

Practical Concerns for Scalable Synchronization

(author information elided)

Draft of 2005/03/25 12:52

Abstract

With the advent of multi-core chips and simultaneous multithreading technologies, high degrees of parallelism will soon become the norm for both small- and large-scale systems. Operating systems for such highly parallel environments require efficient synchronization. Unfortunately, the ever-increasing overhead of synchronization instructions on modern CPUs has made such efficiency difficult to achieve.

This paper evaluates the performance of synchronization strategies on modern CPU architectures. We show that synchronization instructions can be a thousand times more expensive than normal instructions and that formerly scalable synchronization strategies now perform very poorly. We then evaluate several state-of-the-art solutions that combine copy-based update and deferred reclamation to allow lock-free concurrent reading. These solutions exhibit different update management and reclamation strategies, each of which performs well, but offers a unique trade-off between performance, memory consumption and complexity. We present an experimental evaluation of these strategies and discuss the impact of potential future changes in CPU architecture.

1 Introduction

The quest for ever-higher processing throughput has led to increasingly parallel architectures. Current hardware trends include “simultaneous multi-threading” on a single processor as well as multiple processors on a single “multi-core” chip. When combined with ongoing advances in shared memory multiprocessor design these trends will lead to architectures with very high degrees of parallelism. While these architectures promise to meet the performance demands of enterprise applications and data centers, a substantial obstacle to realizing their performance potential is the limited scalability of today’s operating systems. The current excitement about virtualization as a means of running several operating system instances simultaneously on a single hardware platform is, at least in part, a symptom of this problem [6].

While some commercial operating systems, such as SGI’s IRIX, Sun’s Solaris, Sequent’s (now IBM’s) DYNIX/ptx, and IBM’s AIX, have scaled to execute on many tens or even hundreds of CPUs by using aggressive data partitioning, fine-grain locking, and function shipping, this scalability has come at the cost of increased complexity and decreased performance. Systems based on virtualization face these same problems in the hypervisor and do not completely remove them from the guest operating system which must still scale well if it must run on large scale virtual machines.

Central to the scalability vs. performance vs. complexity trade-off are the synchronization strategies used to maintain the consistency of operating system data structures. Synchronization limits performance and scalability not only due to contention for critical sections but also due to the overhead it imposes on processor

throughput in contention-free scenarios. This overhead takes the form of synchronization instruction complexity, and pipeline stalls due to memory and inter-processor communication latency. These overheads are architecture-dependent, and although they used to be unimportant, developments in processor architecture have made them progressively more problematic. The result is that synchronization strategies designed with these overheads in mind perform and scale dramatically better than those that ignore them. Not surprisingly, synchronization strategies in production operating systems have evolved significantly over recent years in response to these pressures [7, 34].

Today’s production systems use a bewildering assortment of synchronization strategies that make different trade-offs between performance, scalability and complexity. Often these trade-offs are highly dependent on assumptions about processor architecture characteristics, such as the memory latency, memory consistency semantics and the availability and relative cost of specific synchronization instructions. Performance also tends to depend critically on the relative mix of reads and writes in the workload.

Two of the most scalable synchronization strategies currently in use in production systems are Hazard Pointers [39] and Read Copy Update (RCU) [32]. Both of these strategies improve scalability by forcing updaters to create and modify new versions of objects instead of updating the old versions in place. The major advantage of this approach is that readers run lock-free, concurrently with updaters, leading to near optimal performance in the read path. There are two main disadvantages however. The first is that the memory associated with old versions must be reclaimed. The second is that readers must tolerate the possibility of reading stale data. Fortunately, there are several known strategies for deferring memory reclamation and performing it efficiently in batches, and there are many situations in which readers can tolerate stale data. For example, there are almost 300 uses of RCU-related primitives in the Linux 2.6.11 kernel. Hence, deferred-reclamation strategies have widespread practical application.

RCU and Hazard Pointers differ in their strategies for memory reclamation and synchronization among updaters. RCU typically uses a lock-based update strategy and quiescent-state-based reclamation, whereas Hazard Pointers are typically used with non-blocking synchronization and explicit (hazard pointer-based) memory reclamation. It is worth noting, however, that update strategy and reclamation strategy are orthogonal choices. We will explore this orthogonality and the performance implications of different choices later in this paper.

First, Section 2 presents an overview of synchronization strategies that have been used in operating systems. Then Section 3 quantifies the instruction-level overhead of synchronization primitives on modern CPUs and evaluates the effect it has had on the performance of some of these strategies. The best performing strategies, based on

copy-based update with concurrent lock-free reading and deferred reclamation, are examined in more detail in Section 4. Section 5 discusses the impact of future architectural trends, and Section 6 concludes the paper.

2 Background

Early synchronization strategies, based on coarse grained code locking, limit scalability due to lock contention. A variety of strategies have been proposed to reduce lock contention, including queued locks [1, 52], fine-granularity locking, data locking [4, 12, 13, 25, 31, 46], partitioning [6, 14, 44, 49, 50], data ownership [32], asymmetric reader-writer locking [9, 36], and numerous flavors of non-blocking synchronization [11, 17, 18, 19, 20, 21, 37, 38, 39, 47, 51]. Queued locks reduce contention by allowing each competing thread to spin on a separate lock location. Fine-granularity locking reduces contention by splitting large critical sections into multiple small ones. However, as critical section size reduces, the overhead of synchronization mechanisms becomes progressively more important. Spinlocks are one way of protecting small critical sections, but they require careful design to avoid introducing memory contention [32]. Even with low-overhead spinlocks, frequently executed critical sections limit scalability.

Data locking improves scalability by carefully partitioning data structures and associating distinct locks with each partition. If data is partitioned such that each partition is accessed by a single CPU, significant performance gains can be achieved by avoiding the need to shuttle locks among CPU caches. However, in cases where partitioning is mediated by a virtual machine, care must be taken to avoid preempting a given operating system while it is holding a lock [48]. Per-CPU reader-writer locking [2, 24] is an asymmetric approach that achieves a similar effect by assigning a distinct lock to each CPU and requiring reading code to acquire its own CPU's lock, and writing code to acquire all CPU's locks. This strategy is particularly effective for read-mostly scenarios, which tend to be common in operating systems, because it only incurs memory latency and contention for writers. In the Linux 2.4 kernel this technique is referred to as `brlock`.

In parallel with the evolution of scalable locking strategies, there has been extensive work on non-blocking synchronization (NBS). NBS strategies achieve synchronization by optimistically computing new updates and using atomic instructions, such as compare-and-swap (CAS) and load-locked/store-conditional (LL/SC), to atomically replace old values with new, updated values. In the face of conflicts, updates must be retried, and helper mechanisms are required to ensure progress and maintain adequate performance. These mechanisms usually take the form of complex, application-specific algorithms, and may include strategies such as randomized exponential back-off [18].

Herlihy defines various forms of NBS, including wait-free, lock-free and obstruction-free synchronization [20]. A synchronization strategy is wait-free if it ensures that every thread will continue to make progress in the face of arbitrary delays of other threads, lock-free if it ensures that some thread always makes progress, and obstruction-free if it ensures the progress of any thread that eventually executes in isolation. The prospect of being free from problems such as deadlock, thread starvation, scheduling convoys, and priority inversion has lured researchers to use NBS strategies in operating system kernels, such as Synthesis [41], the Cache Kernel [8],

Exokernel [10], and K42 [32].

The specialized stack and queue implementations of the Synthesis kernel were an early use of NBS strategies in operating systems [29, 30]. Some of the Synthesis implementations used only load and store instructions, but depended on sequentially consistent memory, while others required an atomic CAS instruction. Later, Bershad showed how CAS could be simulated on architectures that did not support it directly [5].

Greenwald and Cheriton proposed a more systematic strategy for implementing non-blocking data structures based on the use of an atomic double-compare-and-swap (DCAS) instruction [15]. By associating version numbers with each data structure and using DCAS to atomically compare and update the data structure and its version number, they were able to detect and roll back from concurrent updates to complex data structures. Unfortunately, a DCAS instruction is rarely available in the instruction sets of modern CPUs. Software implementations of multiple compare-and-swap (MCAS) have been proposed, but are still a topic for research [11].

