Two-Phase Update for Scalable Concurrent Data Structures

Paul E. McKenney
IBM NUMA-Q & Oregon Graduate Institute
pmckenne@us.ibm.com

Abstract

Parallel shared-memory software must control concurrent access to shared resources. This is typically accomplished via locking designs that result in low concurrency or that are highly complex.

This paper presents two-phase update, which is an alternative method of updating concurrent data structures, and demonstrates that it is both simple and highly scalable in restricted but commonly occurring situations. Example uses are taken from Sequent's (now IBM's) DYNIX/ptx operating system and from the Tornado and K42 research projects. Two-phase update techniques have been used in production by DYNIX/ptx since 1993, and were independently developed by Tornado and K42.

1 Introduction

Control of concurrent access to shared resources is a central issue in shared-memory software. Higher levels of concurrency usually require higher levels of complexity. For example, searching for and deleting items from a circular singly linked list is trivial in a non-parallel environment. Use of code locking (Hoare monitors) increases complexity only slightly, since locks must be acquired and released. However, since only one CPU at a time may be manipulating the list, concurrency remains low. There are many algorithms that allow concurrent searches, but that are substantially more complex, as exemplified by Manber's and Ladner's search tree [Manber84].

Thus, traditional implementations of large-scale parallel systems are forced to trade off simplicity against scalability. Three important situations necessitating such tradeoffs are deadlock avoidance, access to read-mostly data structures, and access to data structures that must be modified or deleted while the access proceeds. The common thread through these three situations is the need to destructively modify a data structure in face of concurrent access to that data structure. Traditional approaches, exemplified by locking, take a pessimistic approach to consistency, prohibiting any access from seeing stale data. But this pessimistic approach can restrict scalability—or force increasingly complex workarounds to be adopted in order to retain scalability.

Two-phase update addresses this problem by taking a more optimistic approach, allowing the user of two-phase update to decide how much consistency he or she is willing to pay for. In some important cases, two-phase update leads to designs that are both simple *and* highly scalable. Two-phase update accomplishes this by splitting updates into two phases: 1) carrying out the update so that new operations see the new state, while retaining the old state so that already-executing operations may continue using it, then: 2) cleaning up the old state after a "grace period" that is long enough to allow all executing operations to complete. Common-case accesses can then proceed fully concurrently, without disabling interrupts or acquiring any locks to protect against the less-common modifications. This simplifies locking protocols, improves both single- and multi-processor performance, and increases scalability.

The operations that were already executing at the beginning of the update will see the old state, *i.e.*, stale data. In many situations, this is acceptable. For example, any algorithm that tracks state external to the computer (e.g., routing tables or physical position data) must tolerate stale data in any case due to communications delays. Any access that begins after the first phase of the update has completed is guaranteed to see the new state, and this guarantee is often sufficient. In other cases, there are well-known techniques for insulating algorithms from stale data that are both efficient and highly scalable [Pugh90].

If two-phase update is to work well, the "grace period" during which old operations are allowed to complete must be acceptably small. Therefore, two-phase update may not be the method of choice for applications with long-duration operations. However, two-phase update does work well in many event-driven systems, such as operating-system kernels, where operations complete quickly. Example uses of two-phase update are shown in Section 7.

If two-phase update is to be practical, there must be an efficient algorithm that determines the duration of the grace period. A simple algorithm is presented in Section 4. Extremely efficient algorithms are readily available [Sarma01].

Section 2 describes existing solutions to the problem of rare destructive updates in face of frequent concurrent access and Section 3 lists the conditions and assumptions for two-phase update. Section 4 gives details of two-phase update's implementation. Section 5 carries out an analytical comparison of two-phase update and traditional locking techniques, and Section 6 presents measured comparisons for both performance and complexity. Section 7 covers existing uses of two-phase update in both research and production operating systems, Section 8 covers future work, and Section 9 presents conclusions.

2 Existing Solutions

Unlike more traditional synchronization methods, two-phase update is incomplete: it resolves the read-update conflicts, but it is necessary to use conventional synchronization to resolve any concurrent updates. As such, the benefits of two-phase update usually accrue to reads rather than updates. It is nevertheless instructive to compare two-phase update with conventional synchronization methods. The following sections presents such a comparison and surveys work touching on two-phase update.

2.1 Traditional Solutions

Table 1 compares two-phase update to a number of other methodologies.

	Complete	Wait Free	Zero-Cost	Rollback/Redo	Stale Data	Memory Reuse	Race Avoidance
Locking	Y	N	N	N	N	Y	N
Wait-Free Synchronization	Y	Y	N	Y	n	N	N
Optimistic Locking	Y	N	N	Y	n	Y	N
Timestamp-Based Locking	N	N	N	Y	Y	Y	N
Two-phase update	N	*	Y	*	Y	Y	Y

Table 1: *Synchronization Comparison*

A methodology is *complete* if it can solve general concurrency problems, and *wait free* if no process will be indefinitely delayed, even in face of process failure. It provides *zero-cost reads* if reading threads can execute the same sequence of code that would be sufficient in a single-threaded environment. *Rollback/redo* is possible if threads must discard or redo work they have done in response to concurrent accesses or updates. *Stale data* can be observed when reading threads are permitted to access data that is concurrently updated. Some methodologies ensure that reading threads never see stale data (indicated by "N"),

others allow stale data to be accessed, but prevent it from affecting the final state of the computation (indicated by "n"), and still others allow stale data to affect the final state of the computation (indicated by "Y"). *Memory reuse* means that the memory used for a given data structure can potentially be reused for any other data structure. *Race avoidance* prevents races from happening, as opposed to correctly handling them once they occur.

Locking is well represented in the literature; we include token schemes and slotted protocols in this category. The common theme throughout locking is that ownership of some resource, such as a lock, a token, or a timeslot, permits the owner to access or modify the corresponding guarded entity. Locking is complete, not wait free, does not permit zero-overhead reading, is not subject to rollback or stale data, and does allow arbitrary reuse of memory. This holds true for both exclusive locking and reader-writer locking [Court71, MC91, Hsieh91].

Wait-free synchronization [Herlihy93, Ander95] uses atomic instructions (load linked and store conditional on RISC machines, compare-and-swap on CISC machines) to ensure forward progress and to ensure that readers see consistent data. Wait-free synchronization is complete, and, as its name implies, wait free.

However, reading threads must perform *validation* operations in order to see a consistent snapshot of the data, and these operations include modifications to shared data that can result in expensive cache-miss or broadcast-update operations in the underlying hardware.

Furthermore, concurrent accesses and modifications to a given data structure can result in expensive rollback/redo operations (though these can be avoided in a few specific situations). In addition, in the absence of a garbage collector, any memory used for a particular data structure must not subsequently be used for some other data structure, thus making it harder to recover from memory-exhaustion-based denial-of-service attacks. However, wait-free synchronization does prevent any stale data from affecting the state of subsequent computations.

Optimistic locking is reviewed by Barghouti and Kaiser [Bargh91]. It is not normally used in operating systems or to guard memory-based structures in applications, but is presented here because of its handling of stale data. Optimistic locking uses a combination of timestamps and conventional locking to begin updates before it is known to be safe to do so. Optimistic locking is complete, but because of the locks, neither wait free nor zero-overhead to readers. Potentially unsafe situations result in rollbacks, thus preventing any stale data from affecting the state of subsequent computations. Finally, optimistic locking permits arbitrary reuse of memory.

Timestamp-based concurrency control is also reviewed by Barghouti and Kaiser [Bargh91], and is also not typically used in operating systems or general applications. Timestamp-based locking provides a versioning capability so that a reader may see a particular version of the guarded data despite subsequent updates. This versioning capability allows stale data to affect subsequent computation if desired, but requires additional storage, and operations may need to be aborted or rolled back if storage is exhausted. Furthermore, control of the versioning normally uses locking. Unlike the preceding synchronization methodologies, timestamp-based locking is not complete. It cannot be used to resolve concurrent conflicting updates, because giving each writer its own version of the data is usually not appropriate.

Two-phase update is similarly incomplete: normally, either traditional locking or wait-free synchronization is used to resolve concurrent updates. In the former case, two-phase

update is immune from rollback/redo,[1] while in the latter case, it is itself wait free. In either case, it does allow zero-overhead access to reading processes and allows memory to be reused independent of prior uses, even when wait-free synchronization is used to guard updates. However, two-phase update does expose readers to stale data, so they must be capable either of detecting and rejecting it, or of tolerating it.

2.2 Two-Phase Update Algorithms

This section discusses published concurrency control algorithms related to two-phase update.

Manber and Ladner [Manber84] describe a search tree that defers freeing a given node until all processes running at the time that the node was removed have terminated. This allows reading processes to run concurrently with updating processes, but does not handle non-terminating processes such as those found in operating systems and server applications. In addition, they do not describe an efficient mechanism for tracking blocks awaiting deferred free or for determining when the relevant processes have terminated. Nonetheless, this is an example of two-phase update, where the end of the grace period is detected upon termination of all processes running during the first phase of the update.

