The Synergy Between Non-blocking Synchronization and Operating System Structure

Michael Greenwald and David Cheriton [*]
Computer Science Department
Stanford University
Stanford, CA 94305-9040

Abstract

Non-blocking synchronization has significant advantages over blocking synchronization: however, it has not been used to a significant degree in practice. We designed and implemented a multiprocessor operating system kernel and run-time library for high-performance, reliability and modularity. We used non-blocking synchronization, not because it was an objective in itself, but because it became the approach of choice. It was an attractive approach because of the synergy between other structuring techniques we used to achieve our primary goals and the benefits of non-blocking synchronization.

This paper describes this synergy: the structuring techniques we used which facilitated non-blocking synchronization and our experience with this implementation.

1 Introduction

We chose to use non-blocking synchronization in the design and implementation of the Cache Kernel [7] operating system kernel and supporting libraries for several reasons. First, non-blocking synchronization allows synchronized code to be executed in an (asynchronous) signal handler without danger of deadlock. For instance, an asynchronous RPC handler (as described in [25]) can directly store a string into a synchronized data structure such as a hash table even though it may be interrupting another thread updating the same table. With locking, the signal handler could deadlock with this other thread.

Second, non-blocking synchronization minimizes interference between process scheduling and synchronization. For example, the highest priority process can access a synchronized data structure without being delayed or blocked by a lower priority process. In contrast, with blocking synchronization, a low priority process holding a lock can delay a higher priority process, effectively defeating the process scheduling. Blocking synchronization can also cause one process to be delayed by another lockholding process that has encountered a page fault or a cache miss. The delay here can be hundreds of thousands of cycles in the case of a page fault. This type of interference is particularly unacceptable in an OS like the Cache Kernel where real-time threads are supported and page faults (for non-real-time threads) are handled at the library level. Non-blocking synchronization also minimizes the formation of convoys which arise because several processes are queued up waiting while a single process holding a lock gets delayed.

Finally, non-blocking synchronization provides greater insulation from failures such as fail-stop process(or)s failing or aborting and leaving inconsistent data structures. Non-blocking techniques allow only a small window of inconsistency, namely during the atomic compare-and-swap sequence itself. In contrast, with lock-based synchronization the window of inconsistency spans the entire locked critical section. These larger critical sections and complex locking protocols also introduce the danger of deadlock or failure to release locks on certain code paths.

There is a strong synergy between non-blocking synchronization and the design and implementation of the Cache Kernel for performance, modularity and reliability. First, signals are the *only* kernel-supported form of notification, allowing a simple, efficient kernel implementation compared to more complex kernel message primitives, such as those used in V [6]. Class libraries implement higher-level communication like RPC in terms of signals and shared memory regions [25]. Non-blocking synchronization allows efficient library implementation without the overhead of disabling and enabling signals as part of access and without needing to carefully restrict the code executed by signal handlers.

Second, we simplified the kernel and allows specialization of these facilities using the C++ inheritance mechanism by implementating of most operating system mechanisms at the class library level, particularly

[*] {michaelg, cheriton}@cs.stanford.edu. This work was sponsored in part by ARPA under US Army contract DABT63-91-K-0001. Michael Greenwald was supported by a Rockwell Fellowship.

the object-oriented RPC system [25]. Non-blocking synchronization allows the class library level to be tolerant of user threads being terminated (fail-stopped) in the middle of performing some system library function such as (re)scheduling or handling a page fault.

Finally, the isolation of synchronization from scheduling and thread deletion provided by non-blocking synchronization and the modularity of separate class libraries and user-level implementation of services leads to a more modular and reliable system design than seems feasible by using conventional approaches.

This synergy between non-blocking synchronization and good system design and implementation carries forward in the more detailed aspects of the Cache Kernel implementation. In this paper, we describe aspects of this synergy in some detail and our experience to date.

The main techniques we use for modularity, performance and reliability are atomic DCAS(or `Double-Compare-and-Swap`), *type-stable memory management* (TSM), and *contention-minimizing data structures* (CMDS).

DCAS (discussed in detail in Section 5) is defined in Figure 1. That is, DCAS atomically updates locations `addr1` and `addr2` to values `new1` and `new2` respectively if `addr1` holds value `old1` and `addr2` holds `old2` when the operation is invoked.

The next section describes type-stable memory management, which facilitates implementing non-blocking synchronization as well as providing several independent benefits to the software structure. Section 3 describes the contention-minimizing data structures which have benefits in performance and reliability for lock-based as well as non-blocking synchronization. Section 4 describes our approach to minimizing the window of inconsistency and the systems benefits of doing so. Section 5 describes the non-blocking synchronization implementation in further detail with comparison to a blocking implementation. Section 6 describes the non-blocking synchronization primitives that we assumed for our approach and a potential hardware implementation. Section 7 describes the performance of our implementation using simulation to show its behavior under high contention. Section 8 describes how our effort relates to previous and current work in this area. We close with a summary of our conclusions and directions for future work.

2 Type-Stable Memory Management (TSM)

Type-stable memory management (TSM) refers to the management of memory allocation and reclamation so that an allocated portion of memory, a *descriptor*, does not change type within some time bound t_{stable}. This is a fancy name for an extension of an old idea. For example, the process descriptors in many operating systems are statically allocated at system initialization and are thus type-stable for the lifetime of the system execution.

```
int DCAS(int *addr1, int *addr2,
         int old1,   int old2,
         int new1,   int new2)
{
 <begin atomic>
 if ((*addr1 == old1) && (*addr2 == old2)) {
   *addr1 = new1;  *addr2 = new2;
   return(TRUE);
 } else {
   return(FALSE);
 }
 <end atomic>
}
```

Figure 1: Pseudo-code definition of DCAS (`Double-Compare-and-Swap`)

Our notion of TSM incorporates three basic extensions to this conventional type of implementation. First, a descriptor remains a valid instance of the type even when it is not active, i.e. on the free list. Second, TSM allows multiple memory allocation pools for the same type. For example, there can be a pool of thread descriptors per cluster of processors on a large-scale multiprocessor to minimize contention between clusters. Finally, the type of a portion of memory can change over time, but only as long as it is type-stable over some time t_{stable}. More specifically, a descriptor has to be inactive for at least t_{stable} before it can be reallocated as a different type[1]. However, for simplicity, we assume an infinite t_{stable} for this discussion.

