Non-Blocking Synchronization and Object-Oriented Operating System Design

C. Scott Ananian
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
`cananian@csail.mit.edu`

Abstract

Mutual exclusion is typically employed for multiprocess synchronization. However, Massalin and Pu [MP91], and Greenwald and Cheriton [GC96] have implemented complete operating systems using non-blocking synchronization instead of mutual exclusion. Although most operating systems are implemented using procedural languages, both of these systems have significant object-oriented characteristics. We show the interaction of non-blocking synchronization and object-oriented language design for operating systems, and present a proposal for an implementation of the Java programming language with non-blocking synchronization based on functional arrays. We present an experimental plan to run portions of a Java operating system under our system and show the benefits.

1 Introduction

Multiprocess synchronization is typically done with mutual exclusion: only one processor at a time may execute "critical regions" of the software. This can lead to low parallelism, fault-intolerance, and unexpected dependencies between tasks. Non-blocking synchronization has been shown to have competitive performance with mutual exclusion mechanisms, and prevents the failure, delay, or execution of one task from interfering with the progress of other parallel tasks. This technique is not yet widespread.

Massalin and Pu built the multiprocessor version of their Synthesis kernel [MP91, MP92] using non-blocking synchronization to avoid perceived performance problems with mutual exclusion locks. They succeeded in demonstrating the practicality of non-blocking synchronization and achieved very high performance using a small number of non-blocking data structures in a constrained manner. Greenwald and Cheriton used non-blocking synchronization in a slightly more general manner to implement their Cache Kernel [CD94, GC96].

Object-orientation[1] is important in both Synthesis and the Cache Kernel. The Cache Kernel is implemented in C++ and exports only three basic object types [CD94], while Synthesis implements "quajects" leveraging only macro-assembler [PM91]. Synchronization is associated with separate objects in both systems, informally following the basic principles of *monitor synchronization* laid out by the Emerald system [JS91].[2] The follow-on project to Synthesis, Synthetix, made a strong argument that strong object encapsulation was a necessary requirement for a high-performance adaptive kernel [BW95]; other recent work on object-oriented operating systems has explored replacing more of the basic operating system protection mechanisms with compiler-enforced software invariants [BSP$^+$95]. Often significant language co-design issues are involved, as evidenced by the DrScheme [FFKF99] and Lisaac [SC02] systems.

```
Writer:                 Reader:
  v1++;                   do {
  write;                    temp = v2;
  v2++;                     read;
                          } while (temp != v1);
```

Figure 1: Lamport's single-writer multiple-reader non-blocking synchronization algoritm [Lam77].

The Java programming language incorporates monitor synchronization, which is typically implemented using mutual exclusion. We outline a non-blocking implementation of Java synchronization based on fast functional arrays, and show how this can be used in the construction of an object-oriented operating system. Monitor synchronization with encapsulated objects lends itself well to non-blocking implementations, and compiler analyses can be used to reduce the overhead of synchronization still further. A plan for experiments including benchmarks of portions of a Java operating system is presented.

The next section describes the history, advantages and implementations of non-blocking synchronization. Section 3 will discuss the Synthesis and Cache Kernels in detail. Section 4 will discuss object-oriented operating systems in general, including language design issues. Section 5 will describe our proposal for extending Java with non-blocking monitor synchronization. Section 6 will present our experimental plan. We will close with a summary of the research and our conclusions.

[1] Jonathan Rees, among others, has criticized the increasing vagueness of the term "object-oriented". We will primarily use the term to describe encapsulation, protection, and a sum-of-product-of-function pattern (his items 1, 2, and 9 in [Ree01]).

2 Non-blocking synchronization

Lamport presented the first alternative to synchronization via mutual exclusion in [Lam77], for a limited situation involving a single writer and multiple readers. Lamport's technique, presented in Figure 1, relies on reading guard elements v_1 and v_2 in an order opposite to that in which they are written, guaranteeing that a consistent data snapshot can be recognized. The writer always completes its part of the algorithm in a constant number of steps; readers are guaranteed to complete only in the absence of concurrent writes.

Herlihy formalized *wait-free* implementations of concurrent data objects in [Her88]. A wait-free implementation guarantees that any process can complete any operation in a finite number of steps, regardless of the activities of other processes. Lamport's algorithm, for example, is not wait-free because readers can be delayed indefinitely. Wait-free algorithms typically involve "recursive helping," whereby active processes can complete operations on behalf of stalled processes, ensuring that all operations are eventually completed.

Massalin and Pu introduced the term *lock-free* to describe algorithms with weaker progress guarantees. A lock-free implementation guarantees only that *some* process will complete in a finite number of steps [MP91]. Unlike a wait-free implementation, lock-freedom allows starvation. Since other simple techniques can be layered to prevent starvation (for example, exponential backoff), simple lock-free implementations are usually seen as worthwhile practical alternatives to more complex wait-free implementations.