Pugh's skiplist search structures [Pugh90] use a technique similar to that of Manber and Ladner, but accommodate non-terminating processes by requiring that reading threads explicitly record the fact that they are accessing the data structure. This explicit recording requires writes to shared memory, resulting in expensive cache misses or global broadcasts at the hardware level. However, Pugh does not discuss a mechanism for efficiently tracking blocks awaiting deferred free in absence of a garbage collector. Nevertheless, this is an example of two-phase update, where the end of the grace period is detected when all processes that were accessing the data structure during the first phase of the update complete their accesses.

Kung [Kung80] describes use of a garbage collector to manage the list of blocks awaiting deferred free, allowing reading processes to run concurrently with updating processes. However, garbage collectors are not always available, and their overhead renders them infeasible in some situations. Even when garbage collectors are available and acceptable, they do not address situations where some operation other than freeing memory is to be performed soon after all reading processes have dropped references to the blocks awaiting deferred free. For example, a thread might wish to use two-phase update to determine when a log buffer was completely filled in, in order to flush it to disk. Again, this is another example of two-phase update, with the grace period ending when the memory becomes reclaimable by the garbage collector.

Jacobson [Jacob93] describes perhaps the simplest possible two-phase update technique: executing the first phase of the update, then simply waiting a fixed amount of time before executing the second phase. This works if there is a well-defined upper bound on the length of time that reading threads can hold references. However, if threads hold their references longer than expected (perhaps due to greater-than-expected load or data-structure size), memory corruption can ensue, with no reasonable means of diagnosis. Once again, this is another example of two-phase update, with the second phase being entered after the passage of a given amount of time.

[1] There are some wait-free algorithms that are immune from rollback/redo [Michael98], and if these are used to handle updates, then the enclosing two-phase update algorithm is also immune from rollback/redo.

McKenney and Slingwine [McK98a] describe an efficient algorithm for detecting states during which no operation can be in progress on a given CPU (called "quiescent states") within an operating system or server application. Any time a CPU passes through such a state, all ongoing operations running on that CPU must have completed. They describe use of this algorithm for an two-phase update algorithm named read-copy update, along with a few analytical and measured comparisons between read-copy update and its traditional lock-based equivalents.

Gamsa et al. [Gamsa99] describe a two-phase update algorithm that is used to make existence guarantees. Generation counters and tokens are used to guarantee that data structures are not returned to free storage while they are still being referenced. This algorithm differs from read-copy update in that the system maintains a count of busy threads rather than directly counting quiescent states (states that indicate that no operation is in progress on the current CPU or thread).

2.3 Directory-of-Services Example

Since locking is the oldest and perhaps most commonly used of the existing solutions, it is worthwhile to look at an example applying it to a singly linked list. This example ilustrates the problem of correctly handling requests for a service that arrive just as that service is torn down. If this race condition is not correctly handled, a particularly unlucky request will acquire a reference to the data structure representing the service just as this structure is deleted. The request will then find itself referencing an outdated structure, or, worse yet, an element on the freelist. Either possibility can result in failure, or even in a crash.

The simplest way to handle concurrent uses and teardowns of a service is to prohibit concurrency entirely through use of a global lock: either a service is present when requested or it is not. Unfortunately, this design is also incapable of making good use of more than a single CPU.

Introducing separate locks for each service can increase concurrency, so that multiple services can execute in parallel. Although this design has the advantage of being able to make good use of multiple CPUs, it also raises the specter of deadlock, since an additional lock will be required to guard the list itself. In addition, since the list lock will still be a global lock, Amdahl's law dictates that concurrency will be poor if the units of work performed by the services tend to be of short duration. For example, if the units of work take three times as long as the list search, then this design will be able to make good use of at most four CPUs.

If the linked list's search key allows hashing, then the singly linked list may be replaced by a hash table of list headers, each with its own list lock. The concurrency of this design is excellent, being limited only by the number of hash buckets, the quality of the hash function, and the number of list elements. However, it is still plagued by deadlock issues.

As will be shown in Section 4, two-phase update provides a simple and scalable solution to this problem.

3 Conditions and Assumptions

Use of two-phase update is most likely to be helpful with read-intensive data structures, where a modest amount of memory may be spared for a list of deferred actions, where stale data may be either tolerated or suppressed, and where there are frequently occurring natural or artificial quiescent states.

"Read intensive" means that the update fraction (ratio of updates to total accesses) f is much smaller than the reciprocal of the number of CPUs. However, in some special cases, two-phase update can provide performance

benefits even though *f* exceeds 0.9. It is possible for *f* to be as small as 10^{-10}, for example, in storage-area network routing tables (consider 100 disks, each with 100,000-hour mean time between failure, connected to a system doing 3,000 I/Os per second).

Ever-increasing memory sizes tend to make any space required to retain state across the grace period a non-problem, but the need for tolerance of stale data cannot always be so easily dismissed. However, any reading thread that *starts* its access *after* an update completes is guaranteed to see the new data. This guarantee is sufficient in many cases. In addition, data structures that track state of components external to the computer system (e.g., network connectivity or positions and velocities of physical objects) must tolerate old data because of communication delays. In other cases, old data may be flagged so that the reading threads may detect it and take explicit steps to obtain up-to-date data, if required [Pugh90].

Two-phase update requires that the modification be compatible with lock-free access. For example, linked-list insertion, deletion, and replacement are compatible: a reading access will see either the old or new state of the list. However, if a list is reordered in place, the reading thread can be forced into an infinite loop if the last element is consistently moved to the front of the list each time a reading thread reaches it. It is possible to perform an arbitrary two-phase-update modification of any data structure by making a copy of the entire structure, but this is inefficient for large structures. More work is needed to better understand how more general modifications can be efficiently cast into two-phase update form.

Finally, two-phase update is less applicable to non-event-driven software, such as some CPU-bound scientific applications, although similar techniques have been used, as reviewed by Adams [Adams91].

4 Details of Solution

Two-phase update exploits the fact that many software systems (such as most operating systems) continually perform many small and quickly completed operations.

4.1 Illustrative Example

Consider once again the singly linked list example from Section 2.3, as shown in Figure 1.

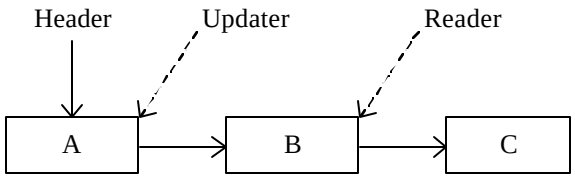

Figure 1: *List Initial State*

To delete element B, the updater thread may simply link A's pointer to C, as shown in Figure 2. This action constitutes the first phase of the update.

At this point, any subsequent searches of the list will find that B has been deleted. However, ongoing searches, such as that of the reader thread, may still find element B: these threads see stale data.

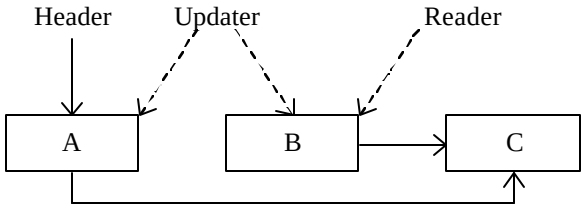

Figure 2: *Service B Unlinked From List*

The question answered by two-phase update is "when is it safe to return element B to the freelist?" The updating thread need wait only until all ongoing searches complete before returning B to the freelist, as shown in Figure 3. Again, any new searches will be unable to acquire a reference to B.

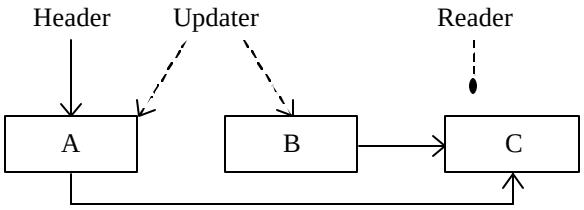

Figure 3: *List After Ongoing Operations Complete*

At this point, the updater thread can safely return B to the freelist, as shown in Figure 4. This constitutes the second phase of the update.

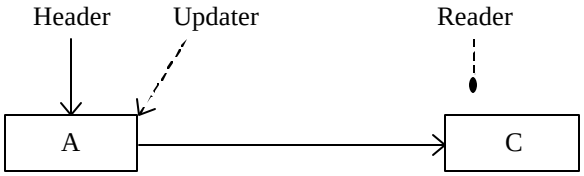

Figure 4: *List After Service B Returned to Freelist*

For this return to freelist to be safe, the reader thread must be prohibited from retaining a reference to element B across operations. This is equivalent to the prohibition against maintaining similar references outside of the corresponding critical sections in traditional locking. In either case, the data structure might be arbitrarily modified in the meantime, possibly rendering the reference invalid.