Herlihy proposed a methodology for creating lock-free and wait-free implementations of concurrent objects using LL/SC [18]. He also proposed a similar approach based on CAS, but it resulted in increased complexity and worse performance [18]. While many NBS algorithms have been developed, experience has shown that building practical NBS algorithms directly from available primitives is a complex task. For this reason, there is currently much interest in higher-level abstractions for simplifying the task. One such abstraction is transactional memory to group operations into transactions that either commit atomically, or abort without being observed. Transactional memory can be implemented in hardware [22] or software [11, 45].

Since NBS updates to complex data structures typically involve creating new versions of elements, updating the new version in place, and then atomically replacing pointers to the old version with pointers to the new one, the problem of how, or more specifically when, to reclaim the memory of the old versions must be addressed. The problem is that updates may occur concurrently with readers traversing the same data structure. In this case, readers can be left holding references to old versions. If the memory associated with these versions is reclaimed while the references are still in use, traversing the data structure may direct readers into the free pool, or elsewhere if the memory has already been reused. To avoid exposing readers to this "hijacking" danger, additional mechanisms are required.

One approach is to place the burden of checking for updates on the reader. The problem with this approach is the performance impact on the read path. At a minimum readers would either have to execute a LL/SC sequence on the pointer to the data, or two memory barriers to check a version number associated with the data. In the absence of performing these operations on every read of the data, additional memory management mechanisms are required.

An alternative approach is to never free memory. However, this is unacceptable for production systems. A more efficient approach is to use type-stable memory management (TSM) [15]. TSM ensures that readers can not be hijacked out of the data structure they are traversing by maintaining separate memory pools per data structure type. However, this too has proven to be unacceptable for production systems [39]. A better approach is to distinguish between logical and physical deletion of an object, and to delay the physical

deletion, until it is guaranteed that no reader continues to hold a reference to it. We use the term “deferred reclamation” to describe this class of memory management strategies.

An obvious technique for detecting whether any readers hold references to logically deleted data is to associate reference counts with the data. However, the manipulation of these reference counts requires synchronization operations in the read-path, which has a substantial impact on performance [39].

Fraser proposed an approach to deferred reclamation in which NBS operations are associated with epochs [11]. The idea is to delay the physical deletion of data logically deleted in epoch x until all threads in the system with access to that data are in an epoch later than x . This behavior is accomplished by having each thread maintain its own local epoch and participate in the maintenance of a system-wide global epoch. On every NBS operation a thread observes the global epoch. If it is later than the thread’s local epoch, the local epoch is advanced to the same value as the global epoch and the NBS operation proceeds. Each thread maintains a “limbo list” of logically deleted elements associated with the thread’s local epoch. These elements can be physically deleted once the thread has observed a global epoch greater than its own local epoch. Threads maintain a local counter of NBS operations associated with the current epoch and use it to determine when to attempt to advance the global epoch, which can only be advanced once every thread has observed its current value.

Epoch-based reclamation is safe, in the sense that it never reclaims memory prematurely, but it incurs overhead for maintaining a per-thread count of NBS operations per local epoch and a global count of the threads that have observed the global epoch. To ensure that the global epoch can be advanced and memory reclaimed, both readers and writers incur these overheads.

Michael proposed an approach to safe memory reclamation using hazard pointers [39]. Whenever a thread obtains a reference to a shared data object it inserts a hazard pointer for the object into its list of hazard pointers. When the reference is discarded, the hazard pointer is removed from the list. In order to physically delete an object, the hazard pointer lists of all threads in the system must be searched to ensure that there are no hazard pointers for that object. The overhead of this search can be amortized by deleting objects in batches. In this case, the batch size represents a trade-off between overhead and memory usage, and by keeping track of the number of logically deleted objects it is possible to impose a tight bound on memory usage. Hazard pointers are also a safe memory reclamation strategy, but they too incur overhead in the read-path, since readers must insert and delete hazard pointers using an algorithm that requires memory barriers on most modern CPUs.

Hazard pointers and epochs can be viewed as explicit mechanisms, manipulated by readers, to indicate when it is safe to physically delete objects. An alternative approach, called read-copy update (RCU), is used in the VM/XA [16], DYNIX/ptx [35], K42 [12], and Linux [33] operating systems. RCU *infers* safe deletion times by observing global system state. In order to understand this approach, it is necessary to recall that all synchronization mechanisms rely on conventions, which are enforced either manually or via software tools. For example, locking imposes the convention that threads are prohibited from holding references to shared data unless they also hold the corresponding lock. Similarly, NBS imposes the convention that threads use specific sequences of atomic

instructions to access and update shared data. The enclosing environment may impose additional conventions, for example, in non-preemptive operating-system kernels, threads are prohibited from switching context while holding a spinlock. Failure to observe this last convention risks a deadlock condition where the thread holding the lock is blocked and all CPUs are running threads that are spinning on this same lock.

RCU also imposes a coding convention, namely, that threads are prohibited from holding references to shared data while in “quiescent states”. Quiescent states are specific to a given environment, and the task of identifying a set of quiescent states appropriate for a given environment can be a significant challenge. However, a simple example of a useful quiescent state in an operating system kernel is a voluntary context switch. In this example, the RCU coding convention prohibits threads from holding references to shared data when blocking, since blocking results in voluntary context switches. Note that this coding convention is quite similar to the locking prohibition against blocking while holding a lock, which indicates that it might be successfully applied by practitioners who are familiar with spinlocks.

The RCU coding convention implies that once a given thread has been observed in a quiescent state after a given data item was removed from a shared data structure, then this thread can no longer be holding a reference to this data item. Because this thread is no longer holding a reference to this data item, the data item may be reclaimed without adversely affecting the thread.

Applying this same line of reasoning to all threads in the system, we conclude that the following procedure, analogous to Fraser’s use of epochs, may be used to remove and reclaim a data item: (1) remove the data item from the shared data structure, (2) wait until each thread has subsequently been observed in a quiescent state, and (3) reclaim the data item. This procedure is safe, because once all threads have been observed in a quiescent state, by convention none of them may hold a reference to the data item. Note that this procedure does not require readers to make use of any synchronization mechanisms, atomic instructions, or memory barriers whatsoever, allowing readers to attain near-ideal performance. Instead, readers must follow the RCU convention of refraining from entering a quiescent state (blocking, in this case) while traversing a shared data structure. Due to this use of quiescent states, RCU is termed a quiescent-state-based reclamation (QSBR) mechanism.

Any time period during which all threads have occupied a quiescent state is termed a “grace period”. After removing a data item from a shared data structure, a thread must wait for a grace period to elapse before reclaiming that data item. Implementations of the RCU infrastructure must contain a mechanism for detecting thread-local quiescent states as well as a mechanism for determining when a grace period has elapsed. This second mechanism is typically a barrier computation that takes quiescent state detection as input. Such an implementation must be extremely efficient, lest its overhead overwhelm the performance increase obtained by removing synchronization from readers. By definition, any time interval that includes a grace period is itself a grace period, and this property can be used to process reclamations in arbitrarily large batches, thereby amortizing the cost of grace-period detection over a large number of updates.

In the example above, we considered only voluntary context switch as a quiescent state. Other naturally occurring quiescent

states that have been used in operating systems include involuntary context switch, idle-loop execution, return to user mode, and thread termination [32], as well as explicit calls indicating quiescent states [3].