An even weaker criterion, *obstruction-freedom*, was introduced by Herlihy, Luchangco, and Moir in [HLM03]. Obstruction-freedom only guarantees progress for threads executing in isolation; that is, although other threads may have partially completed operations, no other thread may take a step until the isolated thread completes. Obstruction-freedom not only allows starvation of a particular thread, it allows contention among threads to halt all progress in all threads indefinitely. External mechanisms are used to reduce contention (thus, achieve progress) including backoff, queueing, or timestamping.

Revisiting Lamport's algorithm, we conclude it is neither lock-free nor obstruction-free, because halting the writer between the guard increments will prevent readers from ever getting a consistent snapshot.

We will use the term *non-blocking* to describe generally any synchronization mechanism which doesn't rely on mutual exclusion or locking, including wait-free, lock-free, and obstruction-free implementations. We will be concerned mainly with lock-free algorithms.[3]

[2]Note, however, that race-freedom via monitor synchronization is not *sufficient* to prevent unexpected concurrent behavior [FQ03], nor is it *necessary*. It is certainly a helpful structuring principle and reasoning aid.

[3]Note that some authors use "non-blocking" and "lock-free" as synonyms, usually meaning what we here call *lock-free*. Others exchange our definitions for "lock-free" and "non-blocking", using lock-free as a generic term and non-blocking to describe a specific class of implementations. As there is variation in the field, we choose to use the parallel construction *wait-free*, *lock-free*, and *obstruction-free* for our three specific progress criteria, and the dissimilar *non-blocking* for the general class.

2.1 Advantages over mutual exclusion

Non-blocking synchronization offers a number of advantages over mutual exclusion within critical regions. Foremost for the concerns of this paper is fault-tolerance: a process which fails while holding a lock within a critical region can prevent all other non-failing processes from ever making progress. In an operating system context, this means that thread termination must be done very carefully to avoid inadvertently killing a process within a critical region. If a thread dies while holding a user-mode lock, all other threads in its address space may deadlock; if it dies while holding a kernel lock, the whole system may crash. Although one can use external means to recognize orphaned locks and release them, it is in general not possible to restore the locked data structures to a consistent state after such a failure. Non-blocking synchronization offers a graceful means out of these troubles, as non-progress or failure of any one thread will not affect the progress or consistency of other threads or the system. These fault-tolerant properties are even more relevant in distributed systems where entire nodes may fail without warning.

Non-blocking synchronization offers performance benefits as well. Even in a failure-free system, page faults, cache misses, context switches, I/O, and other unpredictable events may result in delays to the entire system when mutual exclusion is used; non-blocking synchronization allows undelayed processes or processors to continue to make progress. In loosely coupled asynchronous systems such unexpected delays are the norm, rather than the exception.

Real-time systems have other problems with mutual exclusion. A low-priority task which acquires a lock and is then delayed may hold up higher-priority tasks which contain critical regions protected with the same lock. This situation is called *priority inversion*, and is responsible for a number of high-profile system failures, including whole-system resets during the Mars Pathfinder mission [Jon97]. Non-blocking synchronization can guarantee that the high-priority task makes progress.[4]

2.2 Efficiency

Herlihy presented the first *universal* method for wait-free concurrent implementation of an arbitrary sequential object [Her88, Her91]. This original method was based on a *fetch-and-cons* primitive, which atomically places an item on the head of a list and returns the list of items following it; all concurrent primitives capable of solving the n-process consensus problem—*universal* primitives—were shown powerful enough to implement *fetch-and-cons*. In Herlihy's method, every sequential operation is translated into two steps. In the first, *fetch-and-cons* is used to place the name and arguments of the operation to be performed at the head of a list, returning the other operations on the list. Since the state of a deterministic object is completely determined by the history of operations performed on it, applying the operations returned in order from last to first is sufficient to locally reconstruct the object state prior to our operation.

[4]Note that the progress guarantees made are different for wait-free, lock-free, and obstruction-free algorithms. For example, priority-inversion can still occur on obstruction-free implementations if a lower priority thread contends persistently for the resource. On a uniprocessor, a valid solution in this case might be to simply not interleave executions of tasks with differing priorities; obstruction-freedom then guarantees that the high-priority task "in isolation" will make progress.

We then use the prior state to compute the result of our operation without requiring further synchronization with the other processes.

This first universal method was not very practical, a shortcoming which Herlihy soon addressed [Her93]. In addition, his revised universal method can be made lock-free, rather than wait-free, resulting in improved performance. In the lock-free version of this method, objects contain a shared variable holding a pointer to their current state. Processes begin by loading the current state pointer and then copying the referenced state to a local copy. The sequential operation is performed on the copy, and then if the object's shared state pointer is unchanged from its initial load it is atomically swung to point at the updated state.

Herlihy called this the "small object protocol" because the object copying overhead is prohibitive unless the object is small enough to be copied efficiently (in, say, $O(1)$ time). He also presented a "large object protocol" which requires the programmer to manually break the object into small blocks, after which the small object protocol can be employed. This trouble with large objects is common to many non-blocking implementations; our solution is presented in Section 5.