TSM simplifies the implementation of non-blocking synchronization algorithms. That is, because a descriptor of type `T1` is type-stable, a pointer of type `T1 *` to the descriptor cannot end up pointing to a descriptor of another type as a result of this area of memory being freed and reallocated as type `T2`.

Consider, for example, the code shown in Figure 2 to do a non-blocking deletion from a linked list[2].

The delete operation searches down a linked list of descriptors to find the desired element or detect the end of

[1] An example of a TSM implementation is a collection of descriptors that are stored in a set of page frames which are allocated and released over time. When more descriptors are required, additional page frames can be allocated from the general pool and when the number of descriptors falls, the descriptors may be consolidated into a smaller number of pages and the excessive page frames returned to the general pool. However, the release of page frames to the general pool must be delayed sufficiently to ensure the type-stability property. This delay provides a useful hysteresis to the movement of pages between this descriptor collection and the general page pool.

[2] The list is initialized with a dummy node at the head, thus deletion of the first element works correctly.

```
/* Delete elt */
do {
retry:
  backoffIfNeeded();
  version = list->version;

  for (p = list->head;
       (p->next != elt);
       p = p->next) {
    if (p == NULL) {      /* Not found */
      if (version != list->version)
        { goto retry; }     /* Changed */
      return NULL; /* Really not found */
    }
  }
} while(!DCAS(&(list->version), &(p->next),
              version,          elt,
              version+1,        elt->next))
```

Figure 2: Deletion from the middle of list, protected by DCAS and version number.

the list. If the element is found, the element is atomically deleted from the list by the DCAS operation. The DCAS succeeds only if the list has not been modified since the delete operation started, as determined from the version field.

The code only checks for conflicts once it reaches the desired element or the end of the list. The descriptors are TSM so each pointer is guaranteed to point to a descriptor of this type. Without TSM, the link pointer, `p`, may point to a descriptor that has been deleted and reallocated as a different type. This type error can cause a random bit-field to be interpreted as a pointer, and cause the search to perform incorrectly, raise an exception due to unaligned access, or read a device register. TSM is a simpler and more efficient way of ensuring this type safety than other techniques we are aware of that prevent reallocation (such as automatic garbage collection mechanisms or reference counts), or that detect potential reallocation (such as per-list-element version numbers).

Besides these benefits to non-blocking synchronization, TSM has several important advantages in the construction of modular, reliable, high-performance operating systems. First, TSM is efficient because a type-specific memory allocator can normally allocate an instance of the type faster than a general-purpose allocator can. For example, allocation of a new thread from a free list of (fixed-size) thread descriptors is a simple dequeue operation whereas a general-purpose allocator like `malloc` may have to do a search and subdivision of its memory resources. The class-specific `new` and `delete` operators of C++ support a clean source code representation of TSM. This allocation can be made even more efficient with many types because a free (or inactive) descriptor is already an instance of this type, and so may require less initialization on allocation than a random portion of memory.

Second, TSM aids reliability because it is easier to audit the memory allocation, locating all the descriptors of a given type and ensuring that pointers that are supposed to point to descriptors of a given type actually do so. With fixed-size descriptors, TSM also avoids fragmentation of memory that arises with general-purpose allocators. Fragmentation can cause failure as well as poor performance. Relatedly, TSM makes it easier to regulate the impact of one type of descriptor on the overall system resources. For example, with a collection of descriptors that are allocated dynamically using the page frame approach described above, the number of pages dedicated to this type can be controlled to avoid exhausting the memory available for other uses, both from overallocation and from fragmentation of memory.

TSM also minimizes the complexity of implementing the caching model [7] of descriptors in the operating system kernel. In this approach, the number of descriptors of a given type is limited but an allocation never fails. Instead, as in a cache, a descriptor is made available by its *dirty* data being written back to the higher-level system management and then reused to satisfy the new allocation request. This mechanism relies on limiting the number of descriptors, being able to locate an allocated descriptor to reclaim, and being able to determine the dependencies on these descriptors. TSM simplifies the code in each of these cases.

TSM also allows a modular implementation. From an object-oriented programming standpoint, there can be a base class descriptor manager class that is specialized to each type of descriptor. For example, there is a `CacheKernelObjMan` class in our operating system kernel that provides the basic TSM allocation mechanism, which is specialized by C++ derivation to implement `Thread`, `AddressSpace` `Kernel` and `MemMap` types as well as several other types.

3 Data Structures that Minimize Contention

The Cache Kernel was also designed and implemented to minimize both logical and physical contention to provide for efficient non-blocking synchronization. By *logical contention*, we mean contention for access to data structures that need to be controlled to maintain the consistency and semantics of these data structures. By *physical contention*, we mean the contention for access to shared memory that needs to be controlled to maintain the consistency and semantics of the memory system[3].

[3] Physical contention is separate from logical contention because one can have logical contention without physical contention as well as vice versa, so called *false sharing*. For example, if two shared vari-

Minimizing logical contention with non-blocking synchronization minimizes the overhead of conflicting operations failing and being retried. It also avoids the complexity of complex backoff mechanisms as part of the retry.

Most of our techniques for contention minimization are well-known. For example, one aspect of contention minimization is replicating data structures for each processor. In particular, there are per-processor ready and delay queues in the Cache Kernel, so contention on these structures is limited to signal/interrupt handlers and management operations to load balance, etc. being executed by a separate processor.