The discussion of RCU thus far has focused solely on synchronization between updaters and readers. Any algorithm making use of RCU must in addition use some synchronization mechanism to coordinate updates. It is interesting to note that for both RCU and hazard pointers, and, we believe, for any deferred-reclamation mechanism, updaters may be coordinated using locking, NBS, or any other synchronization mechanism. The Read-Copy-Update (RCU) strategy used in VM/XA, DYNIX/ptx, K42, and Linux is an example of quiescent-state-based deferred reclamation using lock-based writers, and K42 also uses RCU with NBS-based writers [32]. Although RCU is used in a variety of ways, a prominent RCU design pattern allows lock-based writers to create new versions of data objects instead of updating them in place, while concurrent readers continue to read old versions. Writers logically delete old versions by removing them from the data structure in which they resided, then queueing callbacks for their deferred reclamation. Again, since reader completion is inferred using quiescent-state observations, the read path remains free of synchronization instructions.

The performance of the synchronization strategies discussed in this section depends critically on the costs of synchronization operations. The next section therefore discusses the overhead of some of these synchronization operations on modern CPU architectures, and the effect of this overhead on selected synchronization strategies.

3 Synchronization on Modern CPUs

This section discusses the memory consistency semantics and atomic instruction overhead of modern CPU architectures and evaluates their impact on well-known synchronization strategies. We show that most synchronization strategies require memory barriers and atomic instructions that can be over a thousand times more expensive than normal instructions. A simple benchmark then shows that most well-known synchronization strategies have extremely poor performance and scalability on today's CPUs.

3.1 Memory Consistency Semantics

When designing a synchronization strategy, it is tempting to assume an execution environment with sequential consistency. Unfortunately, few modern CPUs implement sequential consistency. Instead, each CPU architecture defines its own weaker form of consistency, making it necessary to use special memory barrier, or fence, instructions to impose specific ordering constraints on memory operations. A memory barrier instruction forces all memory operations preceding it to be committed before any following it. Hence, such instructions disrupt the CPU's pipeline.

Table 1 summarizes the memory ordering characteristics of a range of modern CPU architectures. It shows that there are significant differences across architectures, and that most allow extensive reordering of instructions. The CPU names in parentheses correspond to less-favored modes of operation, for example, some x86 CPUs may be configured to reorder stores, but since most x86 software does not expect such reordering, these CPUs are rarely so configured. A "Y" in a given cell indicates that the answer to the column's question is "yes" for the row's CPU. For example, x86 CPUs permit loads to be reordered after subsequent loads and stores, and

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?
Alpha	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y			
IA64	Y	Y	Y	Y	Y	Y	
(PA-RISC)	Y	Y	Y	Y			
PA-RISC CPUs							
POWER	Y	Y	Y	Y	Y	Y	
(SPARC RMO)	Y	Y	Y	Y	Y	Y	
(SPARC PSO)			Y	Y		Y	
SPARC TSO				Y			
x86	Y	Y		Y			
(x86 OOSTore)	Y	Y	Y	Y			
zSeries				Y			

Table 1: Memory-Consistency Models

stores to be reordered after subsequent loads. However, x86 CPUs never reorder stores, never reorder atomic instructions, and never reorder dependent loads.

These weak consistency models impact synchronization strategies in two ways: complexity and performance. They increase complexity because the correctness of synchronization algorithms, particularly those used in NBS strategies, depends critically on the correct placement of memory barrier instructions. This task is difficult and error prone. Their impact on performance depends on the cost of memory barriers and the number required. The cost of memory barriers and other atomic instructions is addressed in the next section.

3.2 Instruction-Level Overhead

Table 2 quantifies the instruction-level overhead of a range of synchronization instructions on two widely-used modern CPU architectures, the Intel Xeon and the IBM POWER4. The performance figures for each instruction are normalized to the cost of a regular instruction that hits in the top-level cache, on each architecture.

Measuring overheads of single instructions on modern super-scalar microprocessors requires extreme care.¹ The approach used to generate the results in Table 2 was to measure a long series of instructions, but to execute them in a loop. For the first six rows of the

¹For example, sampling a high-precision time source before and after the instruction will give wildly inaccurate results due to instruction reordering by the CPU, in fact, negative values may be produced by such methods. Use of serializing instructions that disable such reordering have overheads exceeding, by orders of magnitude, that of the instruction being measured. One can repeat the instruction to be measured so that the error sources are amortized down to acceptable levels, but this approach introduces cache-miss, TLB-miss, and page-fault overheads which again exceed that of the instruction by orders of magnitude.

table, the loop overhead was removed by subtracting the overhead of a loop containing a single local non-atomic increment. Compiler optimizations were disabled to ensure that the code controlling the loop was the same in both cases.

Operation	Cost (Instructions)	
	1.45GHz POWER	3.06GHz Xeon
Instruction	1.0	1.0
Atomic Increment	183.1	402.3
SMP Write Memory Barrier	328.6	0.0
Read Memory Barrier	328.9	402.3
Write Memory Barrier	400.9	0.0
Local Lock Round Trip	1057.5	1138.8
CAS Cache Transfer and In- validate	* 247.1	847.1
CAS Blind Cache Transfer	* 257.1	993.9

* Varies with system topology, small-system value shown

Table 2: Synchronization Instruction Overhead

The measurements shown in the last two rows of the table required a pair of CPUs alternately writing to a cache line in a loop. In this case, the loop was coded so that the instructions controlling the loop executed concurrently with the movement of the cache line between the CPUs. In all cases, the cache-line-movement latency exceeds the overhead of the instructions controlling the loop by orders of magnitude, so this approach yields accurate results.

The first row measures the overhead of a no-operation instruction, providing the best-case instruction execution overhead. The overhead figures presented in all of the other rows are normalized with respect to this value. The second row measures an atomic increment instruction. This is simulated by an LL/SC sequence on POWER, which has no atomic increment. Atomic increment is used to implement concurrent counters and, in the Linux kernel, locking primitives. The third row measures an SMP write memory barrier. It is implemented as an `eieio` instruction on POWER, and compiles to nothing (not even a no-op) on x86.² SMP write memory barriers are used in RCU algorithms to ensure that data structures are perceived to have been initialized before they are linked into globally visible data structures. The fourth row measures a read memory barrier, implemented as an `lwsync` instruction on POWER and an atomic increment on x86. The fifth row measures a write memory barrier, which is a `sync` instruction on POWER and again compiles to nothing on x86. Both the read and write barriers are used in the implementation of locking primitives, and when running NBS algorithms on machines not featuring sequential consistency. A write memory barrier is distinguished from an SMP write memory barrier in that the former must order writes to memory-mapped I/O registers as well as to normal memory. This additional constraint means that the stronger `sync` POWER memory barrier must be used in place of the weaker `eieio` instruction that is used for the SMP write memory barrier, and that this code must be generated even in a kernel built specifically for uniprocessors. The sixth row measures

²However, the SMP write barrier disables any compiler optimizations that would otherwise reorder code across the barrier. That said, since we disabled optimization, this effect was not visible in our testing.

the cost of a local lock using a pair of LL/SC sequences on POWER, along with an `isync` barrier for acquisition and an `eieio` for release. On x86, the local lock uses CAS instructions. Since these instructions act as memory barriers on x86, no additional memory barriers are required.

The final two rows measure the cost of shuttling a cache line between a pair of CPUs. The first of these two rows reads the value, then uses that value in a subsequent CAS, while the last row blindly uses constant values for the CAS. On some systems there is a significant difference between these, due to interactions with the cache-coherence protocol [32].

The blind CAS (last row) is sometimes used for simple spinlocks, where the lock variable should be atomically changed to a constant “held” value (typically 1), but only if this variable previously contained a constant “not held” value (typically 0). The non-blind CAS (second-to-last row) is used for almost all NBS operations, where a pointer is updated, but only if it has not changed from its previous value.

The results show that synchronization instructions are very expensive on modern CPUs. Most synchronization instructions cost between two and three orders of magnitude more than normal instructions. The factors that account for these costs include instruction complexity, pipeline stalls, memory latency, and CPU-to-CPU communication latency. With pipelines becoming deeper and memory hierarchies taller, these costs have become significantly larger over recent years.