4.2 Implementation of Solution

Figure 5 shows how two-phase update progresses in an event-based system. The boxes represent individual operations, and each numbered arrow represents an active entity, for example, a CPU or a thread, with time progressing to the right. The dotted line labeled f_1 indicates the time of the first phase of the update. The second phase of the update may proceed as soon as all operations that were in progress during the first phase have completed, namely, operations A, E, and L. The earliest time the second phase can safely be initiated is indicated by the dotted line labeled f_2.

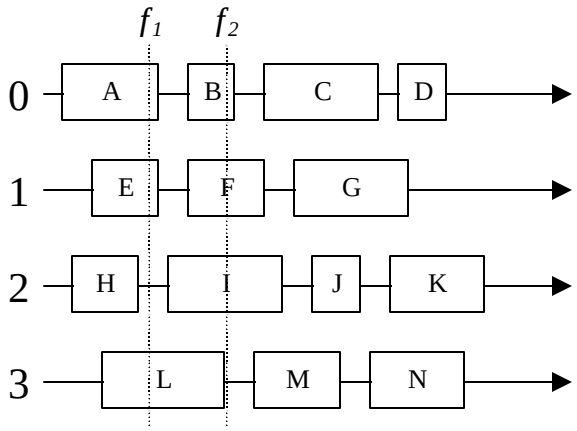

Figure 5: *Two-Phase Update in Event-Based Systems*

A simple but inefficient procedure to determine when the second phase may safely be initiated in a non-preemptive operating-system kernel is depicted in Figure 6. The updater simply forces itself to execute on each CPU or thread in turn. The boxes labeled "*u*" represent this updater's execution. Once it has run on each CPU or thread, then the non-preemptive nature of the environment guarantees that all operations that were in progress during phase one must have completed.

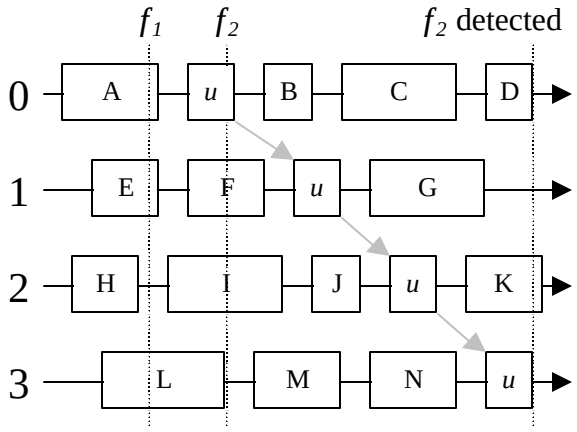

Figure 6: *Simple Phase-Two Detection*

Pseudo-code for two-phase update based on this approach is shown in Figure 7. Line 1 does the phase-one operation (in the example above, removing service B from the list). Lines 2 and 3 force execution onto each CPU in turn, potentially blocking until that CPU becomes available.

The key point is that a given CPU cannot possibly become available until after the completion of any operation that was in progress concurrently with the phase-one operation. Line 5 performs the phase-two operation (in the example in Section 4.1, freeing up the structure representing service B).

This procedure may be adapted to preemptive environments by requiring that preemption be suppressed during searches of the services directory, perhaps by suppressing interrupts or by setting a per-thread or per-CPU flag that suppresses preemption. Any "operations" that did not suppress preemption would be executed as multiple operations if preempted.

```
1  /* Perform phase-one modifications */
2  for (i = 0; i < n; i++) {
3      run_on(i);
4  }
5  /* Perform phase-two modifications */
```

Figure 7: *Pseudocode to Wait for End of Grace Period*

This procedure, shown in Figure 7, is quite straightforward, but has a number of shortcomings. The first is that switching from CPU to CPU (or from thread to thread) is quite expensive. The second is that there is no attempt to share grace periods among multiple concurrent updates, so that each update incurs the full cost of switching among all CPUs. The last shortcoming is that this style of use of two-phase update is too verbose.

Much more efficient procedures exist. These operate by counting *quiescent states* that are guaranteed not to occur within an operation. A useful set of quiescent states for a non-preemptive operating-system kernel includes context switch, execution of user code, system calls, traps from user code, and execution in the idle loop. Once each CPU has passed through a quiescent state, the system has passed through a grace period, at which point, any operations in progress at the start of the grace period are guaranteed to have completed.

Counting and combining-tree techniques are used to efficiently detect grace periods. In addition, detection of a single grace period can allow any number of two-phase updates to enter their second phase, so that the overhead of detecting a grace period may be amortized over a large number of requests. An efficient callback-based implementation of two-phase update has been produced for the Linux kernel [Sarma01] as well as for DYNIX/ptx [McK98a]. These callback-based implementations allow a callback to be registered, so that the specified callback function will be invoked at the end of the grace period.

In addition, many uses of two-phase update simply free memory in phase two. These uses can be greatly simplified via a deferred-free primitive such as DYNIX/ptx's kmem_deferred_free(). This primitive places memory to be freed on a list, and the items on this list are freed only after a grace period has passed. This means that many algorithms may be transformed to use two-phase update simply by using kmem_deferred_free() to free memory. See Section 6.2 for an example.

5 Analytical Comparison

This section presents analytical results comparing an efficient callback-based implementation of a two-phase update algorithm [Sarma01] to traditional locking methods. The analysis assumes low levels of contention, and further assumes that memory latency dominates the cost of computation. These assumptions allow performance to be estimated by simply counting memory references and estimating the latencies based on the probable prior state of the corresponding cache line [McK99]. This section compares the overhead of the concurrency-control primitives only. Real systems would also consider the overhead of the critical sections guarded by the locking primitives, but the locking-primitive-only comparisons in this section allow the locking design to be evaluated

independently of the data structure being manipulated by the critical section.

Details of the derivations may be found in Appendix A and in a companion paper and technical report [McK98a, McK98b].

5.1 Nomenclature

The symbols used in the analysis are defined in Table 2. This analysis assumes a four-level memory hierarchy that has a small L1 cache with negligible latency, a larger L2 cache, local memory (or an L3 cache shared by a quad's CPUs for repeated references to remote memory), and remote memory.

	Definition
f	Fraction of data-structure accesses that do updates. This is called the *update fraction*.
m	Number of CPUs per quad. A "quad" is a unit of hardware containing CPUs and memory that is combined with other quads to obtain a CC-NUMA system. The results presented here have $m=4$.
n	Number of quads.
t_c	Time required to access the fine-grained hardware clock.
t_f	Latency of a memory access that hits the CPU's L2 cache.
t_m	Latency of a memory access that hits local memory or the L3 cache.
t_s	Latency of a memory access that misses all caches.
r	$t_s/t_f = (t_m/t_f)^2$, also called the *latency ratio*.
l	Expected number of updates per grace period.

Table 2: *Analysis Nomenclature*

The graphs shown in Figure 8 through Figure 17 label traces and regions between traces with the corresponding locking algorithm, as shown in Table 3.

Figure 8 through Figure 11 display memory latency, normalized by t_f, on the y-axis. Memory latency is an appropriate cost measure for today's computer systems because several decades governed by Moore's law have increased CPU performance to such an extent that CPU overhead is usually insignificant when compared to memory latency. Although it is possible to analyze performance using memory latencies of a specific computer system, this section instead uses latency ratio r. This normalization of the latencies with respect to t_f allows the analysis to be easily applied to many types of computer systems.

Label	Description
`sl`	Simple spinlock
`drw`	Distributed reader-writer spinlock [Hsieh91]
`qrw`	Per-quad distributed reader-writer spinlock
`2pb`	Best-case two-phase update
`2pn`	Two-phase update with l equal to 0.1
`2pz`	Two-phase update with l equal to 1
`2pp`	Two-phase update with l equal to 10
`2pw`	Worst-case two-phase update

Table 3: *Key for Figure 8 Through Figure 17*

5.2 Costs

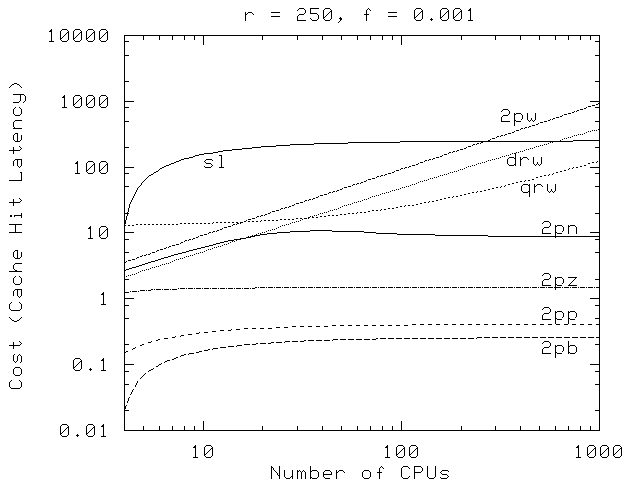

Figure 8: *Cost vs. Number of CPUs (Key: Section 5.1)*

Figure 8 displays two-phase update overhead as a function of the number of CPUs, with a latency ratio *r* of 250 and an update fraction of 0.001. At these typical latency ratios and moderate-to-high values of *l*, two-phase update outperforms the other locking primitives. Note particularly that overhead of non-worst-case two-phase update does not increase with increasing numbers of CPUs, due to the batching capability of two-phase update. Although simple spinlock also shows good scaling, this good behavior is restricted to low contention. The poor behavior of simple spinlock under high contention is well documented.