Similarly, there is a signal delivery cache per processor which allows a significant number of signals to be delivered by a processor without accessing the shared signal mapping data structure, which cannot be made per-processor without replicating the entire structure. This per-processor "cache" approach is similar to that provided by a per-processor TLB for address translation. The TLB reduces access to the real virtual address space mapping structure, which is necessarily shared among threads in the address space.

Contention on a data structure is also reduced in some cases by structuring it as a multi-level hierarchy. For example, a list that is searched frequently may be revised to be a hash table with a version number or lock per bucket. Then, searches and updates are localized to a single bucket portion of the list, reducing the conflict with other operations, assuming they hash to different buckets. The upper levels of the hierarchy are read-only or read-mostly: descriptors are only added at the leaves.

Physical contention is also reduced by using cache-aligned descriptors. TSM with its restricted allocation of descriptors can also reduce the number of pages referenced as part of scan and search operations, reducing the TLB miss rate, another source of physical contention. Finally, in this vein, commonly updated fields are placed contiguously and aligned to hopefully place them in the same cache line, thereby making the updates more efficient.

The *spatial locality* of data access achieved by these techniques provides significant benefit for synchronization, whether non-blocking or conventional locks. This spatial locality also minimizes the consistency overhead when the system is running across multiple processors, with each caching portions of this shared data. In general, our experience (e.g. [10]) suggests that it is better to (re)structure the data structures to reduce contention rather than attempt to improve the behavior of synchronization techniques under high contention. Low-contention algorithms are simpler and thus easier to get right, and faster as long as contention is actually low.

4 Minimizing the Window of Inconsistency

The Cache Kernel was also structured to minimize the window in which a data structure was inconsistent. This provides *temporal locality* to a critical section. Again, we use familiar techniques. The basic pattern is to read all the values, compute the new values to be written, and then write these new values all at once after verifying that the values read have not changed. Since a structure is generally inconsistent from the time of the first write to the point that the last write completes, removing the computation from this phase minimizes the window of inconsistency. To minimize the cost of verifying that the read values have not changed, we use a version number that covers the data structure and is updated whenever the data structure changes. The use of a version number also avoids keeping track of the actual location read as part of the operation.

The window of inconsistency is also minimized by structuring to minimize physical contention as part of data structure access.

Physical contention increases the time for a processor to perform an operation because it increases the effective memory access time.

These techniques allow efficient non-blocking synchronization. In particular, an update typically consists of a DCAS operation that updates the version number plus one other location, with the version number ensuring that the data structure has not been changed by another concurrent update. That is, the window of inconsistency is reduced to the execution of the DCAS operation itself.

These techniques have other benefits as well. In particular, the reduced window of inconsistency reduces the probability of a failure, such as a thread termination, corrupting the system data structures. They also reduce the complexity of getting critical section code right because it is shorter with fewer separate control paths through it and therefore easier to test. Some of this structuring would be beneficial, if not required, for an implementation using lock-based synchronization because it reduces lock hold time, thereby further reducing contention.

5 Non-Blocking Synchronization Implementation

With the structuring of the Cache Kernel and supporting class libraries described above, non-blocking synchronization is relatively simple to implement. Most data

able can reside in the same cache line unit so there can be physical contention without logical contention if two processor attempt to update the variables simultaneously, each processor updating a separate variable.

structures are collections of fixed-size descriptors. Several collections are queues for service. For example, thread descriptors are queued in the ready queue and a delay queue of their associated processor. Other collections are lookup or search structures such as a hash table with linked list buckets. For example, we organize page descriptors into a lookup structure per address space, supporting virtual-to-physical mapping for the address space.

5.1 The Base Approach

The non-blocking synchronization for these structures follows a common base structure. There is a version number per list. The DCAS primitive is used to atomically perform a write to a descriptor in a list and increment the version number, checking that the previous value of both has not been changed by a conflicting access to the list. Figure 2 illustrated this structure for deleting a descriptor from a list, where the single write to the descriptor was to change the link field of the predecessor descriptor. Inserting a new descriptor D entails initializing D, locating the descriptor in the linked list after which to insert D, writing the D's link field to point to the next descriptor, and then performing the DCAS to write the link field of this prior descriptor to D and to increment the version, checking both locations for contention as part of the update.

Dequeuing a descriptor from a TSM free list is a degenerate case of deletion because the dequeue always takes place from the head. It is possible to optimize this case and use a single CAS to dequeue without a version number. However, with efficient DCAS support, it is attractive to use DCAS with a version number to allow the version number to count the number of allocations that take place. (As another special case, an operation requiring at most two locations for the reads and writes can be updated directly using DCAS. We have used this approach with array-based stacks and FIFO queues.)

Some operations that involve multiple writes to the same descriptor can be performed by creating a duplicate of this descriptor, performing the modifications and then atomically replacing the old descriptor by the new descriptor if the list has not changed since the duplicate descriptor was created. This approach is a variant of Herlihy's general methodology [13] which can convert a sequential implementation of any data structure into a wait-free, concurrent one. However, we use DCAS to ensure atomicity with respect to the entire data structure (the scope of the version number) even though we are only copying a single descriptor[4]. As a variant of this approach, the code can duplicate just a portion of the descriptor, update it and use DCAS to insert it in place of the original while updating a version number. If a thread fails before completing the insertion, we rely on a TSM-based audit to reclaim the partially initialized descriptor after it is unclaimed for t_{stable} time.

As a further optimization, some data structures allow a descriptor to be removed, modified and then reinserted as long as the deletion and the reinsertion are each done atomically. This optimization saves the cost of allocating and freeing a new descriptor compared to the previous approach. This approach requires that other operations can tolerate the inconsistency of this descriptor not being in the list for some period of time. For example, the Cache Kernel signal delivery relies on a list of threads to which a signal should be delivered. A thread fails to get the signal if it is not in the list at the time a signal is generated. However, we defined signal delivery to be best-effort because there are (other) reasons for signal drop so having signal delivery fail to a thread during an update is not a violation of the signal delivery semantics. Programming the higher-level software with best-effort signal delivery has required incorporating timeout and retry mechanisms but these are required for distributed operation in any case and do not add significant overhead [25]. These techniques, related to the transport-layer in network protocols, also make the system more resilient to faults.