One way to understand the significance of these costs is in terms of critical section efficiency, or the number of normal instructions required within a critical section to amortize the cost of entering and leaving the critical section. If critical section efficiency is low, then performance will be poor even in the absence of lock contention. Similarly, when the cost of entering and leaving a critical section exceeds the cost of executing the code within the critical section, strategies such as reader-writer locking will not scale even in extreme read-mostly scenarios, because lock acquisition costs will prevent more than one reader from executing concurrently.

3.3 Impact on Synchronization Strategies

This section explores the impact of synchronization instruction overhead on the performance and scalability of various well-known synchronization strategies. Our ultimate interest is the scalability of synchronization strategies for operating system kernels. Therefore, we have constructed a benchmark typical of that environment. This benchmark consists of a mixed workload of hash-table searches and updates, with the hash table stored as a dense array of pointers, each of which references the hash-chain list for the corresponding bucket. The fraction of updates in the workload can be varied from zero (read-only) to one (update-only), and the size of the hash-table can be varied to explore the impact of caching effects on the results.

Figures 1 and 2 show the performance of various synchronization strategies, evaluated using the hash-table benchmark, on an 8-CPU 1.45 GHz POWER machine. The overhead of synchronization instructions on this CPU architecture is fairly typical of today’s CPUs (see Section 3.2). We evaluated five of the synchronization strategies described in Section 2: “globalrw” is global reader-writer locking; “brlock” is per-CPU reader-writer locking; “spinbkt” is per-bucket locking; “SMR” (safe memory reclamation) is non-blocking synchronization with hazard pointers; and “RCU” is read-copy-

update. “Ideal” represents the hypothetical optimal performance in which hash-table accesses are performed synchronization free.

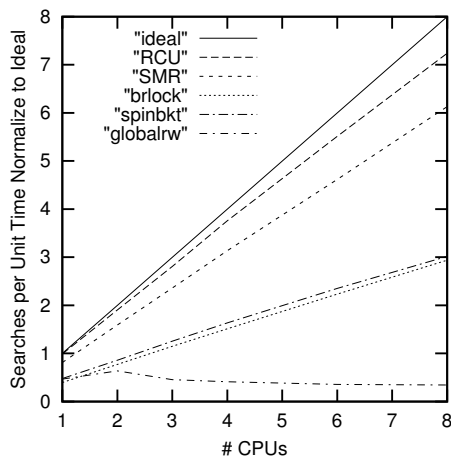


Figure 1: Scalability of Synchronization Strategies

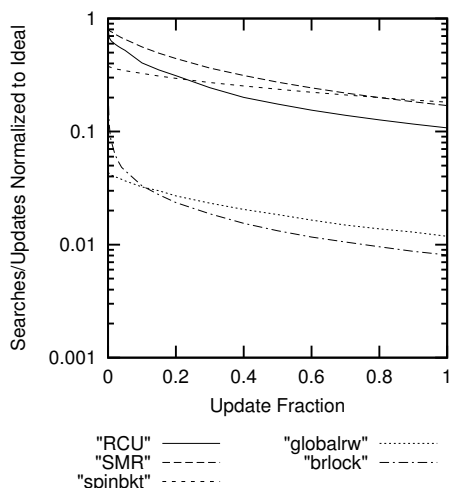


Figure 2: Performance of Synchronization Strategies at Different Update Fractions (8 CPUs)

Figure 1 shows the scalability of the strategies using a large hash-table and an update fraction of zero. The hash-table size was chosen to significantly exceed the cache size of the machine. Hence searches incur memory latency. We obtained similar, although more pronounced, results with hash-tables sized to fit entirely within first-level cache. The update fraction of zero was chosen to model extreme read-mostly scenarios, which are common in operating system kernels. The relative performance of the strategies across the full range of update fractions is shown in Figure 2.

Figure 1 shows, as expected, that globalrw scales negatively. This is because the overhead of lock acquisition exceeds the work in the critical section, and hence prevents readers from ever executing concurrently. Although brlock fares better, it still only achieves around a third of the ideal performance. Again, this is due to the overhead of lock acquisition in the read path. A per-bucket spinlock fares slightly better than does brlock, due to the poorer spatial cache lo-

cality of brlock.³ The strategies based on concurrent reading and deferred reclamation (RCU and SMR) fare much better, with RCU achieving close to the ideal performance. This result is not particularly surprising given that both strategies optimize the read path and these results were gathered on a read-only workload.

Figure 2 shows the effect of varying the update fraction in the workload. This experiment was run on 8 CPUs using a large hash table, running at most one thread per CPU. The figure shows that the synchronization strategies based on concurrent reading and deferred reclamation (RCU and SMR) performance is competitive with the others even when the workload is update-dominated. In light of these results, we focus our attention in the remainder of this paper on these strategies.

4 Evaluation of Deferred-Reclamation-Based Synchronization Strategies

The RCU and SMR strategies evaluated in the previous section were based on implementations already described in the literature. The RCU strategy uses lock-based update and quiescent-state-based memory reclamation. This approach is used extensively in the Linux kernel [32]. The SMR strategy uses NBS update and hazard-pointer-based memory reclamation. Our implementation of this strategy is consistent with Michael’s original design [39]. As noted earlier, however, update strategy and reclamation strategy are orthogonal choices. Hence, in this section we introduce two additional synchronization strategies. One uses an NBS update strategy and quiescent-state-based reclamation. We refer to this strategy as RCU-NBS, in contrast to the lock-based RCU strategy which we refer to as RCU-L. RCU-NBS is similar to the use of RCU for lock-free hash tables in the K42 kernel [32]. The other new strategy uses hazard-pointer-based reclamation with a lock-based update strategy. We refer to this strategy as SMR-L, in contrast to the NBS-based SMR strategy which we refer to as SMR-NBS. These four strategies allow us to explore the performance implications of update strategy and reclamation strategy independently.

The following subsections compare the performance, scalability and memory usage of these strategies under various workloads. All experiments use the hash-table benchmark described in Section 3.3 and were run on an 8-CPU 1.45GHz POWER4 system. Since an obvious cost of deferred reclamation-based strategies is memory overhead, we configured all of the experiments in Sections 4.1 and 4.2 to incur the same worst-case memory consumption. This was achieved in the SMR-based approaches by limiting the per-thread reclamation list size, and in the RCU-based approaches by limiting the number of operations per quiescent state for each CPU. The impact of different memory constraints and the memory usage characteristics of each strategy in extreme cases are then discussed in Sections 4.3 and 4.4, respectively. Finally, Section 4.5 presents a qualitative comparison of SMR and RCU.

4.1 Impact of Memory Latency

Given the high overhead of synchronization instructions on modern CPUs, and the fact that this overhead comes in part from the large disparity between cache and memory latency, it is natural to expect the performance characteristics of synchronization strategies to dif-

³Note that the performance of brlock increases sharply compared to that of per-bucket spinlock in cases where the hash tables are sized to fit entirely in first-level cache.

fer between workloads that always hit in first-level cache and those that go to memory. For example, one would expect synchronization instruction overhead to have more impact on workloads that always hit in cache. This section explores the impact of memory latency on the performance and scalability of the four strategies, by using two different workloads: one with a small hash-table, sized to fit entirely in first-level cache, and another with a hash-table that is much larger than the cache. The results for the small and large hash-tables are presented in Figures 3 and 4, respectively. For both experiments, an update fraction of zero was used to model extreme read-mostly scenarios. The impact of update fraction on the relative performance of the strategies is discussed in the next section.