Figure 9 shows two-phase overhead as a function of the update fraction *f* given a latency ratio *r* of 250 and 30 CPUs. As expected, two-phase update performs best when the update fraction is low.

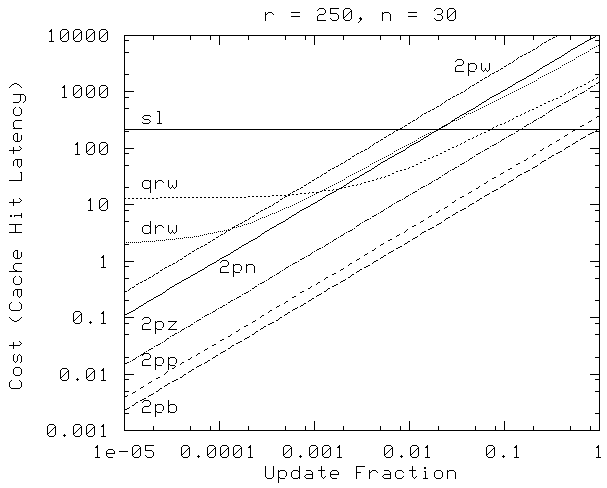

Figure 9: *Cost vs. Update Fraction (Key: Section 5.1)*

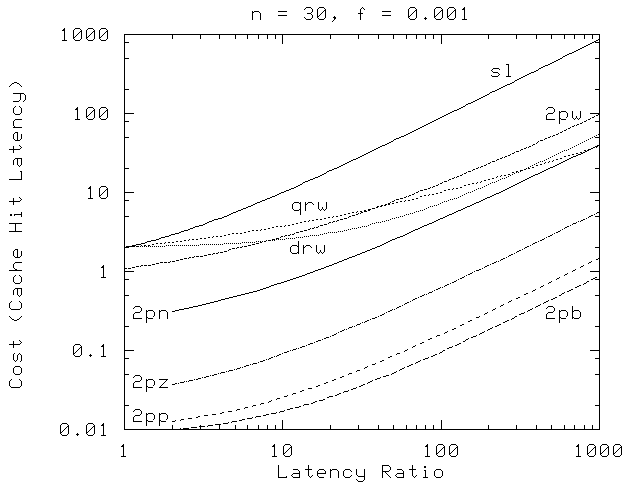

Figure 10: *Cost vs. Latency Ratio (Key: Section 5.1)*

Figure 10 shows two-phase overhead as a function of the latency ratio r, for 30 CPUs and with an update ratio of 0.001. The distributed reader-writer primitives perform well at high latency ratios, but this performance is offset in many cases by high contention, larger numbers of CPUs, or by lower update fractions. This last is illustrated in Figure 11, which shows the effects of an update fraction of 10^{-6}.

The situation shown in Figure 11 is far from extreme. As noted earlier, common situations can result in update fractions below 10^{-10}. Two-phase update will remain optimal for these situations for the foreseeable future, despite the likelihood that latency ratios will increase over time [Hennes91, Stone91, Burger96].

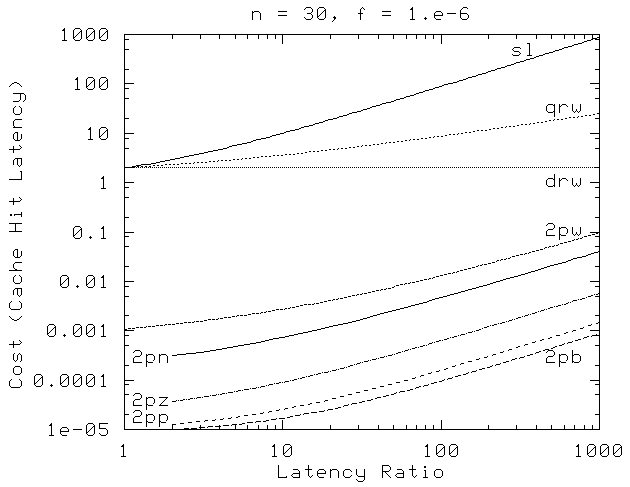

Figure 11: *Overhead vs. Latency Ratio for Low f (Key: Section 5.1)*

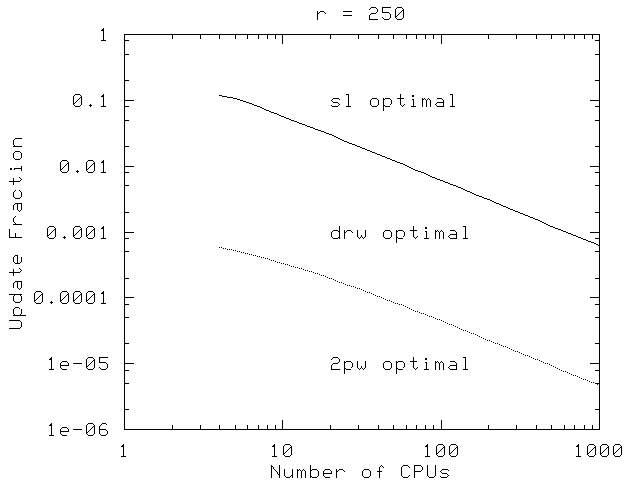

Figure 12: *Worst-Case Two-phase Update Breakevens (Key: Section 5.1)*

5.3 Breakevens and Optimal Regions

Since different techniques have the lowest costs under different conditions, this section presents plots showing the regions in which each technique is best. Table 3 shows the key for Figure 12 through Figure 17.

Section 5.3.1 presents breakeven update fractions for varying numbers of CPUs with the latency ratio *r* fixed at 250. Section 5.3.2 presents breakeven update fractions as *r* varies, with the number of CPUs fixed at 32.

5.3.1 Update Fraction vs. Number of CPUs

This section shows how well each of simple spinlock, Hsieh and Weihl reader-writer spinlock, and two-phase update scale with the number of CPUs.

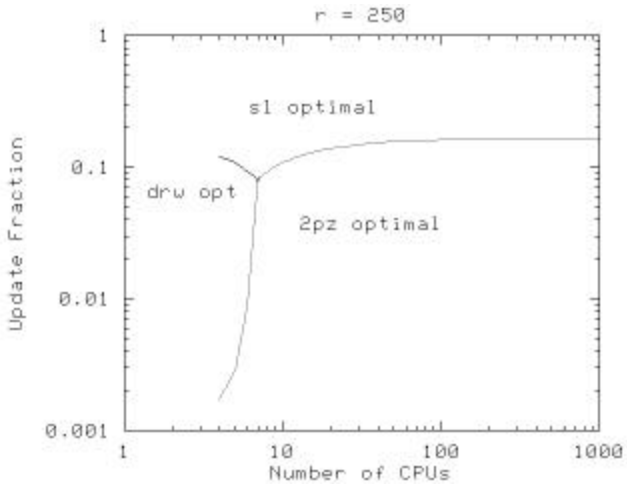

Figure 13*: l=1 Two-phase Update Breakevens (Key: Section 5.1)*

Figure 12 shows breakevens for small *l*, so that each update bears the full cost of detecting a grace period. There are three distinct regimes where each of simple spinlock, reader-writer spinlock, and two-phase update are optimal. As expected, given sufficiently low values of the update fraction, two-phase update is optimal. However, the larger the number of CPUs, the smaller the update fraction must be for two-phase update to be optimal. Similarly, a larger number of CPUs requires a smaller

update fraction in order for reader-writer spinlock to perform better than simple spinlock.

Figure 13 shows breakevens for *l=1*, so that on average one update occurs per grace period. Here, reader-writer spinlock ("`drw`") is optimal only in a small area with fewer than 10 CPUs. This is due to reader-writer lock's inability to amortize write-acquisition overhead over multiple updates. The breakeven between simple spinlock and reader-writer spinlock has not changed, but the breakeven between simple spinlock and two-phase update has moved up substantially due to the amortization of grace-period detection over multiple updates.

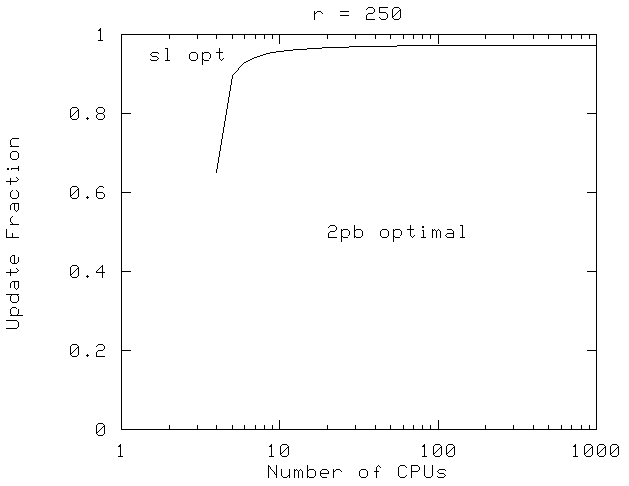

Figure 14: *Best-Case Two-phase Update Breakevens (Key: Section 5.1)*

Figure 14 shows breakevens for large *l*, so that per-update costs of detecting grace periods approaches zero. Again, reader-writer spinlock is never optimal. Two-phase update is optimal almost everywhere, even for very update-intensive workloads. Note that this figure has a linear-scale y-axis so that the breakeven between simple spinlock and two-phase update may be distinguished from the y=1 axis.