Barnes provided the first universal non-blocking implementation method which avoids object copying [Bar93]. He eliminates the need to store "old" object state in case of operation failure by having all threads cooperate to apply operations. For example, if the first processor begins an operation and then halts, another processor will complete the first's operation before applying its own. Barnes proposes to accomplish the cooperation by creating a parallel state machine for each operation, so that each thread can independently try to advance the machine from state to state and thus advance incomplete operations.[5] Although this avoids copying state, the lock-step cooperative process is extremely cumbersome and does not appear to have ever been implemented. Furthermore, it does not protect against errors in the implementation of the operations, which could cause *every* thread to fail in turn as one by one they attempt to execute a buggy operation.

Alemany and Felten [AF92] identified two factors hindering the performance of non-blocking algorithms to date: resources wasted by operations that fail, and the cost of data copying. Unfortunately, they proceeded to "solve" these problems by ignoring short delays and failures and using operating system support to handle delays caused by context switches, page faults, and I/O operations. This works in some situations, but obviously suffers from a bootstrapping problem as the means to implement an operation system.

Although lock-free implementations are usually assumed to be more efficient that wait-free implementations, LaMarca [LaM94] showed experimental evidence that Herlihy's simple wait-free protocol scales very well on parallel machines. When more than about twenty threads are involved, the wait-free protocol becomes faster than Herlihy's lock-free small-object protocol, three OS-aided protocols of LaMarca and Alemany and Felten, and a *test-and-Compare&Swap* spin-lock.

[5]It is interesting to note that Barnes' cooperative method for non-blocking situation plays out in a real-time system very similarly to priority inheritance for locking synchronization.

3 Lock-free operating systems

The Synthesis Kernel V.1 [MP91, MP92] and the Cache Kernel [CD94, GC96] are complete operating systems implemented using only non-blocking synchronization. Motivations varied: Synthesis appears to have adopted non-blocking synchronization in a quest for high performance, while the motivation in the case of the Cache Kernel seems to have been the improved system decoupling and modularity achievable without locks. In both cases, the implementation of non-blocking synchronization interacted with object-oriented features of the operating system design.

3.1 The Synthesis kernel

The Synthesis Kernel V.0 was built to explore in-kernel code synthesis for improved performance [PMI88]. The code synthesizer generates specialized code for frequently-executed kernel functions, using three optimizations:

- **Factoring Invariants** is a special case of partial evaluation which applies constant folding and constant propagation using constants gleaned from dynamic properties and data structure traversals. For example, the Synthesis implementation of the `open` system call will invoke the code synthesis engine to generate a specialized implementation for `read`. The `read` implementation is less than 100 instructions long and is specialized for the access mode, file descriptor, thread id, and device known after the `open`. Factoring invariants eliminates all access checking and device handler dispatch, using precompiled templates where possible.
- **Collapsing Layers** seeks to make abstraction cheap by eliminating function call overhead between abstraction boundaries. This is done using procedure inlining and buffer/move coalescing to avoid unnecessary copies. UNIX system calls emulated in Synthesis consists of little more than invocations of the appropriate Synthesis equivalents. Collapsing Layers is used to inline the Synthesis functionality into the emulation routine, avoiding the extra layer of call indirection. Similarly, the network protocol stack eliminates unnecessary copying by sharing message buffer space allocated at the top level with all collapsed lower levels.
- **Executable Data Structures** explicitly code for iteration, eliminating the need to interpret the data structure. For example, the Synthesis run queue is implemented as a collection of routines each of which switches to the "next" thread context, hard-coded as the destination of a jump instruction to eliminate even address load overhead. The routines are directly installed as the appropriate preemption signal handlers.

The code synthesis technique was very successful, achieving as much as 56-fold speed improvement over a conventional SUNOS kernel on the same hardware for small byte reads of a pipe. Even larger 4-kilobyte reads, which spend less time crossing abstraction boundaries, showed almost 4-fold better performance, illustrating the practical utility of code specialization for these call paths [MP89]. Process context switch times were also very fast due to the executable data structure implementation of the run queue: between

Kind of Reference	User Thread		ByteQueue	ByteQueue	Device Driver	Hardware
callentry	**write**	$\Longrightarrow$	`Q_put`	`Q_get`	$\Longleftarrow$	*send-complete* interrupt
callback	**suspend**	$\Longleftarrow$	`Q_full`	`Q_empty`	$\Longrightarrow$	turn off *send-complete*
callback	**resume**	$\Longleftarrow$	`Q_full-1`	`Q_empty-1`	$\Longrightarrow$	turn on *send-complete*

Figure 2: Quaject composition for blocking `putchar`, from [PM91].

7 and 50 microseconds depending on the amount of state being switched.[6]

3.1.1 Multiprocessor issues

The original Synthesis kernel ran on a uniprocessor and used traditional semaphores for synchronization, although lock-free queues were later adopted in some places [MP89]. When Synthesis was ported to the dual-processor Sony NeWS [MP92], lock-free synchronization was used throughout the kernel to protect every shared data structure.