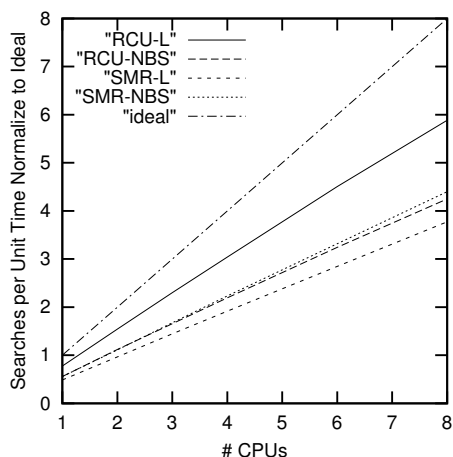


Figure 3: RCU and SMR Scalability When Working Set Fits in Cache

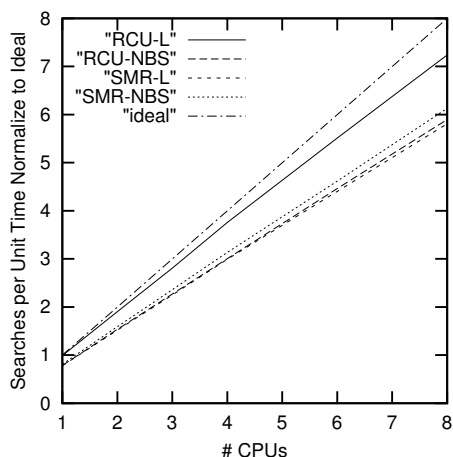


Figure 4: RCU and SMR Scalability When Working Set Does Not Fit in Cache

The two graphs have been normalized to better show scalability; however, the absolute ideal performance for searching a small hash table is a factor of 2.3 times better than that for the large hash table. This difference in performance is due to the increased cache-miss rate for the large hash table over that of the small hash table. Note that this increased cache-miss rate affects the ideal performance as

well as that of each synchronization mechanism, so that differences between these mechanisms are muted in the large-hash-table case.

In the small-table case, RCU-L has the best performance, followed by RCU-NBS, SMR-NBS, and SMR-L, in that order. In the large-table case, RCU-L again has the best performance, followed by SMR-NBS, with RCU-NBS and SMR-L having very nearly the same performance. In both cases, SMR-L and SMR-NBS are slowed by the memory barriers needed to manage read-side hazard-pointer manipulations on this weak-memory-consistency machine, while RCU-NBS and SMR-NBS are slowed by the read-side checks needed to “help” updates. As with many NBS algorithms, such read-side helping is required to handle races between concurrent updates that can leave elements partially removed from the list. The improved relative performance of SMR-NBS in the large-table case is due to the CPU’s ability to overlap the hazard-pointer-induced memory-barrier overhead with cache-miss latencies. However, the performance differences between the RCU-NBS, SMR-L, and SMR-NBS approaches is small enough to be sensitive to minor variations in both the system hardware and the compiler.

4.2 Impact of Update Fraction

Since the four strategies differ in the way they distribute overhead between the read and update path, it is interesting to evaluate their relative performance at different update fractions and to identify the break-even points. Figure 5 presents the performance of the four strategies over the full range of update fractions from zero (read-only) to one (update-only). The experiment was run on an 8 CPU machine using the large hash table. Figure 6 presents the same experiment run on a 2 CPU machine to determine the impact of varying the number of CPUs.

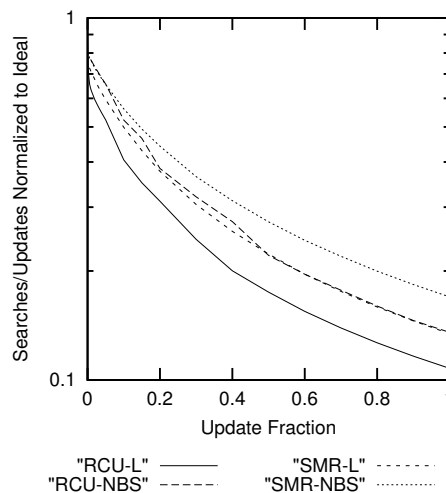


Figure 5: Performance of Deferred-Reclamation-Based Strategies at Different Update Fractions on 8 CPUs

Figure 5 shows that SMR-NBS performs better than the other strategies at all update fractions above 0.1, and that RCU-NBS is the best-performing strategy at all update fractions below this (at least those that are visible on this graph). Note however that in the read-only cases shown in Figures 3 and 4 RCU-L was the best performer. This result begs the question of what happens at the read-mostly end of the spectrum, and where the break-even point is among the various strategies. In order to answer this question, Figure 7 presents

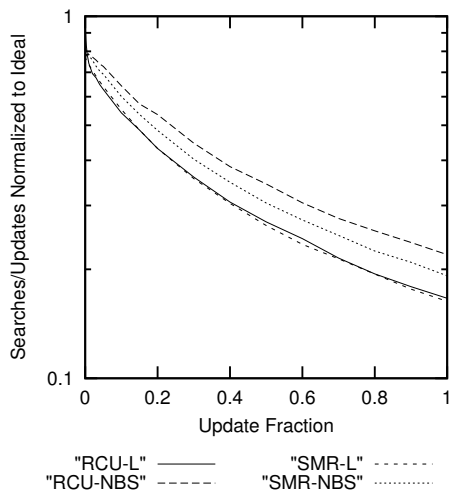


Figure 6: Performance of Deferred-Reclamation-Based Strategies at Different Update Fractions on 2 CPUs

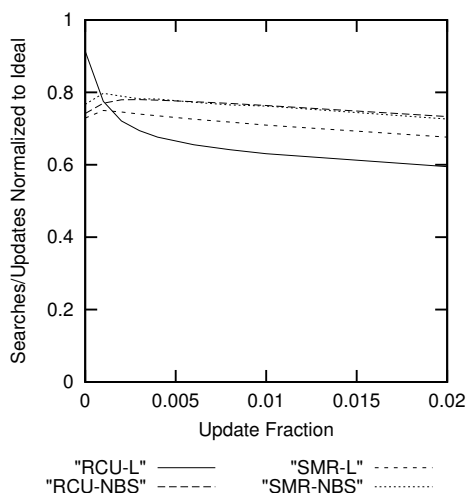


Figure 7: Relative Performance of Deferred-Reclamation-Based Strategies in Read-Mostly Scenarios

a zoomed in view of the relative performance of the four strategies between update fractions of 0 and 0.02. Note that while RCU-L has the worst performance at all update fractions above 0.001, it gains a significant advantage as the update fraction decreases to zero.⁴ This is partially due to the fact that RCU-L's memory reclamation has nothing to do given a read-only workload, and partially do to the fact that we have chosen a simple memory-reclamation algorithm that was tuned on a two-CPU system. We do not fully understand the slight increase in performance of the RCU-NBS, SMR-NBS, and SMR-L algorithms with increasing values of update fraction in the extreme read-mostly regime. This effect is most pronounced for RCU-NBS, and quite small for SMR-L. We believe that this effect is due to the ability of the super-scalar CPUs to overlap the update-side cache misses with read-side operations. This overlapping would be expected to be most pronounced for RCU-NBS, since there are no read-side memory barriers and only one update-side memory barrier.

⁴However, there are numerous data structures in production operating systems with update fractions well below 10^{-6} , for example, data structures that track software and hardware configuration, including routing tables.

ers. RCU-L and SMR-L would be expected to show the least effect, since the update-side locking and memory barriers incur the greatest increases in overhead with increasing update fraction, and since the pair of update-side memory barriers greatly limit the attainable overlap. Needless to say, this sort of effect is quite machine- and compiler-dependent.

For update-intensive workloads, the relative performance depends on the number of CPUs, with RCU enjoying a 2%-to-13% advantage at two CPUs and SMR enjoying a 25% advantage at eight CPUs. It is not yet clear whether RCU's scaling can be improved to match that of SMR in update-intensive workloads, or whether SMR has an inherent advantage in this regime.

4.3 Impact of Memory Constraints

Figure 8 shows the impact of varying the amount of extra memory provided to RCU-L and SMR-NBS. Note that the y-axis is linear rather than logscale. The pairs of traces, from bottom to top, correspond to 1, 2, 3, 4, 6, and 8 CPU configurations, respectively. Each run was conducted at an update fraction of 0.2, so that there was on average two updates per ten operations. Of each pair, the shorter curve corresponds to SMR-NBS and the longer to RCU-L. Both algorithms are quite insensitive to memory constraint over a range of from one to three orders of magnitude. Even outside of this "flat range", the sensitivity of the algorithms is small compared to the effect of cache misses or memory barriers.