5.3.2 Update Fraction vs. Latency Ratio *r*

The previous section showed how two-phase update scales with numbers of CPUs. However, changes in computer architecture have resulted in large changes in latency ratios. This section shows how such changes affect the locking-primitive breakevens. Table 3 shows the key for the figures in this section.

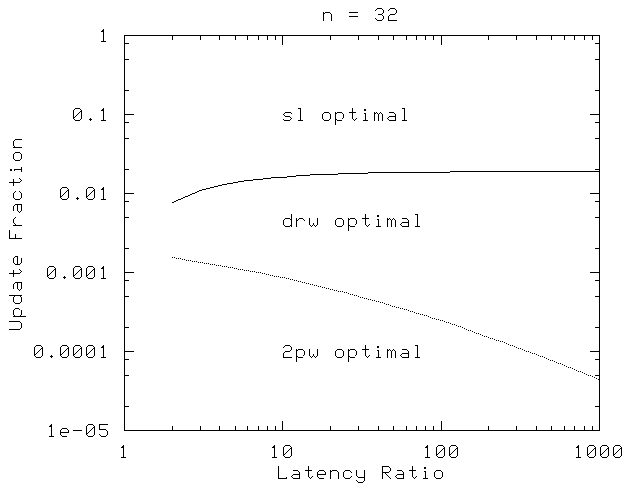

Figure 15: *Worst-Case Two-phase Update Breakevens (Key: Section 5.1)*

Figure 15 shows breakevens for small *l*, so that each update bears the full cost of detecting a grace period. There are three distinct regimes where each of simple spinlock, reader-writer spinlock, and two-phase update are optimal. Again, given sufficiently low values of the update fraction, two-phase update is optimal. However, the larger the latency ratio *r*, the smaller the update fraction must be for two-phase update to be optimal. Given the historically increasing latency-ratio trends, Figure 15 seems to indicate that use of two-phase update will become less advantageous as time goes on. But this conclusion is unwarranted, because: (1) the movement towards placing multiple CPUs on a single silicon die will likely slow the increase of latency ratios, (2) as noted previously, there are many situations with extremely low update fractions, and

(3) this is the worst case for two-phase update: more advantageous situations are shown below.

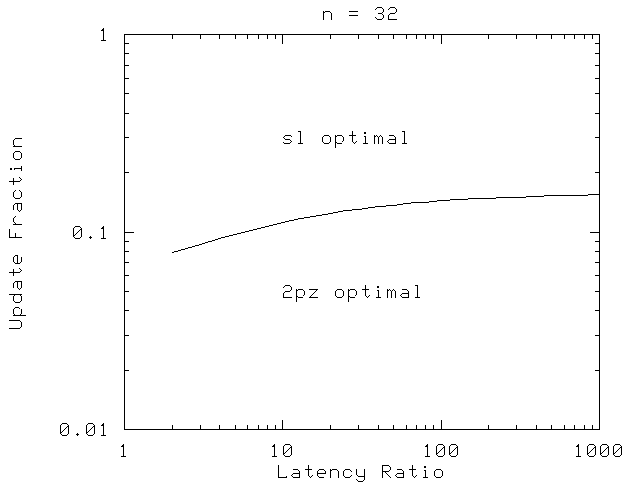

Figure 16: *l=1 Two-phase Update Breakevens (Key: Section 5.1)*

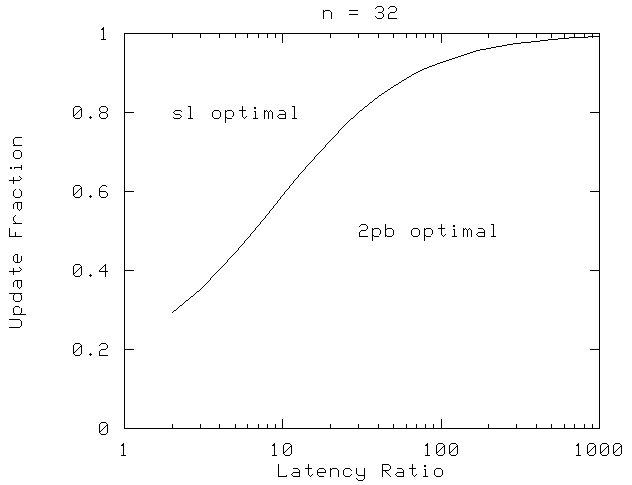

Figure 17: *Best-Case Two-phase Update Breakevens (Key: Section 5.1)*

Figure 16 shows breakevens for *l=1*, for which there is some amortization of the cost of grace-period detection over multiple updates. This amortization results in reader-writer spinlock never being optimal, since, unlike two-phase update, reader-writer spinlock cannot amortize the cost of write-acquisition over multiple updates. The breakeven update fraction for simple spinlock and two-phase update actually increases with increasing latency ratio. This is because simple spinlock pays a large memory-latency penalty for both reading and updating, while two-phase update pays the penalty only for updating. Therefore, increasing the latency ratio puts simple spinlock at a comparative disadvantage.

Figure 17 shows breakevens for large *l*, so that the per-update cost of detecting grace periods is minimized (note the linear y-axis). Again, reader-writer spinlock's inability to amortize write-side lock acquisition over multiple updates means that it is never optimal. Again, the breakeven between simple spinlock and two-phase update increases with increasing latency ratio, but this time quite sharply. This results in two-phase update being optimal almost everywhere.

5.4 Update-Side Locking

Note finally that all of these costs and breakevens assume that the update-side processing for two-phase update is guarded by a simple spinlock. In cases where the update-side processing may use a more aggressive locking design (for example, if only one thread does updates), two-phase update will have a greater performance advantage.

6 Measured Comparison

The following sections present measurements of two-phase update's performance and complexity, and compare them to those of traditional locking.

6.1 Performance

Data shown in this section was collected on a Sequent NUMA-Q machine with 32 Xeon CPUs, each running at 450 MHz [Lovett96].

The results for simple spin-lock shown in Figure 18 show good agreement between the analytic model and measurements taken on real hardware. Note that there are

small but measurable deviations both for smaller and for larger numbers of quads. The deviations at the low end are due in part to hardware optimizations and speculative execution, neither of which are accounted for in the model. The deviations at the high end are due in part to the increased cost of invalidation operations on the hardware under test for larger numbers of CPUs. Although it is possible to design more complex locking primitives that do not suffer from these deviations, such designs are beyond the scope of this paper.

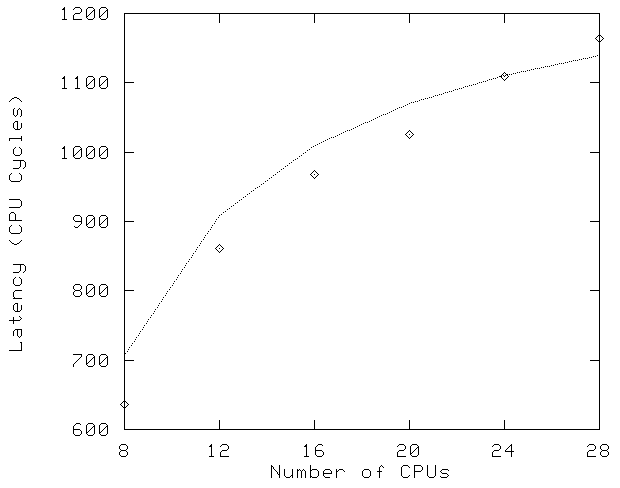

Figure 18: *Measured vs. Analytic Latencies for Simple Spin-Lock*

The results for distributed reader-writer spin-lock shown in Figure 19 also show good agreement between the analytic model and real hardware. Deviations are less apparent than for simple spin-lock because of the log-scale latency axis used to accommodate the wide range of latencies measured for reader-writer spin-lock.

Measured and analytic results for two-phase update also show good agreement, but only for older Pentium CPUs that do not feature speculative and out-of-order execution. The very short and low-overhead code segments implementing two-phase update make it impossible to accurately measure the overhead of individual two-phase update operations at low levels of contention on modern speculative multi-issue CPUs.

Instead, we measured the aggregate overhead of large numbers of operations at high levels of contention for simple spin-lock, reader-writer spin-lock, and two-phase update. This approach allows accurate bulk-measurement techniques to be applied. However, since the analytic results assume low contention, these measurements can only be compared to each other, not to the analytic results. The good agreement of individual-operation measurements on older CPUs indicates that the analytic results are a reliable guide for locking design.