Massalin and Pu's primary motivation for adopting lock-free synchronization for their multiprocessor kernel was performance. In a uniprocessor kernel, disabling interrupts is the cheapest means to suppress asynchronous execution, and hence to obtain synchronization among processes and signal/interrupt handlers. As long as interrupts are not disabled for too long, the adverse effects on the system are small. However, this technique cannot be used to synchronize multiprocessor systems.

Locking synchronization methods can be classified as spin-locks or semaphores, although hybrids also exist. Spin-locks run a processor in a tight loop while waiting to enter a critical region. This can be almost as cheap as disabling interrupts when there is no contention, but wastes processor time which could be spent doing useful work (as long as the processor has more than one active task). Semaphores maintain a waiting queue of blocked processes, which allows the processor to move on to other useful work after adding to the queue. However, Massalin and Pu judge the overhead of queue maintenance and semaphore operations to be prohibitive.

All locking synchronization methods potentially decrease the available parallelism in the system, by restricting the serialization of tasks through a critical region. Priority inversion, as discussed in Section 2.1, is also a concern for the Synthesis kernel, which includes real-time scheduling features for media tasks. Finally, locking synchronization is seen to require tedious programmer bookkeeping of acquire and release orders in order to avoid deadlock.

The Synthesis kernel uses optimistic lock-free synchronization to achieve efficiency comparable to spin-locks without the above-mentioned problems. The designers chose not to use more-expensive wait-free constructs because they felt the probability of starvation in an OS kernel is low. The overhead of non-blocking synchronization is slight; after replacing the executable run queue implementation of uniprocessor Synthesis V.0 with an optimistic lock-free queue in multiprocessor Synthesis V.1, context-switch times rose only slightly, to 12–56 microseconds from 7–50 [MP92].

[6]These numbers come from [MP92], which corrects erroneous times cited in earlier work. An undetected assembler bug had caused most of the floating-point state to be omitted from the context save and restore operations performed in the earlier benchmarks.

3.1.2 Quaject organization

The Synthesis V.1 kernel is organized in an object-oriented manner around *quajects* [PM91]. As the entire kernel is written in macro-assembler there is no language support for quajects, although there are macros to support quaject definition [MP92].

Quajects provide strong encapsulation: except for their interfaces no internal state is revealed. This allows substitution of specialized implementations of a quaject. Although message-passing interfaces were in vogue in the object-oriented community, quajects use an efficient procedural interface with a dispatch table similar to C++ or Java. The synthesis engine uses Collapsing Layers to make interface dispatch even more efficient at runtime, when targets are known. There is no notion of inheritance among quajects, although they may provide and share default implementations of their callback interfaces.

Because the quaject implementation was unconstrained by any higher-level language, the interface employs a novel calling convention which allows both asynchronous and synchronous exception return. For example, the `Q_put` *callentry* interface on a queue quaject returns reporting success if there was space in the queue for the write. If the queue is full, a *callback* named `Q_full` is invoked instead of returning. The caller-supplied callback can choose to suspend the thread, creating a blocking synchronous interface, or to return with an error code (as if from `Q_put`) creating a non-blocking interface. A separate `Q_full-1` callback is invoked when the queue later drains. This callback either resumes the suspended thread and returns from the `Q_put`, or else signals the asynchronous producer.

The quaject structure interacts well with synchronization and code synthesis. Quaject encapsulation allows efficient substitution of specialized implementations of quajects. This allows the presence and type of synchronization in a quaject to exactly match its use. Quajects can be small and modular because code synchronization guarantees efficient composition, removing procedure call overhead at abstraction boundaries at runtime. All references from user code dispatch via the *kernel quaject table*, which because quajects are encapsulated allows efficient protection.

Quajects in Synthesis include threads, memory segments, symbol tables, and *data channels*, which abstract I/O, pipes, and filters. Figure 2 illustrates quaject composition for a blocking `putchar` to an I/O device.

```
CAS(compare, update, mem_addr)
{
    if (*mem_addr == compare) {
        *mem_addr = update;
        return SUCCEED;
    } else
        return FAIL;
}

DCAS(compare1, compare2, update1, update2,
     mem_addr1, mem_addr2)
{
    if (*mem_addr1 == compare1 &&
        *mem_addr2 == compare2) {
        *mem_addr1 = update1;
        *mem_addr2 = update2;
        return SUCCEED;
    } else
        return FAIL;
}
```

Figure 3: Defintion of Compare-And-Swap (CAS) and Double-Compare-And-Swap (DCAS) from [MP91, Appendix A]. Note that each operation is atomic.

```
Push(elem)
{
retry:
    old_SP = SP;
    new_SP = old_SP - 1;
    old_val = *new_SP;
    if(DCAS(old_SP, old_val, new_SP, elem,
            &SP, new_SP) == FAIL)
        goto retry;
}

Pop()
{
retry:
    old_SP = SP;
    new_SP = old_SP + 1;
x:  elem = *old_SP;
    if(CAS(old_SP, new_SP, &SP) == FAIL)
        goto retry;
    return elem;
}
```

Figure 4: Synthesis implementation of LIFO stack, from [MP91, Figure 1].