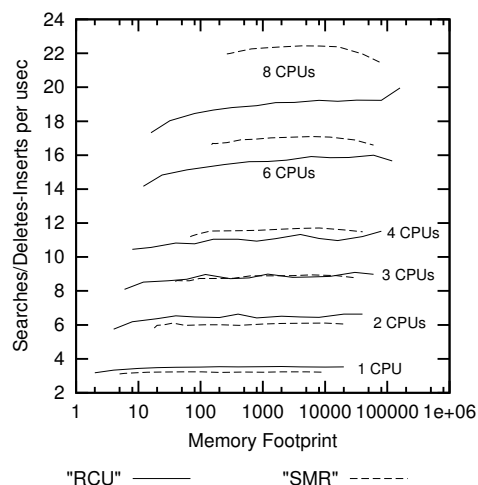


Figure 8: Impact of Memory Constraints on the Performance of Deferred-Reclamation-Based Strategies

4.4 Memory Usage

The fact that hazard-pointer-based designs must explicitly track readers' references means that these readers must do additional stores in order to update hazard pointers. Additionally, since almost all modern systems feature weak memory consistency, these readers must also make use of expensive memory-barrier instructions. However, one benefit that hazard-pointer-based designs derive from explicitly tracking readers' references is that data items can be actively reclaimed, for example, under low-memory conditions. In contrast, QSBR-based designs are less able to actively reclaim memory, since the only way to do so is to cause each active thread to execute a quiescent state, which may not be possible.

4.5 Qualitative Evaluation

This section discusses the less easily quantifiable characteristics of the update strategy and reclamation strategy choices.

4.5.1 Update Strategy

A number of well-known problems are commonly associated with locking, including deadlock, lock contention, convoys, blocking due to page faults, blocking due to preemption, lockup due to thread failure, and the high overhead of synchronization instructions. While these problems have not been entirely solved, many have partial solutions that are considered acceptable, from an engineering standpoint, in production systems. For example, good engineering practices and tools have significantly reduced the likelihood of deadlock [4, 12, 23, 31]. The synchronization strategies described earlier, in Section 2, reduce lock contention. Convoys and blocking due to preemption can be avoided through tighter integration of scheduling and locking [26], and blocking due to page faults can be addressed by over-provisioning memory, which is particularly attractive given the ever-increasing size and declining cost of memory. Thread failure is often addressed by the simple but effective expedient of terminating the application. The high overhead of synchronization instructions remains a problem, but only for update-intensive scenarios, since the synchronization strategies discussed in this paper remove such instructions from the read path.

NBS can reduce the cost of synchronization instructions in update-intensive scenarios. In fact, the results presented earlier showed that NBS performs up to 30% better than locking for update-intensive workloads. However, these results should be applied with caution, since they were gathered using a benchmark for which an efficient NBS implementation exists. NBS also solves the problems of deadlock, contention, blocking due to page faults, blocking due to preemption, and lockup due to thread failures, but it does so at the expense of increased code complexity, more difficult integration with legacy code, and increased memory contention. Recent work on obstruction-free synchronization sidesteps livelock and memory contention issues in the same manner that locking practitioners have done, namely by requiring that the engineering design maintain low contention. There has also been recent work aimed at reducing NBS complexity [11, 19], but it remains to be seen how effective these approaches are. Integration with legacy code is still an open issue. NBS algorithms have been successfully integrated with legacy code, but there has been little work on incrementally migrating a large body of code from locking to NBS. In contrast, lock-based RCU strategies have been incrementally integrated into at least four major operating systems, three of which have seen extensive production use.

4.5.2 Reclamation Strategy

Reclamation schemes based on hazard pointers explicitly flag the specific data items that are being referenced by readers. Therefore, they can actively reclaim all memory that is not so referenced. In contrast, reclamation strategies based on quiescent states do not maintain explicit lists of referenced data items and hence must make more conservative assumptions about which data items might be referenced. These conservative assumptions can delay the reclamation of large amounts of memory. Therefore, hazard-pointer-based approaches have smaller memory footprints than quiescent-state-based approaches.

The reclamation strategies differ not only in their ability to eagerly and accurately reclaim memory, but also in their internal use of memory. For example, in hazard-pointer-based approaches, memory must be allocated for the hazard pointers themselves. The hash-table benchmark used in the experiments presented earlier requires only two hazard pointers per thread, and these can be allocated statically. However, static allocation of hazard pointers is not practical for more complex applications, which may feature nested data structures and recursive searches. The API proposed by Herlihy [19] permits dynamic allocation of hazard pointers, but such dynamic allocation requires a mechanism for freeing, which imposes an additional burden on programmers in environments that lack a garbage collector.⁵ In addition, dynamic allocation raises the question of how to handle allocation failure. One could simply block until memory becomes available, which limits performance and robustness, or one could return a failure indication. Failure indications place yet another burden on the programmer, who must either preallocate all the hazard pointers that might be needed for a given operation, or must carefully code unwind paths that recover from allocation failure. In contrast, since quiescent-state-based approaches do not explicitly track referenced memory blocks, they do not need to do any read-side memory allocation.

Another complication with hazard-pointer-based reclamation schemes is their handling of data structures with variable length aggregates of other data structures. These cases present a problem because they allow different references to different portions of the same data item. Simple comparisons of hazard pointers do not detect these conflicts and can result in premature reclamation. To solve these problems, hazard-pointer-based schemes need a way to map from a reference to a portion of a data item to a canonical reference for that data item. This requirement places constraints on the implementation of the environment's memory allocators, and can be expected to increase cost and complexity. Such reference mapping is not necessary in quiescent-state-based approaches that do not explicitly track referenced memory blocks.

The main price paid by quiescent-state-based schemes for these advantages, apart from the memory overhead, is the need for environmental support for maintaining quiescent state information. The degree to which these approaches can be successful depends on the availability of quiescent states in the environment, their type, and the frequency with which they occur. These factors are environment-specific.

In principle, hazard-pointer-based approaches should offer tighter control over latency in real-time systems, since the use of hazard pointers enables preemption of read-side critical sections without affecting reclamation of data items that are not specifically referenced by the preempted thread. In contrast, most quiescent-state-based approaches disable preemption across read-side critical sections, with the notable exception of the K42 operating system. K42 permits preemption by excluding involuntary context switch from its set of quiescent states. Since the choice of quiescent states affects both overhead and memory footprint, there are a range of potential design points for quiescent-state-based approaches that trade real-time performance, memory overhead, and scalability in different ways. Again, the available design choices are environment-specific.

⁵Note that environments *with* garbage collectors already have built-in reclamation, and therefore have no need for either RCU or SMR.

5 Impact of Future Trends

The synchronization developments discussed in this paper occurred in response to developments in CPU architecture. The evolution of CPU architecture is ongoing, and unpredictable. In this section we explore six possible future trends in CPU architecture and consider the impact they would have on the synchronization techniques discussed earlier.

Trend 1: Single-threaded uniprocessors

If the combination of Moore's-Law increases in CPU clock rate and continued progress in horizontally scaled computing render shared-memory multiprocessor systems irrelevant, synchronization instructions on the resulting uniprocessor systems will not suffer the cache-thrashing, contention, and memory barrier overheads of today's systems. In this scenario, synchronization techniques that use deferred reclamation to avoid these overheads in the read-path will become less relevant, and their use may only continue for niche applications such as interacting with non-maskable interrupts. However, recent trends indicate that this scenario is quite unlikely.