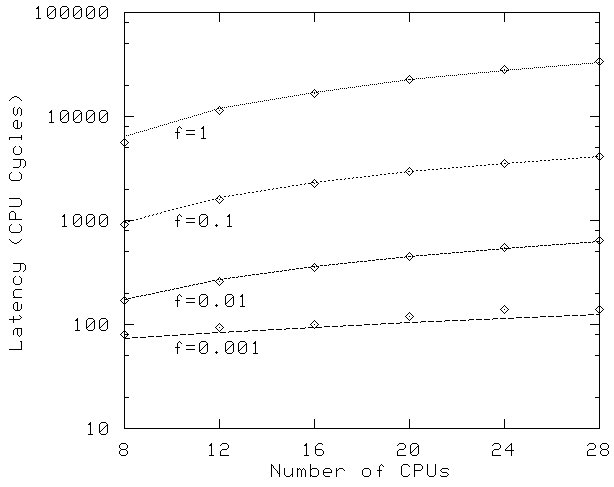

Figure 19: *Measured vs. Analytic Latencies for Reader-Writer Spin-Lock*

Figure 20 shows the expected results for simple spin-lock at high contention. The sharp drop in system-wide critical sections per microsecond between four and eight CPUs is due to the greater latency of remote-memory accesses. It is possible to create spin-locks that behave much better under high contention, but these are beyond the scope of this paper.

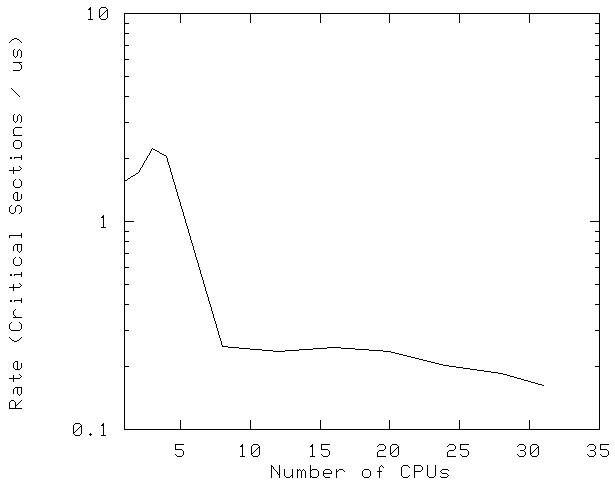

Figure 20: *Simple Spin-Lock at High Contention*

Figure 21 shows that distributed reader-writer spin-locks produce much greater levels of critical-section throughput under high contention, but only if read-side acquisitions dominate. The uppermost trace in this figure corresponds to $f=10^{-6}$, and each lower trace corresponds to an order-of-magnitude increase in *f*, up to $f=0.1$ in the lowermost trace. Note that the traces for 10^{-6} and 10^{-5} are almost overlapping. The erratic nature of the $f=0.1$ and $f=0.01$ traces is due to extreme write-side contention, which results in "beating" interactions between the locking algorithm and the cache-coherence protocol.

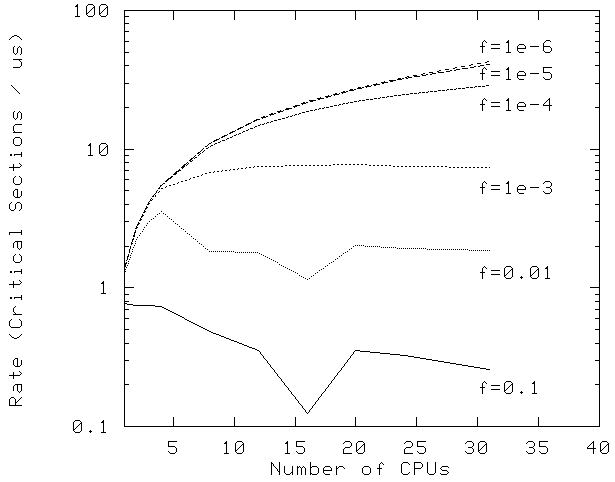

Figure 21: *Reader-Writer Spin-Lock at High Contention*

Figure 22 shows that two-phase update enjoys linear scaling with increasing numbers of CPUs even under high contention, independent of the value of *f*. Again, the lowermost trace corresponds to $f=0.1$, with each higher trace corresponding to an order-of-magnitude decrease in *f*. The traces for $f=10^{-4}$ through $f=10^{-6}$ are overlapping in the figure. Even at moderately low values of *f*, two-phase update achieves more than an order of magnitude more critical sections per second than does reader-writer spin-lock, and many orders of magnitude greater than does simple spin-lock.

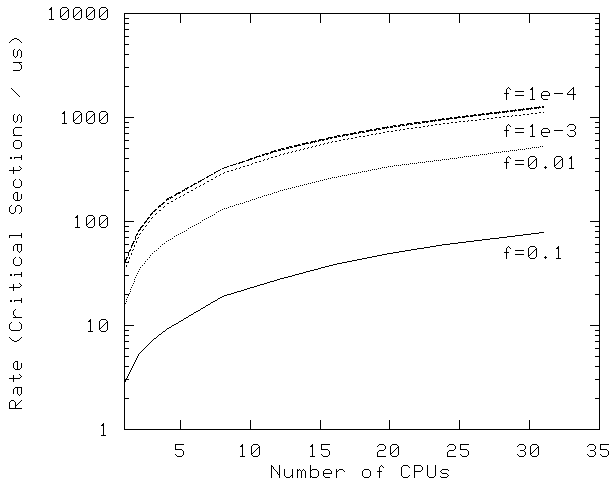

Figure 22: *Two-phase Update at High Contention*

However, it must be noted that high levels of contention result in high values of λ, which allows the overhead of grace-period detection to be amortized over many requests.

6.2 Complexity

This section compares the complexity of hierarchical-lock and two-phase-update algorithms for search and deletion in a doubly linked list. The requirements are as follows: 1) allow parallel operation on different elements of the list, 2) prevent readers from seeing stale data.

Figure 23 shows the data-structure layout of a list element. The last two fields are used only by the two-phase-update algorithms. The "kd" field is used by the

kmem_deferred_free() primitive that keeps a list of blocks of memory awaiting a deferred free. Each element on this list of blocks will be freed after all ongoing operations complete.

```
1  struct element {
2      struct element *next;
3      struct element *prev;
4      spinlock_t element_lock;
5      int key;
6      int data;
7      int deleted;  /* 2-phase only... */
8      struct kmem_defer_item kd;  /* " " */
9  };
```

Figure 23: *List Element Data Structure*

Figure 24 shows a traditional hierarchical-locking search algorithm. The global "list_lock" lock is used to protect the list while searching it, and is acquired on line 4 and released on lines 9 and 14. The loop covering lines 6 through 13 searches the list, and if line 7 finds a match, it acquires the per-element lock on line 8, releases the global lock on line 9 (after which operations on different elements in the list may be processed in parallel), and returns a pointer to the element located on line 10. If no matching element is found, line 14 releases the global lock and line 15 returns a NULL pointer.

```
1  struct element *search(int key)
2  {
3      struct element *p;
4      spin_lock(&list_lock);
5      p = head->next;
6      while (p != head) {
7          if (p->key == key) {
8              spin_lock(&p->element_lock);
9              spin_unlock(&list_lock);
10             return (p);
11         }
12         p = p->next;
13     }
14     spin_unlock(&list_lock);
15     return ((struct element *)NULL);
16 }
```

Figure 24: *Traditional Hierarchical-Lock Search*

Figure 25 shows a traditional hierarchical-locking deletion algorithm. Again, the global "list_lock" lock is used to protect the list. However, the per-element lock is already held, so line 4 cannot unconditionally acquire the lock, since this could deadlock with the search algorithm. Instead, line 4 conditionally acquires the lock. If the attempted lock acquisition fails, then lines 5 through 9 record the address, release the lock, and redo the search in order to acquire the locks in the correct order. Note that the address *must* be captured *before* releasing the lock, since it is not safe to dereference pointer "p" after the lock has been released.