Note that just having a search mechanism retry a search when it fails in conjunction with this approach can lead to deadlock. For example, if a signal handler that attempts to access descriptor D, retrying until successful, is called on the stack of a thread that has removed D to perform an update, the signal handler effectively deadlocks with the thread.

5.2 Dealing with Multiple Lists

A descriptor that is supposed to be on multiple lists simultaneously complicates these procedures. So far, we have found it feasible to program so that a descriptor can be in a subset of the lists, and inserted or deleted in each list atomically as separate operations. In particular, all the data structures that allow a descriptor to be absent from a list allow the descriptor to be inserted incrementally.

Overall, the major Cache Kernel [7] data structures are synchronized in a straightforward manner. Threads are in two linked lists: the ready queue and the delay queue. Descriptor free lists are operated as stacks, making allocation and deallocation simple and inexpensive. The virtual to physical page maps are stored in a tree of depth 3 with widths of 128, 128, and 64 respectively. Although the 128 immediate descendants of the root are never deleted, sub-trees below them can be unloaded. Modifications to a map on level 3 are synchronized using DCAS with its parent's version number to make sure that the entire

[4] The basic Herlihy approach involves copying the entire data structure, modifying the copy, and then atomically replacing the old copy with the new copy using CAS, and retrying the entire copy and modifying if there is a conflict. Our approach reduces the allocation and copy cost to a single descriptor rather than the entire data structure but requires DCAS.

subtree has not been modified in conflict with this update. Finally, the Cache Kernel maintains a "dependency map" that records dependencies between objects, including physical to virtual mappings. It is implemented as a fixed-size hash table with linked lists in each bucket. The signal mapping cache structure, (an optimization for signal delivery to active threads), is also a direct mapped hash table with linked lists in each bucket. The majority of uses of single CAS are for audit and counters.

Synchronization of more complex data structures than we have encountered can be handled by each operation allocating, initializing and enqueuing a "message" for a server process that serially executes the requested operations. Read-only operations can still proceed as before, relying on a version number incremented by the server process. Moreover, the server process can run at high priority, and include code to back out of an operation on a page fault and therefore not really block the operation anymore than if the operation was executed directly by the requesting process. The server process can also be carefully protected against failure so the data structure is protected against fail-stop behavior of a random application thread, which may be destroyed by the application.

This approach was used by Pu and Massalin [17]. For example, a general-purpose memory page allocator can be synchronized in this manner, relying on a TSM memory pool to minimize the access to the general allocator. However, in our code to date, the only case of queueing messages for a server module arises with device I/O. This structure avoids waiting for the device I/O to complete and is not motivated by synchronization issues.

Other work has investigated other alternatives or optimizations of this approach, in which helper functions are executed by a new thread if there is work left to complete or rollback by a previous thread accessing this data structure. For example, Israeli et al. [16] describe a non-blocking heap implemented using 2-word `LL`/`SC` along these lines, performing multiple updates as multiple distinct operations. However, to date, we have not needed to employ these so-called *helper* techniques and therefore cannot comment on their actual practicality or utility. Moreover, it seems questionable from a reliability standpoint to have threads from separate address spaces sharing access to complex data structures. These data structures are also more difficult to program and to maintain and often provide marginal performance benefits in practice, particularly when synchronization overhead is taken into account. Their asymptotic performance benefits are often not realized at the scale of typical operating system data structures.

5.3 Comparison to Blocking Synchronization

Much of the structuring we have described would be needed, or at least beneficial, even if the software used blocking synchronization. For instance, TSM has a strong set of benefits as well as contributing to the other techniques for minimizing contention and reducing the window of inconsistency.

We have found that the programming complexity of non-blocking synchronization is similar to conventional blocking synchronization. This differs from the experience of programmers using CAS-only systems. DCAS plays a significant part in the complexity reduction. Using the crude metric of lines of code, a CAS implementation (Valois) of concurrent insertion/deletion from a linked list requires 110 lines, while the corresponding DCAS implementation requires 38 (a non-concurrent DCAS implementation takes 25). The CAS-only implementation of a FIFO queue described in [18] requires 37 lines, our DCAS version only 24. The DCAS versions are correspondingly simpler to understand and to informally verify as correct. In many cases, using DCAS, the translation from a well-understood blocking implementation to a non-blocking one is straightforward. In the simple case described in Figure 2, the initial read of the version number replaces acquiring the lock and the DCAS replaces releasing the lock.

In fact, version numbers are analogous to locks in many ways. A version number has a *scope* over some shared data structure and controls contention on that data structure just like a lock. The scope of the version number should be chosen so that the degree of concurrency is balanced by the synchronization costs. (The degree of concurrency is usually bounded by memory contention concerns in any case). Deciding the scope of a version number is similar to deciding on the granularity of locking: the finer the granularity the more concurrency but the higher the costs incurred. However, a version number is only modified if the data structure is modified whereas a lock is always changed. Given the frequency of read-only operations and the costs of writeback of dirty cache lines, using read-only synchronization for read-only operations is attractive. Finally, version numbers count the number of times that a data structure is modified over time, a useful and sometimes necessary statistic.

Finally, the overall system complexity using blocking synchronization appears to be higher, given the code required to get around the problems it introduces compared to non-blocking synchronization. In particular, special coding is required for signal handlers to avoid deadlock. Special mechanisms in the thread scheduler are required to avoid the priority inversion that locks can produce. And, additional code complexity is required to achieve reliable operation when a thread can be terminated at a random time. For example, some operations may have to be implemented in a separate server process.

A primary concern with non-blocking synchronization is excessive retries because of contending operations. However, our structuring has reduced the probability of contention and the conditional load mechanism described in the next section can be used to achieve behavior similar to lock-based synchronization.