3.1.3 Lock-free quajects

Almost all Synthesis synchronization is done with one of three quajects: a LIFO stack, FIFO queue, or general linked list. Synthesis targets 680X0 platforms, and the lock-free implementations use the architecture's Compare And Swap (CAS) and Double Compare And Swap (DCAS) instructions. CAS atomically compares the contents of a memory location against a value, if they match stores a new value into the location. DCAS atomically compares the contents of two locations against two values, and if both values match stores two new values into the locations. Figure 3 gives pseudo-code for these atomic instructions.

The LIFO stack implementation in Synthesis is presented in Figure 4. When pushing an item on the stack we atomically increase the stack pointer and store the new item at the top of the stack, *if and only if* the stack pointer has not changed since we read it last. This guarantees that exactly one `Push` succeeds if more than one are simultaneously attempted. Likewise, a `Pop` which succeeds after the first read of the stack pointer will prevent the `Push` from writing to the wrong location.

However, the `Pop` implementation presented by Massalin and Pu contains an error. The intent is that a successful CAS on the stack pointer guarantees that the element we previously read from the top of the stack is valid. However, consider the case where processor one reads `elem` from the top of the stack at label `x` inside `Pop`. Processor two then executes a `Pop` followed by a `Push` of a different value before the first processor makes any more progress. The first processor will then succeed with its `Pop`, even though the value it holds from the top of the stack is stale. This is what is known as an *ABA problem*: we can't tell the second value of the stack pointer A from the first value A, even though a different value B intervened.

The solution is to use DCAS in order to ensure that that the top of the stack has not changed when we write the new stack pointer. This solution is subject to the ABA problem as well, only now the only possible confusion is whether a pop followed by a push *of the same value that was popped* intervened. This doesn't matter, because the result of our pop operation is the same in either case.

It is likely that this error in `Pop` was never observed in Synthesis because its hardware platform had only two processors of equal speeds. Thus it becomes very unlikely that one processor can squeeze in both a `Pop` and a `Push` during the critical region in `Pop`. If more processors were involved, the bug scenario would become increasingly likely.

The FIFO queue implementation in Synthesis has four variations, depending on the number of producers and consumers for the queue. Figure 5 shows the single-producer single-consumer (SP-SC) version of the algorithm. It uses only atomic loads and stores, and like Lamport's original non-blocking synchronization algorithm [Lam77], it relies on dependency ordering between the loads and stores for correctness. Because the producer updates `Q_head` last while the consumer checks `Q_head` first (before its read of `data`), the consumer will never see invalid data at `Q_head-1`. Similarly, because the consumer updates `Q_tail` last while the producer checks `Q_tail` first (before its write to the queue), the producer will never overwrite data being read by `Q_get`.

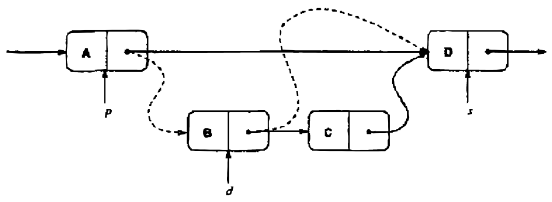
The multiple-producer single-consumer (MP-SC) version of the queue is the same as the SP-SC implementation, except that the last two lines of `Q_put` are implemented atomically by a DCAS instruction, guaranteeing that no other producer has advanced `Q_head` before our store.[7] The other two versions of the FIFO queue have not been published, although [MP89] contains an additional multiple-item variation of the MP-SC queue.

Synthesis also contains a lock-free linked list implemen-

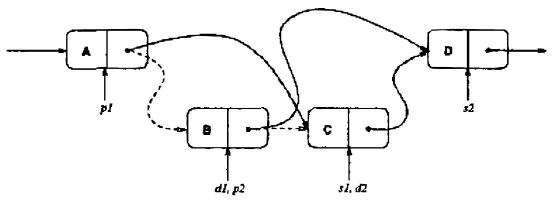
[7]It is unclear from [MP89] whether Synthesis uses DCAS or CAS to implement the MP-SC FIFO queue; the text states, "a single compare-and-swap instruction at the end." If CAS is used instead of DCAS, then this implementation is also faulty, as the store to `Q_buf[h]` just before a failing CAS may overwrite data written by a successful concurrent `Q_put`.

```
next(x) {
    return (x+1) % Q_size;
}

Q_get() {
    t = Q_tail;
    if (t == Q_head)
        wait;
    data = Q_buf[t];
    Q_tail = next(t);
    return data;
}

Q_put(data) {
    h = Q_head;
    if (next(h) == Q_tail)
        wait;
    Q_buf[h] = data;
    Q_head = next(h);
}
```

Figure 5: Synthesis implementation of single-producer single-consumer FIFO queue, from [MP89, Figure 1].



Deletion of B concurrent with insertion of C.



Concurrent deletion of B and C; second is undone.