Once both locks have been acquired, lines 11 and 12 unlink the element from the list, lines 13 and 14 release the locks, and line 15 frees up the deleted element.

```
1  void delete(struct element *p)
2  {
3      int key;
4      if (!spin_trylock(&list_lock)) {
5          key = p->key;
6          spin_unlock(&p->element_lock);
7          if ((p = search(key, 0)) == NULL) {
8              return;
9          }
10     }
11     p->next->prev = p->prev;
12     p->prev->next = p->next;
13     spin_unlock(&p->element_lock);
14     spin_unlock(&list_lock);
15     kfree(p);
16 }
```

Figure 25: *Traditional Hierarchical-Lock Deletion*

Figure 26 shows a two-phase-update search algorithm. This algorithm is very similar to its hierarchically locked counterpart. One difference is that the "list_lock" acquisitions and releases have been deleted (these would have to be replaced with operations to suppress preemption in preemptive environments). Another difference is that the "if" statement on line 8 must check the "deleted" flag in order to prevent access to stale data.

```
1  struct element *search(int key)
2  {
3      struct element *p;
4      p = head->next;
5      while (p != head) {
6          if (p->key == key) {
7              spin_lock(&p->element_lock);
8              if (!p->deleted) {
9                  return (p);
10             }
11             spin_unlock(&p->element_lock);
12         }
13         p = p->next;
14     }
15     return ((struct element *)NULL);
16 }
```

Figure 26: *Two-Phase Update Search*

Figure 27 shows a two-phase-update deletion algorithm. Since the search algorithm need only acquire the per-element lock, the delete algorithm is free to acquire the "list_lock" lock while holding the per-element lock. Thus, there is no need for the conditional locking and deadlock-avoidance code that is in the hierarchical locking deletion algorithm. A key change is the replacement of "kfree()"by "kmem_deferred_free()". The "kmem_deferred_free()" primitive adds the specified block of memory to a list. The elements of this list are freed only after all ongoing operations complete. This prevents any races with the search algorithm: the element is actually freed only after all ongoing searches complete.

```
1  void delete(struct element *p)
2  {
3      int addr;
4      spin_lock(&list_lock);
5      p->deleted = 1;
6      p->next->prev = p->prev;
7      p->prev->next = p->next;
8      spin_unlock(&p->element_lock);
9      spin_unlock(&list_lock);
10     kmem_deferred_free(p);
11 }
```

Figure 27: *Two-Phase-Update Deletion*

The search algorithms are both the same size: 16 lines. However, two-phase-update deletion is only 11 lines compared to 16 lines for hierarchical-locking deletion. This illustrates the reductions in complexity that two-phase update can provide.

Use of two-phase also reduces testing effort. To see this, compare the traditional and two-phase deletion algorithms. Testing lines 5 through 9 of the traditional algorithm is difficult, because this code is reached only in response to a low-probability race between deletion and search. Testing line 8 is even more difficult, because a second race between a pair of deletions is required on top of the original race.

In contrast, the two-phase-update version of deletion is straight-line code, which is easily tested. Any test that reaches any part of this code will reach it all.

7 Use of Two-Phase Update

A form of two-phase update named read-copy update has been in production use within Sequent's DYNIX/ptx kernel since 1993. DYNIX/ptx is a highly scalable non-preemptive Unix kernel supporting up to 64 CPUs that is primarily used for high-end database servers.

Two-phase update is used as shown below. The most common use is maintaining linked data structures as described in Section 4.

1. Distributed lock manager: recovery, lists of callbacks used to report completions and error conditions to user processes, and lists of server and client lock data structures. This subsystem inspired two-phase update.
2. TCP/IP: routing tables, interface tables, and protocol-control-block lists.
3. Storage-area network (SAN): routing tables and error-injection tables (used for stress testing).
4. Clustered journaling file system: in-core inode lists and distributed-locking data structures.

5. Lock-contention measurement: B* tree used to map from spinlock addresses to the corresponding measurement data (since the spinlocks are only one byte in size, it is not possible to maintain a pointer within each spinlock to the corresponding measurement data).

6. Application regions manager (a workload-management subsystem): maintains lists of regions into which processes may be confined.

7. Process management: per-process system-call tables as well as the multi-processor trace data structures used to support user-level debugging of multi-threaded processes.

8. LAN drivers: resolve races between shutting down a LAN device and packets being received by that device.

The Tornado [Gamsa99] and K42 research operating systems independently developed a form of two-phase update, which is used to provide existence guarantees throughout these operating system. These existence guarantees simplify handling of races between use of a data structure and its deletion.

8 Future Work

Two-phase update has been used extensively on a variety of data structures. However, more work is needed to determine the set of modifications that can efficiently be cast into two-phase-update form.

Two-phase update has seen production use primarily in non-preemptive kernel environments. Future work includes use in preemptive environments, such as user-level applications. Additional work is also required to determine whether two-phase update is useful in a distributed computing environment.

9 Conclusions

In restricted but commonly occurring situations, two-phase update can significantly reduce complexity while simultaneously improving performance and scaling. It accomplishes this feat by exploiting the event-driven nature of many parallel software systems, such as operating system kernels, which continually perform many small, quickly completed operations.

The key concept of two-phase update is splitting destructive modifications into two phases: (1) performing updates that prevent new operations from seeing old state, while allowing old operations to proceed on the old state, then (2) completing the modification. The second phase is initiated only after all operations that might be accessing old state have completed. This deferred initiation of the second phase eliminates the possibility of races with ongoing operations, since any operations that can access the old state have been allowed to complete. The benefits of two-phase update stem directly from the elimination of these races.

Traditional techniques (such as locking or wait-free synchronization) are used to coordinate concurrent two-phase-update *modifications*, but *read-only* do not need to use any sort of synchronization mechanism. These read-only operations may instead be coded as if they were executing in a single-threaded environment, without even the possibility of interrupts. Dispensing with synchronization primitives can greatly increase performance in read-mostly situations, and can greatly simplify deadlock avoidance and recovery. In addition, two-phase update may be used in conjunction with wait-free synchronization to remove the latter's restrictions on reuse of memory.

Both analytical and measured comparisons demonstrate the performance and complexity-reduction benefits of two-phase update in the restricted, but commonly occurring,

situations to which it is applicable. Two-phase update has been in production use since 1993.

10 Acknowledgements

I owe thanks to Stuart Friedberg, Doug Miller, Jan-Simon Pendry, Chandrasekhar Pulamarasetti, Jay Vosburgh, Dave Wolfe, and Peter Strazdins for their willingness to try out two-phase update, to Dipankar Sarma, Andi Kleen, Keith Owens, and Maneesh Soni for looking at two-phase update from a Linux perspective, to Ken Dove, Brent Kingsbury, John Walpole, James Hook, and to Phil Krueger and his Base-OS Reading Group for many helpful discussions, to Dylan McNamee and Andrew Black for noting the connection between two-phase update and wait-free synchronization, and to Chris Lattner for discussions on the relation between compile-time dependency analysis and two-phase update. I owe special thanks to Orran Kreiger for many valuable discussions that helped clarify the relation between two-phase update and event-driven software systems, and to Jack Slingwine for the collaboration that lead to the first implementation of read-copy update [McK98a]. I am indebted to Kevin Closson for machine time used to collect measured data. I am grateful to Leslie Swanson, Daniel Frye, and Dale Goebel for their support of this work.

This work was done with the aid of Macsyma, a large symbolic manipulation program developed at the MIT Laboratory for Computer Science and supported from 1975 to 1983 by the National Aeronautics and Space Administration under grant NSG 1323, by the Office of Naval Research under grant N00014-77-C-0641, by the U. S. Department of Energy under grant ET-78-C-02-4687, and by the U. S. Air Force under grant F49620-79-C-020, between 1982 and 1992 by Symbolics, Inc. of Burlington Mass., and since 1992 by Macsyma, Inc. of Arlington, Mass. Macsyma is a registered trademark of Macsyma, Inc.

11 References

[Adams91] G. R. Adams. Concurrent Programming, Principles, and Practices, Benjamin Cummins, 1991.

[Ander95] J. H. Anderson and Mark Moir. “Universal constructions for large objects”, International Workshop on Distributed Algorithms, 1995

[Bargh91] N. S. Barghouti and G. E. Kaiser. “Concurrency control in advanced database applications”, ACM Computing Surveys, 23(3), pp. 269-318, September, 1991.

[Burger96] D. Burger, J. R. Goodman, and A. Kägi. “Memory bandwidth limitations of future microprocessors”, Proceedings of the 23rd International Symposium on Computer Architecture, pp. 78-89, May, 1996.

[Court71] P. J. Courtois, F. Heymans, and D. L. Parnas. “Concurrent control with ‘readers’ and ‘writers’”. Communications of the ACM, 14(10), pages 667-8, Oct. 1971.

[Gamsa99] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. “Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System”, Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, LA, February 1999.

[Hennes91] J. L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. IEEE Computer, 24(9), pp. 18-28, September, 1991.

[Herlihy93] M. Herlihy. Implementing highly concurrent data objects, ACM Transactions on

Programming Languages and Systems, 15(5), pp. 745-770, November, 1993.

[Hsieh91] W. C. Hsieh & W. E. Weihl, "Scalable Reader-Writer Locks for Parallel Systems", Tech report MIT/LCS/TR-521, November, 1991

[Jacob93] V. Jacobson. “Avoid read-side locking via delayed free”, private communication, September, 1993.

[Kung80] H. T. Kung and Q. Lehman. “Concurrent manipulation of binary search trees”, ACM Trans. on Database Systems, 5(3), pp. 354-382, September, 1980.

[Lovett96] T. Lovett and R. Clapp. “STiNG: A CC-NUMA computer system for the commercial marketplace” Proceedings of the 23rd International Symposium on Computer Architecture, pp. 308-317, May 1996.