6 Non-blocking Synchronization Primitives

Our approach assumes an efficient implementation of DCAS functionality. In this section, we briefly outline an instruction set extension to the `load-linked`/`store-conditional`) instructions to support DCAS. (A software implementation is discussed in Section 6.1.) With a processor supporting `load-linked` (LL) and `store-conditional` (SC) instructions, add two instructions:

1. LLP (load-linked-pipelined): load and link to a second address after a LL. This `load` is linked to the following SCP.

2. SCP (store-conditional-pipelined): Store to the specified location provided that no modifications have been made to either of the memory cells designated by *either* of the most recent LL and LLP instructions and these cache lines have not been invalidated in the cache of the processor performing the SCP.

If a `LLP`/`SCP` sequence nested within an LL/SC pair fails, the outer LL/SC pair fails too.

DCAS is then implemented by the instruction sequence shown in Figure 3 (using R4000 instructions in addition to the LL/SC(P) instructions). The LL and LLP instructions in lines 1 and 2 "link" the `load`s with the respective `store`s issued by the following SC and SCP instructions. Lines 3 and 4 verify that `(T0)` and `(T1)` contain `V0` and `V1`, respectively. The SCP and SC in lines 5 and 6 are conditional. They will not issue the `store`s unless `(T0)` and `(T1)` have been unchanged since lines 1 and 2. This guarantees that the results of CAS in lines 3 and 4 are still valid at line 6, or else the SC fails. Further, the `store` issued by a successful SCP is buffered pending a successful SC. Thus, SC in line 6 writes `U1` and `U0` to `(T1)` and `(T0)` atomically with the comparison to `V0` and `V1`[5].

```
/*
 * If (T0) == V0, and (T1) == V1, then
 * atomically store U0 and U1 in T0 and T1
 */
DCAS(T0, T1, V0, V1, U0, U1)
   ;; Get contents of addresses in registers.
1  LL        T3, (T1)
2  LLP       T2, (T0)
   ;; Compare to V0 and V1. If unequal, fail.
3  BNE       T2, V0, FAIL
4  BNE       T3, V1, FAIL
   ;; If equal, and unchanged since LOAD,
   ;;     store new values
5  SCP       U0, (T0)
6  SC        U1, (T1)
   ;; Success of SC and SCP is stored in U1
   BLEZ      U1, FAIL
   ...
FAIL:
```

Figure 3: DCAS Implementation using `LL`/`SC` and `LLP`/`SCP`. Success or failure of `SC` (and thus of the DCAS operation) is returned in U1 or whatever general register holds the argument to `SC`. 1 denotes success, 0 failure. If the next instruction tries to read U1, the hardware interlocks (as it already does for `LL/SC`) if the result of `SC` is not already in U1.

We have worked out a detailed design for the implementation of these two instructions in a RISC processor such as the R4000 but the description is omitted for brevity.

6.1 Software Implementation of DCAS

DCAS functionality can be implemented in software using a technique introduced by Bershad [4]. DCAS is implemented using a lock known to the operating system. If a process holding this locks is *delayed* by a context switch, the operating system rolls back the process out of the DCAS procedure and releases the lock. The rollback procedure is relatively simple because the DCAS implementation is simple and known to the operating system. Moreover, the probability of a context switch in the middle of the DCAS procedure is low because it is so short, typically a few instructions. Thus, the rollback cost is incurred infrequently.

This technique can be used more generally to implement other primitives such as n-location CAS. We focus on DCAS implementation because the primary relation to our work is offering a software implementation of DCAS as an alternative to our proposed hardware support. It also seems simpler to just implement rollback for DCAS compared to more general primitives.

This approach has the key advantage of not requiring hardware extensions over the facilities in existing systems. Moreover, its performance may be comparable to our hardware extensions, especially on single processors or small-scale multiprocessors. Further measurements are required here. However, there are a few concerns.

[5] Given data structures that are protected by a version number, te DCAS is actually a `Compare-And-Double-Swap` (CADS) — the second value cannot have changed if the version number is unchanged. In these cases a minor optimization is possible and line 4 can be deleted.

First, there is the cost of locking. The straight-forward implementation requires the DCAS procedure to access a common global lock from all processes. In a multi-level memory with locks in memory, the memory contention between processors for this lock can be significant. For example, the data structure may be in a shared segment that is mapped in by two independent processes. If the locks are associated with each DCAS instance, there is more cost and complexity to designate the locks and critical section to the operating system and to implement the rollback. The locking and unlocking also modifies the cache line containing the lock, further increasing the cost of this operation because writeback is required.

Second, Bershad's approach requires rereading the two locations from memory as well as an extra read and write to set the lock and write to clear the lock.

Third, on multiprocessors, care must be used by readers of shared data structures if they want to support unsynchronized reads. Without depending on the lock, readers can see intermediate states of the DCAS, and read tentative values that are part of a DCAS that fails. Requiring synchronization for reads significantly increases contention on the global lock. Note that in many cases TSM reduces the danger of unsynchronized reads because the reads cannot cause type errors. Writes are protected by the global lock, and the final DCAS will detect that the unsynchronized reads were suspect, and fail. Systems that provide hardware DCAS require no additional read synchronization beyond that performed automatically by the memory system. Further experience and measurements are required to determine whether this is a significant issue on real systems.

Finally, the Bershad mechanism seems harder to test under all conditions. For instance, it is possible that one of the write operations that the rollback needs to undo is to an area of memory that has been paged out or that one of the addresses is illegal. The system also needs to ensure that a thread is rolled back out of any DCAS critical section if it is terminated. We believe our hardware implementation is simpler to verify and naturally operates on top of the virtual memory management of the system and on top of directly accessible physical memory at the lowest level of the system software. It is of concern that a minor change to the software mechanisms in Bershad's scheme could result in very subtle errors in execution that could go undetected in a system for a long period of time.