Figure 6: Problematic cases among simultaneous updates to a linked list, from [Val95].

tation. Deletion from arbitrary locations in the list poses special problems for non-blocking implementations; Figure 6 illustrates two hard cases. Massalin and Pu solve the deletion problem by setting a flag to DELETED on nodes without actually removing them from the list. Nodes marked DELETED are removed later, when it is “safe”. There is a traversal of the Synthesis run queue in which one thread at a time visits each node; deletion occurs at those points of isolation.[8]

Massalin and Pu compared the execution times of their optimistic lock-free quaject implementations to implementations without any synchronization at all. In the absence of retries, the optimistic implementations were between 1.5 and 4.4 times slower than the unsynchronized implementations [MP91]. This is not bad, considering the speed of the unsynchronized implementations; in absolute terms every lock-free operation completed in less than 3 microseconds without retries, or less than 6 microseconds in case of a single retry.

3.1.4 Beyond Synthesis V.1

In evaluating Synthesis [PW93], Pu and Massalin conclude that strong object encapsulation within the kernel is an important enabler for dynamic code generation. Procedural interfaces are a vital part of encapsulation: allowing clients to see and mutate kernel data structures eliminates the ability to specialize and change the underlying data representation. The “objectification” of the Synthesis kernel allows code specialization and increases the scope for performance optimisation.

However, Synthesis does not provide any language support to aid this “objectification”. The authors point to lack of code generation support in high-level languages as the reason they have stuck with macroassembler. The lack of language support is not uncommon. In fact, *many* modern operating systems are both “object-oriented” and lacking in language support for their object system. The Linux kernel, for example, employs an object-oriented organization of device and file I/O, including virtual dispatch, but the kernel’s implementation language is C.

From the start of the Synthesis project a language with quaject and dynamic code generation support was planned. The language was to be called “Lambda-C” [PMI88]; in [PW93] Pu and Walpole propose that the high-level language support “modularity, strong typing, and well-defined semantics” for code specialization and generation, and that it allow dynamic composition of quajects via code generation. Burden would be shifted off the programmer by including a mechanism to track expected invariants among specialized implementations, and to generate guard statements to detect when invariants no longer hold and generated code must be invalidated. For example, Synthesis V.1 takes great care to ensure that relevant parts of the instruction cache are flushed after code generation, but this attention is expressed as a great deal of hand-optimized template code. The invariants to be satisfied remain completely implicit, which makes it very likely that they will be violated or optimizations will be lost as the code evolves or is ported.

Code generation in Synthesis’ successor was to integrate a more general model of specialization, *incremental partial evaluation.* As details about the dynamic state of the program became known, implementations would become more and more specialized. Ultimately there would exist a hierarchical organization of increasingly specialized implementations, all of which conform to a quaject’s desired abstract type.

Finally, Pu and Walpole note the non-portability of their non-blocking synchronization implementations based on CAS and DCAS, and propose to abstract away the architecture dependence by building their next system on top of transactional memory [HM93].

[8]The Cache Kernel contains a more general lock-free linked list implemenation using DCAS, and Valois more recently published an equally general non-blocking implementation using only compare-and-swap [Val95].

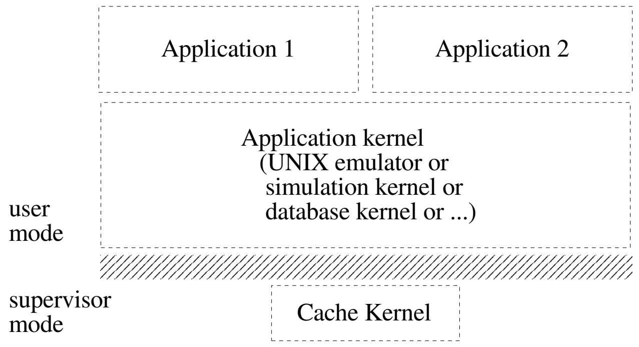

Application 1
Application 2
Application kernel (UNIX emulator or simulation kernel or database kernel or ...)
user mode
supervisor mode
Cache Kernel

Figure 7: Overall System Architecture for the V++ Cache Kernel, from [CD94].

3.2 The Cache Kernel

The V++ Cache Kernel [CD94, GC96] builds a minimal micro-kernel from only three basic object types: address spaces, threads, and application kernels. Signals are the only kernel-supported form of notification. The result is a highly modular object-oriented operating system whose structure and decoupling depends crucially on the use of non-blocking synchronization.

The overall architecture of the V++ operating system, based on the Cache Kernel, is shown in Figure 7. Applications execute on top of an application kernel, either in the same address space as the application kernel or in a separate address space. The Cache Kernel caches active operating system objects: address spaces, threads, and application kernels. The interaction of active and inactive objects—such as scheduling in the case of threads, or virtual memory support for address spaces—is the responsibility of the application kernel, which receives signals from the Cache Kernel and loads and unloads system objects.

3.2.1 Basic object types

Address space objects store a collection of per-page virtual-to-physical address memory mappings. Each mapping describes a mapped page and contains a virtual address, corresponding physical address, and access flags.