Trend 2: Multi-threaded uniprocessors

Current hardware multithreading trends may lead to a predominance of uniprocessor systems that are aggressively multithreaded with hardware-supported threads sharing all levels of the cache hierarchy. In this scenario, CPU-to-CPU communication latency is eliminated and the performance penalty for synchronization instructions is reduced. However, multithreaded CPUs would still incur overhead due to contention and pipeline stalls caused by memory barriers. Furthermore, if all hardware threads share all levels of cache, the cache interference among threads may degrade performance. On the other hand, partitioning some levels of cache on a per-hardware-thread basis re-introduces memory latency for cache-lines that are passed from one thread to another. In both cases, synchronization approaches that use deferred reclamation to avoid both contention and pipeline stalls are likely to be useful. It remains to be seen what performance impact shared or partitioned caches will have on the grace-period management algorithms used by such approaches.

Trend 3: Single-chip multiprocessors

The performance advantages of single-chip multiprocessors (more recently called dual- or multi-core dies) over super-scalar single CPUs with the same chip area were demonstrated almost a decade ago [40], and have recently appeared in commercial CPU offerings and announcements. These performance advantages are due to the limited amount of instruction-level parallelism in typical software.

However, use of multiple cores is not a panacea, since great care must be taken to provide sufficient cache as well as memory and I/O bandwidth, otherwise, the multiple CPUs will simply stall waiting for data to flow on- and off-chip. Nevertheless, latencies among CPUs on the same chip are quite good compared to inter-chip latencies. It remains to be seen what performance characteristics are offered by future single-chip multiprocessors, and what the consequent effect on the performance of memory-reclamation algorithms will be.

In this scenario, synchronization instructions remain expensive due to pipeline stalls. Although the memory-latency cost of CPU-to-CPU communication will be relatively small among CPUs on the same chip, it will still be quite expensive compared to normal instruction execution overhead.

Trend 4: Growing memory latency

If memory latency continues to grow relative to instruction execution overhead, this will increase the benefit of avoiding synchronization instructions, but will also increase the cost of managing deferred-reclamation-based synchronization mechanisms. On the other hand, the deferred-reclamation benefit of avoiding synchronization instructions will exceed the increased costs of memory latency, provided that the deferred-reclamation mechanisms amortize their cost over a sufficiently large number of accesses.

However, it is not clear that this decades-long trend will continue. To see this, consider that 1GHz CPUs first appeared about five years ago. If CPU clock frequencies had continued the earlier trend of doubling every 18 months, they would now be approaching 10GHz. Instead, they have yet to exceed 5GHz in absence of heroic measures, such as liquid-nitrogen cooling. Furthermore, a number of technology trends, such as increased leakage current and decreased tolerance for excessive power dissipation, seem likely to limit future clock-frequency increases. This slowing rate of clock-frequency increase seems likely to end the historic trend of ever-increasing memory latencies, leading to Trend 5.

Trend 5: More of the same

Finally, if increases in interconnect performance match Moore's-Law-driven increases in core CPU performance, memory latencies may remain roughly where they are today. In this scenario, overhead due to pipeline stalls, memory latency, and contention remains significant, hence synchronization approaches that use deferred reclamation will retain the high level of applicability they enjoy today. This scenario seems quite probable, given that CPU-clock frequencies seem to have levelled off over the past few years.

Trend 6: Transactional memory

An alternative way of viewing modern synchronization approaches is in terms of their use of speculation. One could argue that NBS approaches speculate in time. That is, they perform updates speculatively, and if unsuccessful they repeat the speculative update. Increased concurrency increases the time taken to complete the update. RCU- and hazard-pointer-based approaches, on the other hand, speculate in space. That is, their updates create new versions, and increased concurrency increases the number of versions, and hence the space they occupy. Combining NBS with either RCU or hazard pointers therefore results in speculation both in space and time, while combining locking with either RCU or hazard pointers results in speculation only in space. In the future, it would be interesting to explore the relative merits of these forms of speculation given various architectural trends, costs and constraints. For example, one might expect speculation in space to consume more memory, but less power, than speculation in time.

Some of the issues mentioned above are addressed by ongoing research in hardware support for speculative execution and transactional memory [27, 28, 42, 43]. This research leverages the speculative-execution facilities present in many CPUs to execute transactions that either commit or abort atomically. Committing is handled like successful speculation, with results being committed to memory or registers, whereas aborting is treated like failed speculative execution, with the results being discarded. Essentially, such hardware transactions act like an atomic N-way compare-and-swap instruction, with the complexity of the transaction limited by the amount of speculative state that can be stored. Speculative state is stored in the CPU's cache, but specially marked so that it will not be

committed to memory unless the transaction commits. Since CPU caches can be large, this approach suggests that extremely large transactions could be supported. Unfortunately, the associativity limits of hardware caches severely limit the maximum guaranteed-to-commit transaction size, and the tension between access latency and associativity discourages large increases in associativity.

Trend summary

At this writing, it appears that the future lies with trends 2, 3, and 5, in other words, with multi-core, multi-threaded CPUs that have roughly the same relative cost of synchronization that is seen today. These trends will become especially prominent if CPU-clock-frequency growth remains low, since in that case, the only way to increase performance is to increase parallelism, either through addition of general-purpose processors or through addition of hardware accelerators or vector units. As is the case today, smaller systems will enjoy lower costs of synchronization, all else being equal.

6 Conclusions and Future Work

This paper has shown that synchronization instructions on modern CPUs are very expensive, in some cases costing over a thousand times more than normal instructions. To make matters worse, weak memory consistency models require additional instructions, in the form of memory barriers, to be added to synchronization algorithms on most CPUs. These too can be expensive, costing several hundred times more than normal instructions. This instruction-level overhead has dramatically decreased critical section efficiency, causing some formerly scalable synchronization strategies to perform very poorly on today's CPUs.

Synchronization solutions that use a copy-based update approach, with deferred reclamation and synchronization-free concurrent reading, scale much better than other strategies on modern CPUs. They also offer a range of design decisions relating to the specific update management and reclamation strategy to use. The choice of locking vs. non-blocking synchronization for managing updates is orthogonal to the choice of reclamation strategy, and we examined the performance characteristics of four different combinations of update and reclamation strategy. The results showed that in extreme read-mostly scenarios, of which there are many in production kernels, a combination of lock-based update strategy and quiescent-state-based reclamation strategy performs the best. Reclamation strategies based on hazard pointers suffer degraded performance in read-mostly conditions due to the need for expensive memory barriers in the read-path. These performance differences become more pronounced as the working set size decreases and the cache-hit-rate increases. As the percentage of updates in the workload increases, NBS-based update strategies became more efficient than lock-based strategies, due in part to decreased lock contention and to the existence of efficient NBS algorithms for the simple data structures studied in this paper.

At smaller numbers of CPUs, reclamation strategies based on quiescent states performed well, but as the number of CPUs increases, so does the cost of managing quiescent states, pointing to the need for scalable algorithms for quiescent state management. None of the strategies exhibited any performance sensitivity to variations in available memory. However, for extremely tight memory constraints, such as in embedded systems, reclamation strategies based on hazard pointers offer tighter control over memory consumption than those based on quiescent states. Finally, we argued that the rel-

ative performance of these synchronization strategies is dependent on future architectural trends.

6.1 Future Work

All four algorithms studied in this paper expose readers to the possibility of observing stale data items. A data item is stale if it has been logically deleted but not yet reclaimed. A surprisingly large number of operating system algorithms tolerate stale data, as exemplified by the almost 300 uses of the RCU API in the Linux kernel. However, there are many algorithms that cannot tolerate stale data. One strategy for dealing with such algorithms is to transform them into a form that can tolerate stale data before applying this class of synchronization solutions. Several examples of such transformational design patterns are presented in [32]. However, a number of open questions remain. For example, (1) are there algorithms that cannot be so transformed? (2) which algorithms, when so transformed, remain efficient? (3) is there a "best" set of transformations, or are different transformations appropriate for different situations? (4) Do the currently identified transformations form a complete set, or are there others?

Deferred-reclamation-based synchronization algorithms are sufficiently different in nature from traditional locking and NBS algorithms that new formalisms will be required to validate and analyze them. Ideally, these formalisms will form the basis for a set of software tools that aid in the analysis and verification of deferred-reclamation-based algorithms. Similarly, tools are needed to aid in the adaptation of legacy code to these scalable synchronization approaches.