6.2 Hardware Contention Control

As a further extension, a processor can provide a conditional load instruction or `Cload`. The `Cload` instruction is a load instruction that succeeds only if the location being loaded does not have an *advisory lock* set on it, setting the advisory lock when it does succeed.

With `Cload` available, the version number is loaded initially using `Cload` rather than a normal load. If the `Cload` operation fails, the thread waits and retries, up to some maximum, and then uses the normal load instruction and proceeds. This waiting avoids performing the update concurrently with another process updating the same data structure. It also prevents potential starvation when one operation takes significantly longer than other operations, causing these other frequently occuring operations to perpetually abort the former. It appears particularly beneficial in large-scale shared memory systems where the time to complete a DCAS-governed operation can be significantly extended by wait times on memory because of contention, increasing the exposure time for another process to perform an interfering operation. Memory references that miss can take 100 times as long, or more, because of contention misses. Without Ĉload, a process can significantly delay the execution of another process by faulting in the data being used by the other process and possibly causing its DCAS to fail as well.

The cost of using `Cload` in the common case is simply testing whether the `Cload` succeeded, given that a load of the version number is required in any case.

`Cload` can be implemented using the cache-based advisory locking mechanism implemented in ParaDiGM [8]. Briefly, the processor advises the cache controller that a particular cache line is "locked". Normal `load`s and `store`s ignore the lock bit, but the `Cload` instruction tests and sets the cache-level lock for a given cache line or else fails if it is already set. A store operation clears the bit. This implementation costs an extra 3 bits of cache tags per cache line plus some logic in the cache controller. Judging by our experience with ParaDiGM, `Cload` is quite feasible to implement.

7 Performance

The performance on the ParaDiGM experimental multiprocessor is first discussed. We then discuss results from simulation indicating the performance of our approach under high contention. Finally, we discuss aspects of overall system performance.

7.1 Experimental Implementation

The operating system kernel and class libraries run on the ParaDiGM architecture [8]. The basic configuration consists of 4-processor Motorola 68040-based multiprocessors running with 25 MHz clocks. The 68040 processor has a DCAS instruction, namely CAS2. This software also runs with no change except for a software implementation of DCAS, on a uniprocessor 66 MHz PowerPC 603. We have not implemented it on a multiprocessor PowerPC-based system to date.

Kernel synchronization uses DCAS in 27% of the critical sections and otherwise CAS. However, the DCAS uses are performance-critical, e.g. insert and deletion for key queues such as the ready queue and delay queue. The only case of blocking synchronization is on machine startup, to allow Processor 0 to complete initialization before the other processors start execution.

The overhead for non-blocking synchronization is minimal in extra instructions. For example, deletion from a priority queue imposes a synchronization overhead of 4 instructions compared to no synchronization whatsoever, including instructions to access the version number, test for DCAS success and retry the operation if necessary. This instruction overhead is comparable to that required for locked synchronization, given that lock access can fail thus requiring test for success and retry.

The Motorola 68040's CAS2 [26] is slow, apparently because of inefficient handling of the on-chip cache so synchronization takes about 3.5 microseconds in processor time. In comparison, spin locks take on average 2.1 μsecs and queuelocks take about 3.4 μsecs. In contrast, the extended instructions we propose in Section 6 would provide performance comparable to any locking implementation. In particular, it requires 16 extra instructions (including the required no-ops) plus an implicit `SYNC` in an R4000-like processor. A careful implementation would allow all instructions other than the `SYNC` to execute at normal memory speed. The performance would then be comparable to the roughly 24 instruction times required by the R4000 lock/unlock sequence. Figure 4 compares the overhead in terms of instruction times.

Operation	Instruction Times
DCAS using `CAS2` on 68040	114
DCAS using `LLP`/`SCP`	26
SGI R3000 lock/unlock	70
R4000 lock/unlock	24

Figure 4: Approximate instruction times of extra overhead to synchronize deletion from a priority queue. This overhead does not include the backoff computation.

7.2 Simulation-Based Evaluation

The actual contention for the kernel data structures in our current implementation is low and we did not have the ability to create high contention at the time of writing.

To understand how our system behaves under heavy load, we have simulated insertion/deletion into a singly linked list under loads far heavier than would ever be encountered in the Cache Kernel.

Our simulation was run on the Proteus simulator [5], simulating 16 processors, a cache with 2 lines per set, a shared bus, and using the Goodman cache-coherence protocol. All times are reported in cycles from start of test until the last processor finishes executing. Memory latency is modeled at 10 times the cost of a cache reference. The cost of a DCAS is modeled at 17 extra cycles above the costs of the necessary memory references. The additional cost of a CAS over an unsynchronized instruction referencing shared memory is 9 cycles.

Four algorithms were simulated:

1. *DCAS/Cload*: Our DCAS algorithm with contention controlled by advisory locking, as implemented on Paradigm.

2. *DCAS/A&F*: DCAS algorithm with contention controlled by OS intervention as proposed by Allemany and Felten [1] and described in Section 8.4.

3. *CAS*: An implementation using only CAS and supporting a much higher degree of concurrency based on a technique by Valois [24] [6].

4. *SpinLock*: Spin-lock with exponential back-off as a base case.

Each test performed a total of 10,000 insertions and deletions, divided evenly between all processes. We varied the number of processors from 1 to 16 and the number of processes per processor from 1 to 3. We also controlled the rate of access to the list by each process by doing local "work" between the insertion and deletion. The work varied from 20 to 2000 cycles.

These simulations indicate that the Cache Kernel DCAS algorithms perform as well or better than CAS or spin locks.