Address spaces are created with no mappings. A "load mapping" request to the Cache Kernel adds a mapping with the given access flags associating a particular virtual and physical address. The Cache Kernel verifies that the physical address specified is among those reserved for the application kernel making the request. Page faults bounce from the Cache Kernel to the application kernel's fault handler, which may initialize a physical page, load a mapping associating it with the referenced virtual page, and then resume. This exception handling is similar for protection faults, privilege violations, and consistency faults involving distributed shared memory: the application kernel is always responsible for fault resolution, leaving the Cache Kernel free of the mechanisms of memory allocation, deallocation, and paging.

The Cache Kernel may unload mappings and address spaces to make room for more active objects. The Cache Kernel's unload mechanism invokes the application kernel with the object data for writeback (either into application kernel state or onto stable storage) before the object is destroyed. Address spaces can be associated with threads; if so, unloading an address space also unloads the affiliated threads.

Thread objects loaded in the Cache Kernel are always runnable. A thread is descheduled by unloading it from the Cache Kernel. Each thread object is associated with an address space object whose creation must precede the thread's. Threads can execute in the application kernel's address space or in their own. When executing in a private address space, a thread makes "system calls" to its application kernel using the processor trap instruction. This traps to the Cache Kernel and the trap is forwarded to the application kernel. Threads executing in the application kernel's address space can make direct procedure calls to the application kernel; processor traps in this case invoke Cache Kernel interfaces.

The Cache Kernel can contain a number of application kernel objects, corresponding to active application kernels. Application kernels can include UNIX emulators as well as specialized kernels for specific applications. Each application kernel object designates the application kernel's address space, its trap and exception handlers, and its resources, including the physical memory pages allocated to it. Other kernel resources include a CPU time slice and a limited number of "locked" Cache Kernel objects which it can protect from unloading. Application kernel objects can themselves be loaded and unloaded, which allows swapping out large jobs running on their own application kernels.

3.2.2 Interprocess communication and synchronization

All kernel notification in the Cache Kernel is via signals. Interprocess and device communication piggy-backs on the signal mechanism using *memory-based messaging.* The sender and receiver both share a mapped memory region, and the sender sends *address-valued signals* to the receiver pointing at messages it has written in the region. The ParaDiGM machine on which the Cache Kernel runs provides hardware support for *automatic signal-on-write* to dispatch and deliver an address-valued-signal to the appropriate processor upon a write to a certain location.

Given these features, asynchronous signals are ubiquitous in the Cache Kernel. The developers adopted non-blocking synchronization so that their system and libraries would not have to continually disable and enable signals and suffer restrictions on code in signal handlers. The non-blocking implementation allows asynchronous signal handlers to update data structures without the risk of deadlock with synchronous code which might itself have already begun to edit the data structure. In particular, non-blocking synchronization protects page fault and other exception handlers from deadlock when they are triggered from within code accessing protected kernel data structures.

Non-blocking synchronization also protects the application kernel from terminated threads, which might have been executing inside an application kernel interace when terminated. It thus allows user threads to execute on behalf of the kernel without the need to protect them from termination or other faults.