Another challenge for deferred-reclamation-based approaches, particularly those based on quiescent-state-based reclamation, is the need to deal with real-time workloads. In embedded systems this challenge often goes hand in hand with the need to run under tight memory constraints. Together, these two requirements point to the need for efficient active reclamation strategies.

Acknowledgements

(Acknowledgements elided.)

Source code for all experiments will be made available.

References

- [1] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (January 1990), 6–16.
- [2] ANDREWS, G. R. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*: 23, 1 (1991), 49–90.
- [3] APPAVOO, J., HUI, K., STUMM, M., WISNIEWSKI, R., DA SILVA, D., KRIEGER, O., AND SOULES, C. An infrastructure for multiprocessor run-time adaptation. In *Proceedings of WOSS 2002 (ACM SIGSOFT Workshop on Self-Healing Systems)* (August 2002), Association for Computing Machinery, pp. 3–8.
- [4] BECK, B., AND KASTEN, B. VLSI assist in building a multiprocessor UNIX system. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 255–275.

- [5] BERSHAD, B. N. Practical considerations for non-blocking concurrent objects. In *International Conference on Distributed Computing Systems* (1993), pp. 264–273.
- [6] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997), ACM SIGOPS, pp. 143–156.
- [7] CHAVES, E., LEBLANC, T. J., MARSH, B. D., AND SCOTT, M. L. Kernel-kernel communication in a shared-memory multiprocessor. In *Proceedings of the Second Symposium on Distributed and Multiprocessor Systems* (Atlanta, GA, March 1991), USENIX Association, pp. 105–116.
- [8] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)* (November 1994), USENIX Association, pp. 179–193.
- [9] COURTOIS, P. J., HEYMANS, F., AND PARNAS, D. L. Concurrent control with “readers” and “writers”. *Communications of the ACM* 14, 10 (October 1971), 667–668.
- [10] ENGLER, D. R., KAASHOEK, M. F., AND JR., J. O. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), Association for Computing Machinery, pp. 251–266.
- [11] FRASER, K. A. *Practical Lock-Freedom*. PhD thesis, King’s College, University of Cambridge, 2003.
- [12] GAMSA, B., KRIEGER, O., APPAVOO, J., AND STUMM, M. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation* (New Orleans, LA, February 1999), pp. 87–100.
- [13] GARG, A. Parallel STREAMS: a multi-processor implementation. In *USENIX Conference Proceedings* (Berkeley CA, February 1990), USENIX Association, pp. 163–176.
- [14] GOVIL, K., TEODOSIU, D., HUANG, Y., AND ROSENBLUM, M. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Operating Systems Principles* (Charleston, SC, December 1999), ACM SIGOPS, pp. 154–169.
- [15] GREENWALD, M., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), USENIX Association, pp. 123–136.
- [16] HENNESSY, J. P., OSISEK, D. L., AND SEIGH II, J. W. Passive serialization in a multitasking environment. Tech. Rep. US Patent 4,809,168 (lapsed), US Patent and Trademark Office, Washington, DC, February 1989.
- [17] HERLIHY, M. Wait-free synchronization. *ACM TOPLAS* 11, 1 (January 1991), 124–149.
- [18] HERLIHY, M. Implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* 15, 5 (November 1993), 745–770.
- [19] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing* (October 2002), pp. 339–353.
- [20] HERLIHY, M., LUCHANGCO, V., AND MOIR, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)* (Providence, RI, May 2003), The Institute of Electrical and Electronics Engineers, Inc., pp. 73–82.
- [21] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)* (July 2003), pp. 92–101.
- [22] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. *The 20th Annual International Symposium on Computer Architecture* (May 1993), 289–300.
- [23] HOLZMANN, G. J. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [24] HSIEH, W. C., AND WEIHL, W. E. Scalable reader-writer locks for parallel systems. Tech. Rep. MIT/LCS/TR-521, MIT Laboratory for Computer Science, Cambridge, MA, 1991.
- [25] INMAN, J. Implementing loosely coupled functions on tightly coupled engines. In *USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, pp. 277–298.
- [26] KONTOTHANASSIS, L., WISNIEWSKI, R. W., AND SCOTT, M. L. Scheduler-conscious synchronization. *Communications of the ACM* 15, 1 (January 1997), 3–40.
- [27] MARTINEZ, J. F., AND TORRELLAS, J. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues, International Symposium on Computer Architecture* (Gothenburg, Sweden, June 2001).
- [28] MARTINEZ, J. F., AND TORRELLAS, J. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, October 2002), pp. 18–29.
- [29] MASSALIN, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY, 1992.
- [30] MASSALIN, H., AND PU, C. Threads and input/output in the synthesis kernel. *ACM Operating Systems Review, SIGOPS* 23, 5 (1989), 191–201.
- [31] MCKENNEY, P. E. Selecting locking primitives for parallel programs. *Communications of the ACM* 39, 10 (October 1996), 75–82.
- [32] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

- [33] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *Ottawa Linux Symposium* (July 2001).
- [34] MCKENNEY, P. E., SLINGWINE, J., AND KRUEGER, P. Experience with an efficient parallel kernel memory allocator. *Software – Practice and Experience* 31, 3 (March 2001), 235–257.
- [35] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (Las Vegas, NV, October 1998), pp. 509–518.
- [36] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third PPOPP* (Williamsburg, VA, April 1991), pp. 106–113.
- [37] MICHAEL, M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc of the Fifteenth ACM Symposium on Principles of Distributed Computing* (May 1996), pp. 267–275.
- [38] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architecture* (August 2002), pp. 73–82.
- [39] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (June 2004), 491–504.
- [40] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. The case for a single-chip multiprocessor. In *ASPLOS VII* (October 1996).
- [41] PU, C., MASSALIN, H., AND IOANNIDIS, J. The Synthesis kernel. *Computing Systems* 1, 1 (January 1988), 11–32.
- [42] RAJWAR, R., AND GOODMAN, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Austin, TX, December 2001), The Institute of Electrical and Electronics Engineers, Inc., pp. 294–305.
- [43] RAJWAR, R., AND GOODMAN, J. R. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (Austin, TX, October 2002), pp. 5–17.
- [44] ROSENBLUM, M., BUGNION, E., HERROD, S. A., WITCHEL, E., AND GUPTA, A. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles* (Copper Mountain Resort, CO, December 1995), ACM SIGOPS, pp. 285–298.
- [45] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada, August 1995), pp. 204–213.
- [46] SOULES, C. A. N., APPAVOO, J., HUI, K., DA SILVA, D., GANGER, G. R., KRIEGER, O., STUMM, M., WISNIEWSKI, R. W., AUSLANDER, M., OSTROWSKI, M., ROSENBERG, B., AND XENIDIS, J. System support for online reconfiguration. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003), USENIX Association, pp. 141–154.
- [47] TREIBER, R. K. Systems programming: Coping with parallelism. RJ 5118, April 1986.
- [48] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DAN-
NOWSKI, U. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium* (San Jose, CA, May 6-7 2004), pp. 43–56.
- [49] UNRAU, R., KRIEGER, O., GAMSA, B., AND STUMM, M. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing* 9, 1/2 (1995), 345–370.
- [50] UNRAU, R. C., KRIEGER, O., GAMSA, B., AND STUMM, M. Experiences with locking in a NUMA multiprocessor operating system. In *Proceedings of the First Symposium on Operating Systems Design and Implementation* (November 1994), USENIX Association, pp. 139–152.
- [51] VALOIS, J. D. Lock-free linked lists using compare-and-swap. In *Symposium on Principles of Distributed Computing* (1995), pp. 165–172.
- [52] WISNIEWSKI, R. W., KONTOTHANASSIS, L., AND SCOTT, M. L. Scalable spin locks for multiprogrammed systems. In *8th IEEE Int'l. Parallel Processing Symposium* (Cancun, Mexico, April 1994), The Institute of Electrical and Electronics Engineers, Inc.