Figure 5 shows the performance with 1 process per processor, and minimal work between updates. The basic cost of 10,000 updates is shown at $N = 1$, where all accesses are serialized and there is no synchronization contention or bus contention. At $N = 1$, cache contention due to collisions is small, the hit rate in the cache was over 99% in all algorithms. At more than one processor, even assuming no synchronization contention and no bus contention, completion time is significantly larger because the objects must migrate from the cache of one processor to another. When processes/processor = 1 no processes are preempted. In this case the difference between the non-concurrent algorithms is simply the bus contention and the fixed overhead because we are not modelling page faults. All degrade comparably, although DCAS/A&F suffers from bus-contention on the count of active threads. The Valois algorithm using CAS exploits concurrency as the number of processors increase but the overhead is large relative to the simpler algorithms. The bus and memory contention are so much greater that the

[6] It was necessary to derive our own version of the algorithm, as the algorithm presented in [24] is not strictly correct. This is the natural result of the complicated contortions necessary when using only CAS. The DCAS algorithm is relatively straightforward.

A few processors such as the Motorola 68040 provide a multi-word atomic instruction but that functionality is rare and is not present in any RISC processor to our knowledge. The RISC-like extension that we propose in Section 6 suggests that it is feasible to support in modern processors. The CISC approach does not appear viable with most current and future processors and seems likely to die out with the current processors that support it.

Transactional Memory [12] provides hardware support for multiple-address atomic memory operations. It is more general than DCAS but comes at a correspondingly higher cost. The proposed hardware implementation requires six new instructions, a second set of caches in the processor, twice the storage for cache lines actively involved in a transaction, and a more complicated "commit" protocol. Double `LL/SC` appears to be a more practical solution because DCAS functionality is sufficient and significantly simpler to implement.

Oklahoma Update [21] provides an alternate implementation of multiple-address atomic memory operations. Rather than duplicating entire cache lines involved in transactions (as Transactional Memory does), Oklahoma Update requires only a reservation register per *word* used in their version of `Load Linked`. This register contains flags plus two words (and optionally two more). This contrasts with our implementation which requires a "link address retained" register per synchronized word and a single cache-line buffer for the delayed `SCP`. Our design can also work with a word register instead of an entire cache line to buffer the `SCP`. However, this approach adds complexity to the chip's logic, slows down the `SC` and increases the time the cache is locked so the savings are questionable. The Oklahoma Update attempts to implement some features in hardware (e.g. exponential backoff) which are better done in software, and which needlessly increase the complexity and size of the chip. Also, buffering of certain requests that come in during the "pre-commit" phase can cause two processors with non-interfering reservation sets to delay each other[8].

These different designs arise because of different assumptions regarding the number of memory locations that should be atomically updatable at one time. Transactional Memory paper conjectures between 10 and 100 and Oklahoma Update places the knee at 3 or 4. In general, more locations are better and more powerful. However, our implementation at 2 (DCAS) is by far the simplest extension to existing processor designs. A key contribution of our work is experience that indicates that DCAS is sufficient for practical performance, making the extra hardware complexity of the other schemes unnecessary.

8.4 Operating System Support

Allemany and Felten [1] reduce useless concurrency with OS support to provide the same functionality that we support in hardware using cache-based advisory locking. The method is a variation on the technique of Bershad discussed in Section 6.1. They propose incrementing a counter of active threads on entrance to a critical section, and decrementing on exit. The OS decrements the counter while an active thread is switched out. Processes must wait until the count of active threads is below some threshold (1, in our case) before being allowed to proceed. Delayed processes do not excessively delay other processes because the count is decremented by the OS.

These techniques appear valuable for systems without hardware support for advisory locking and in fact their approach works better than ours under high contention. However, hardware advisory locking and conditional load are more resilient to processor failure and have lower overhead in the low-contention case. As with hardware versus software DCAS, the hardware implementation is simple and fast; further measurements are required to determine if it is compellingly so.

In other work, Israeli and Rappaport [15] implement n-way atomic `Compare and Swap` and n-way `LL/SC` for P processors out of single CAS. However, this approach is primarily of theoretical interest because it requires a large amount of space (at least P bits for *every* word in the shared memory), requires words to be P bits wide, takes $O(P)$ to execute, and only interlocks against other multi-word atomic instructions. Anderson and Moir [2] improve upon this, requiring only realistic sized words, $O(1)$ time, but still requiring a prohibitively large amount of space.

Finally, Software Transactional Memory [19] is an attempt to implement Transactional Memory in software, depending only on `LL/SC` . Unfortunately, their implementation will not work correctly on existing implementations of `LL/SC` because their code (`AcquireOwnerships`) depends on the ability to interleave two outstanding `LL/SC`'s simultaneously, which is not supported. The `LLP/SCP` instructions we proposed *would* enable their techniques to be used to provide software transactional memory for multiple, independently chosen, words of memory. However, the space and computational overhead in their implementation is excessive for general use[9]. Moreover, the STM operations are only atomic with respect to other STM operations, and not to general reads and writes.

[8] Consider processors P_1, P_2 and P_3. P_1 accesses cache lines Y,Z, P_2 X,Y, and P_3 W,X (addressed in ascending alphabetical order). P_1 and P_3 should not interact. However, if P_1 holds Y and Z and P_2 holds X, then when P_2 asks P_1 for Y, P_2 stalls, and buffers P_3's request for X. Thus, P_1 delays P_3. Longer chains can be constructed.

[9] Their scheme requires twice the memory for *every* possibly shared location and extra overhead of at least a factor of three for reads and writes even in the case of no contention.

9 Concluding Remarks

Our experience suggests that there is a powerful synergy between non-blocking synchronization and several good structuring techniques for the design and implementation of an operating system and supporting run-time libraries. Non-blocking synchronization significantly reduces the complexity and improves the performance of software in the *signal-rich* environment implemented by the Cache Kernel and supporting class libraries. Moreover, the structuring techniques we have used to achieve our overall system design goals facilitate implementing non-blocking synchronization. The biggest problem has been inadequate performance of the non-blocking synchronization instructions.

