Mashburn, Puneet Kumar, David Steere, and James Kistler. Lightweight Recoverable Virtual Memory. *ACM TOCS*, 12(1):33–57, February 1994.

[18] Frank Schmuck and Jim Wyllie. Experience with Transactions in QuickSilver. In *SOSP-13*, pages 239–253, October 1991.

[19] Eugene Shekita and Michael Zwillling. Cricket : A Mapped, Persistent Object Store. Technical Report TR-956, University of Wisconsin, 1990.

[20] Vivek Singhal, Sheetal Kakkad, and Paul Wilson. Texas: An Efficient, Portable Persistent Store. In *Proc. Fifth International Workshop on Persistent Object Systems*, pages 11–33, September 1992.

[21] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *ASPLOS 6*. ACM, October 1994.

[22] Seth J. White. *Pointer Swizzling Techniques for Object-Oriented Database Systems*. PhD thesis, University of Wisconsin, September 1994.

[23] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *ACM SOSP-11*, pages 63–76, Austin, TX, November 1987.