[Manber84] U. Manber and R. E. Ladner. “Concurrency control in a dynamic search structure”, ACM Trans. on Database Systems, 9(3), pp. 439-455, September, 1984.

[McK98a] P. E. McKenney and J. D. Slingwine. “Read-copy update: using execution history to solve concurrency problems”, Parallel and Distributed Computing and Systems, October 1998.

[McK98b] P. E. McKenney. “Implementation and performance of read-copy update”, Sequent TR-SQNT-98-PEM-4.0, March 1998.

[McK99] P. E. McKenney. “Practical performance estimation on shared-memory multiprocessors”, Parallel and Distributed Computing and Systems, November 1999.

[Michael98] M. M. Michael and M. L. Scott. “Non-Blocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors”, JDPC, May 1998.

[JMC91] J. M. Mellor-Crummey and M. L. Scott. “Scalable reader-writer synchronization for shared-memory multiprocessors”, Proceedings of the Third PPOPP, Williamsburg, VA, pp. 106-113, April, 1991.

[Pugh90] W. Pugh. “Concurrent Maintenance of Skip Lists”, Department of Computer Science, University of Maryland, CS-TR-2222.1, June, 1990.

[Russell01] R. Russell. “synchronize_kernel()”, www.uwsg.indiana.edu/hypermail/linux/kernel/0103.2/0424.html, March 2001.

[Stone91] H. S. Stone and J. Cocke. “Computer architecture in the 1990s”. IEEE Computer, 24(9), pages 30-38, September 1991.

[Sarma01] D. Sarma. “Read-copy update mutual exclusion for Linux”, linux-kernel list, http://lse.sourceforge.net/locking/rclock.html, February 2001.

A Derivation of Analytic Results

This section derives the analytical results for two-phase update presented in Section 5, using the symbols shown in Table 2. It also presents results for other locking primitives derived elsewhere [McK99]. The analysis assumes low levels of contention and high memory latencies. These assumptions allow performance to be estimated by counting memory references and estimating latencies based on the probable prior state of the corresponding cache lines [McK99].

A.1 Derivation for Two-phase Update

There are four components to the overhead of an efficient callback-based implementation of two-phase update [McK98b, Sarma01]:

1. per-hardclock() costs. These are incurred on every execution of the per-CPU scheduling-clock interrupt.
2. per-grace-period costs.
3. per-batch costs. These are incurred during each two-phase batch. Per-batch costs are incurred only by CPUs that have a batch of updates during a given grace period. These costs are amortized over callbacks making up that batch.
4. per-callback costs. These are incurred for every two-phase callback.

Details of the derivations may be found in a companion paper and technical report [McK98b, McK99]. See Section 5.1 for an overview of the nomenclature and methodology.

(1), (2), The cost of (2) is due to accessing the time at which the grace period started by multiple CPUs, taking a snapshot of and checking the per-CPU counters, end-of-grace-period cleanup, and registering each CPU's passage through a quiescent state.

(3), and (4) give the two-phase overhead incurred for each of these four components: per hardclock(), per grace period, per batch, and per callback, respectively.

$$C_h = nmt_c + 3nmt_f \tag{1}$$

The first term of (1) stems from the fact that each CPU must read a clock on each hardclock() invocation, and the second term from the fact that each CPU must check a pair of per-CPU queues and a bitmask on each hardclock() invocation in order to determine that there is nothing for it to do.

$$C_g = \begin{bmatrix} \left(3n + 2nm - m\right)t_s + \\ \left(2nm + m - 1\right)t_m + \\ \left(7nm + 1\right)t_f \end{bmatrix} \tag{2}$$

The cost of (2) is due to accessing the time at which the grace period started by multiple CPUs, taking a snapshot of and checking the per-CPU counters, end-of-grace-period cleanup, and registering each CPU's passage through a quiescent state.

$$C_b = 3t_s \tag{3}$$

The cost of (3) is due to acquiring a lock to guard the global state, initiating processing of the callbacks, and checking a global count of the number of grace periods. Once this global count has increased by two, the grace period will have ended. The overhead of advancing the callbacks from the "next grace period" to the "current grace period" and finally to the "finished with grace period" lists would be included in this cost, but the

cachelines represented by these operations are loaded into the CPU's L1 cache by the hardclock() invocations, so that this overhead is insignificant.

$$C_c = 7t_f \tag{4}$$

The cost of (4) is due to initializing the callback structure, counting the fact that a callback was registered, adding the callback to the per-CPU "next grace period" list (two accesses required), checking to see if the current CPU is in the process of going offline, removing the callback from the "finished with grace period" list, and counting the fact that a callback was invoked.

The best-case incremental cost of a two-phase callback, given that at least one other callback is a member of the same batch, is just C_c, or $7t_f$. Note that this time period is shorter than may be measured accurately on modern speculative and multi-issue CPUs.

The worst-case cost of an isolated callback is m times the per-hardclock() cost plus the sum of the rest of the costs, as shown in (5):

$$C_{wc} = \begin{bmatrix} \left(3n + 2nm - m + 3\right) t_s + \\ \left(2nm + m - 1\right) t_m + \\ \left(3nm^2 + 7nm + 8\right) t_f + \\ nm^2 t_c \end{bmatrix} \tag{5}$$

Note that this worst case assumes that at most one CPU per quad passes through its first quiescent state for the current grace period during a given period between hardclock() invocations. In typical commercial workloads, CPUs will pass through several quiescent states between hardclock() invocations, so that the m^2 factors in (5) would be replaced by m, significantly reducing the cost.

Typical costs may be computed assuming a system-wide Poisson-distributed inter-arrival rate of l per grace period, as shown in (6).

$$C_{typ} = \frac{\sum_{k=1}^{\infty} \frac{l^k e^{-l}}{k!} C_k}{1 - e^{-l}} \tag{6}$$

Here $(l^k e^{-l})/k!$ is the Poisson-distributed probability that k callbacks are registered during a given grace period if on average l of them arrive per grace period. Note that the 0^{th} term of the Poisson distribution is omitted, since there is no two-phase overhead if there are no two-phase arrivals. The division by $1-e^{-l}$ normalizes for this omission. The quantity C_k is defined as follows:

$$C_k = \frac{C_h + C_g + N_b\left(k\right) C_b + kC_c}{k} \tag{7}$$

This definition states that we pay the per-hardclock() and per-grace period overhead unconditionally, that we pay the per-batch overhead for each of $N_b(k)$ batches, and that we pay per-callback overhead for each callback.

The expected number of batches $N_b(k)$ is given by the well-known solution to the occupancy problem:

$$N_b(k) = nm\left(1-\left(1-\frac{1}{nm}\right)^k\right) \quad \textbf{(8)}$$

This is just the number of CPUs expected to have batches given *nm* CPUs and *k* two-phase updates.

Substituting (7) and (8) into (6) and substituting (1), (2), The cost of (2) is due to accessing the time at which the grace period started by multiple CPUs, taking a snapshot of and checking the per-CPU counters, end-of-grace-period cleanup, and registering each CPU's passage through a quiescent state.

(3), and (4) into the result, normalizing t_s, t_m, and t_f in terms of *r*, then multiplying by the update fraction *f* yields the desired expression for the typical cost:

$$\frac{f}{e^l-1}\sum_{k=1}^{\infty}\frac{l^k\left[\begin{array}{c}\left(3n+5nm-m\right)r-\\ 3nm\left(1-\frac{1}{nm}\right)^k r+\left(2nm+m-1\right)\sqrt{r}+\\ \left(10nm+7k+1\right)+nmt_c\end{array}\right]}{k!k} \quad \textbf{(9)}$$

A.2 Equation for Simple Spinlock

The overhead for simple spinlock [McK99] is as follows:

$$\frac{\left(n-1\right)mr+\left(m-1\right)\sqrt{r}+\left(nm+1\right)}{nm} \quad \textbf{(10)}$$

A.3 Equation for Reader-Writer Spinlock

The overhead for Hsieh and Weihl reader-writer spinlock [Hsieh91, McK99] is as follows:

$$\frac{\left[\begin{array}{c}\left[\begin{array}{c}\left[\begin{array}{c}n^3m^3-\left(m+1\right)n^2m^2+\\ \left(m-1\right)nm+m\end{array}\right]f^2+\\ \left(2n^2m^2-\left(2m-1\right)nm-m\right)f\end{array}\right]r+\\ \left[\begin{array}{c}\left[\begin{array}{c}\left(m-1\right)n^2m^2-\\ \left(m-1\right)nm-m+1\end{array}\right]f^2+\\ \left(2\left(m-1\right)nm+m-1\right)f\end{array}\right]\sqrt{r}+\\ \left[\begin{array}{c}\left(n^3m^3-1\right)f^2+\\ \left(2n^2m^2-nm+1\right)f+2nm\end{array}\right]\end{array}\right]}{\left(nm-1\right)nmf+nm} \quad \textbf{(11)`}$$