This work makes several contributions. First, we show that careful design and implementation of operating system software for efficiency, reliability and modularity makes implementing simple, efficient non-blocking synchronization far easier. In particular, type-stable memory (TSM), contention-minimizing data structuring and minimal inconsistency window structuring are important for all these reasons. These techniques are beneficial even with blocking synchronization and yet significantly reduce the complexity and improve the performance of non-blocking synchronization. Conversely, non-blocking synchronization has significant advantages in the signal-centric design of the Cache Kernel and its associated libraries, especially with the large amount of conventional operating system functionality that is implemented at the library, rather than kernel, level.

Second, we describe a number of techniques for implementing non-blocking synchronization using TSM, version numbers and DCAS. These techniques are simple to write, read, and understand, and perform well. *in contrast to the CAS* Our experience suggests that good DCAS support is sufficient for a practical non-blocking OS and run-time system implementation, and that single CAS is not sufficient. In fact, lack of efficient DCAS support in systems is a potential impediment to using our techniques.

Fortunately, our proposed hardware implementation indicates that it is feasible to implement efficient DCAS functionality in a modern processor with minimal additional complexity and full compatibility with the load-store architecture. The conditional load capability coupled to cache-based advisory locking further improves the hardware support, providing the advantages of locking in a lock-free implementation. The existence of software implementations of DCAS and contention reduction demonstrates that our approach is reasonable even on platforms lacking hardware support.

Efficiently supported DCAS would allow fully-synchronized standard libraries and operating system software to be portable across multiprocessors and uniprocessors without extra overhead or code complication. It would allow parallel architectures to use software developed for uniprocessors, relying on the (non-blocking) synchronization required for signals to handle serialization in the parallel processing context. This would significantly reduce the software bottleneck that has slowed the deployment of parallel processing to date.

Further work is required to evaluate the merits of hardware support for DCAS versus various software alternatives, particularly for overall system performance. Further work is also required to validate our experience that DCAS is in fact adequate in practice. However, our experience to date convinces us that the non-blocking approach is an attractive and practical way to structure operating system software. Locks will become more problematic as signals are used more extensively in libraries, synchronization becomes finer grained, and as the cost of memory delays and descheduling become even higher relative to processor speed. We hope our work encourages additional efforts in this area.

References

[1] J. Allemany and E.W.Felton, Performance issues in non-blocking synchronization on shared memory multiprocessors. *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pp 125-134, August 1992.

[2] J.H. Anderson and M. Moir, Universal Constructions for Multi-Object Operations, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 184-193, August 20-23, 1995

[3] G. Barnes, A Method for Implementing Lock-Free Shared Data Structures *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures* 1993

[4] B.N. Bershad, Practical considerations for non-blocking concurrent objects. *Proceedings 13th IEEE International Conference on Distributed Computing Systems*, Los Alamitos CA, IEEE Computer Society Press, pp 264-273, May 25-28, 1993.

[5] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl, "PROTEUS: A High-Performance Parallel-Architecture Simulator", Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[6] D.R. Cheriton, The V Distributed System. *Communications of the ACM*, 31(3), pp 314-333, March 1988

[7] D.R. Cheriton and K. Duda. A Caching Model of Operating System Kernel Functionality. *Proceedings of 1st Symposium on Operation Systems Design and Implementation*, Monterey, CA, pp 179-193, Nov 14-17, 1994.

[8] D.R. Cheriton, H. Goosen, and P. Boyle, ParaDiGM: A highly scalable shared-memory multi-computer architecture. *IEEE Computer*, 24(2), February 1991.

[9] D.R. Cheriton and R. Kutter. Optimizing memory-based messaging for scalable shared memory multiprocessor architectures. To appear in *USENIX Computer Systems Journal* 1996. (available as Stanford Computer Science Technical Report CS-93-123, December 1993.)

[10] D.R. Cheriton, H. Goosen, and P. Machanick, Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pp 23-31, Tokyo, April 1991.

[11] Joseph Heinrich. MIPS R4000 User's Manual, PTR Prentice Hall, Englewood Cliffs NJ, 1993

[12] M.P. Herlihy and J.E.B. Moss. Transactional Memory: Architectural support for lock-free data structures. *1993 20th Annual Symposium on Computer Architecture* San Diego, Calif. pp. 289-301. May 1993.

[13] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), pp 123-149, January, 1991

[14] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects *ACM Transactions on Programming Languages and Systems*, 15(5), 745-770, November, 1993

[15] A. Israeli and L. Rappaport, Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives, *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, pp 151-160, August 14-17, 1994

[16] A. Israeli and L. Rappaport, Efficient wait-free implementation of a concurrent priority queue *7th Intl Workshop on Distributed Algorithms '93,* Lausanne, Switzerland, *Lecture Notes in Computer Science 725*, Springer Verlag, pp 1-17, Sept. 1993

[17] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[18] M. Michael and M. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, pp 267-276, May 1996.

[19] N. Shavit and D. Tovitov, Software Transactional Memory, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 204-213, August 20-23, 1995

[20] R. Sites, ed., DEC Alpha Architecture, Digital Press, Burlington, Mass. 1992

[21] J. Stone, H. Stone, P. Heidelbergher, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, vol 1, no.4, pp 58-71, November, 1993

[22] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Fifth International Conference on Architectural Support for Programminlg Languages and Operating Systems*, pp 162-174, October 1992

[23] J. Turek, D. Shasha and S. Prakash. Locking without blocking: Making Lock-Based Concurrent Data Structure Algorithms Non-Blocking. *Proceedings of the 1992 Principles of Database Systems* pp 212-222, 1992.

[24] J. Valois, Lock-Free Linked Lists Using Compare-and-Swap, *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Ont. Canada, pp 214-222, August 20-23, 1995

[25] M. Zelesko and D. R. Cheriton, Specializing Object Oriented RPC for Functionality and Performance, *Proceedings 16th IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society Press, May 27-30, 1996.

[26] M68000 Family Programmer's Reference Manual, Motorola, Inc. 1989

[27] PowerPC 601 RISC Mircroprocessor User's Manual, Motorola Inc, 1993