Decoupling scheduling and synchronization allows other threads to make progress even when we are in a lengthy handler, and prevents priority inversion when low-priority threads access shared structures.

```
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 8: Lock-free deletion from a linked list, from [GC96].

3.2.3 Lock-free data structure implementations

The Cache Kernel includes lock-free implementations of collections, queues, and search structures. The Cache Kernel runs on 680X0 processors, and so uses the same CAS and DCAS primitives as does the Synthesis kernel.

Cache Kernel data structure implementations rely heavily on version numbers to track changes. DCAS is usually used with the version number as one argument, guaranteeing on success that the data structure has not changed while we were looking at it. An example of the technique is presented in Figure 8, the Cache Kernel algorithm to remove a node from a linked list. The pointer swap occurs atomically with a test-and-increment of the version number; if a change to the list structure occurs while we are locating the nodes before and after the target node, the version number will be incremented and our operation will fail and retry. The version number effectively protects all data in the list. To increase concurrency, some data structures are split (by hashing into buckets, for example) into multiple smaller data structures, each with their own version number, to reduce false conflicts during concurrent updates.

Version numbers with DCAS allow only one location to be updated atomically, although that update can read-depend on the entire contents of the data structure. To make multiple writes atomic, a variant of Herlihy's [Her93] universal scheme is used, where an element is atomically replaced by an updated copy. DCAS with the version number is also used here, to prevent *lost updates* when the original is modified after the copy and before its replacement.

To avoid copying overhead, the Cache Kernel occassionally removes an element from a list in order to modify it, re-adding it when the modification is complete. This prevents concurrent modification and guarantees exclusive access to the element (because only one thread's deletion of the element can succeed), but creates windows during which some resources may be missing from lists describing them. For example, the lists of threads handling a given signal is incomplete, because threads may be removed from the list while the handler is modified. The Cache Kernel accepts this incompleteness in its the low levels, and works around so-called "best-effort signal delivery" with higher-level time-out and retry mechanisms.

Temporarily removing elements for mutation must be applied with care, as it is easy to create mutual exclusion between threads by naïvely retrying searches for elements "temporarily missing" from their resource lists. The search loop might forever prevent the completion of the mutation responsible for removing the element.

Consistency properties which require a resource to be on multiple lists simultaneously also pose a problem for the techniques used in the Cache Kernel, as do data structures significantly more complex than lists or queues. The designers took care such that "best-effort" consistency and appropriate recovery mechanisms were sufficient. They mention the use of a special "synchronization server" which performs operations serially from a queue as a more general solution to the problem. Massalin and Pu [MP91] mention that this technique was used during the development of the Synthesis kernel, although kernel maturation eliminated the need for it. Similarly, the completed Cache Kernel appears to use a separate synchronization process only for I/O.

3.2.4 Type-Stable Memory Management

Greenwald and Cheriton mention a number of enabling techniques which allowed the parts of the Cache Kernel to work well with each other. First among them is Type-Stable Memory Management (TSM). Memory reclamation is always an issue with non-blocking data structures because it is hard to guarantee that there isn't a stalled thread somewhere sitting in the piece of memory you want to reclaim.

The Cache Kernel solution is to ensure that memory is *type-stable*; that is, if there was once a thread descriptor, say, sitting in a piece of memory, than that memory will always represent a valid thread descriptor. Code which stalls looking at that piece of memory may see a different valid thread descriptor but will never see an invalid descriptor, and will never be exposed to wild pointers. If you can bound the maximum amount of time a thread may get "stuck", then you can also bound how long you are required to keep a piece of memory type-stable.

Type-stable storage greatly simplifies the amount of defensive coding required for implementation of non-blocking synchronization, because it guarantees that your data will always "make sense". Greenwald and Cheriton also argue that there are performance benefits to type-specific allocators, and that the locality of allocated type-specific storage aids debugging and auditing. Type-stable allocators also take care to align objects with the cache, which reduces physical memory contention.

4 Object-orientation

Both Synthesis and the Cache Kernel benefit from object-oriented design. The specialization techniques used in Synthesis would not be possible if not for the tight encapsulation provided by its quajects. The Type-Stable Memory Manager and the Object-Oriented RPC system built on top of the Cache Kernel [GC96, ZC96] leveraged objects to allow efficiency through specialization as well. The authors of Synthetix, offspring of Synthesis, argued that, "truely significant performance gains come from choosing the right implementation. . . The encapsulation provided by object-oriented systems should allow us to replug implementations, allowing them to better match their environment" [BW95].

By encapsulating synchronization in objects—lists, stacks, queues—the operating system designers were able

```
const myDirectory == object oneEntryDirectory
  export Store, Lookup
  monitor
    var name : String
    var AnObject : Any

    operation Store [ n : String, o : Any ]
      name ← n
      AnObject ← o
    end Store
    function Lookup [ n : String ] → [ o : Any ]
      if n = name
        then o ← AnObject
        else o ← nil
      end if
    end Lookup

    initially
      name ← nil
      AnObject ← nil
    end initially

  end monitor
end oneEntryDirectory
```

Figure 9: A directory object in Emerald, from [BHJL86], illustrating the use of monitor synchronization.

```
class Account {

  int balance = 0;

  atomic int deposit(int amt) {
    int t = this.balance;
    t = t + amt;
    this.balance = t;
    return t;
  }

  atomic int readBalance() {
    return this.balance;
  }

  atomic int withdraw(int amt) {
    int t = this.balance;
    t = t - amt;
    this.balance = t;
    return t;
  }

}
```

Figure 10: A simple bank account object, adapted from [FQ03], illustrating the use of the `atomic` modifier.

to separate out the "non-blocking synchronization" aspect from the rest of their design. Blocking synchronization is not isolated so easily. Synthesis shows that another benefit of this separated structure is that you can mix-and-match implementations of your data types to get exactly the amount of synchronization you require.

However, contemporary research on object-oriented operating systems has concentrated on orthogonal issues: in particular, in the relationship between language safety properties and operating system protection mechanisms [HCC$^+$98, SC02, FFKF99]. The two currents of specialization and protection meet in the design of "extensible operating systems," where the user is expected to collaborate in providing specialized implementations of kernel services. However, the system would still like to impose safety guarantees on the user's code to limit effects on the rest of the tasks and users in the system. "Extensible kernels based on software protection logically converge to single address space operating systems, like most Java-based operating systems, where all protection mechanisms are in software" [DPM02].

5 A proposal for a non-blocking Java OS

Language-based operating systems promise to allow even further reduction in micro-kernel design by removing even address spaces from the kernel. Further, by providing correct and universal synchronization in the language, we hope to eliminate the difficulty of writing specific correct lock-free data structures, illustrated by Massalin and Pu's errors in Section 3.1.3.

To that end, we propose some slight modifications to the Java language to better support its use for writing operating systems. We concentrate on synchronization in the language, eliding as orthogonal issues related to direct access to hardware. We expect that slight extensions similar to those in Lisaac [SC02] and JEPES [SBCK03] will be sufficient to support interrupt handlers and memory-mapped I/O. We then present an efficient implementation technique for the synchronization primitives in the language, showing first how to synchronize single encapsulated objects, then operations on multiple objects, and finally present the lock-free functional array implementation which allows efficient large object synchronization.

Our implementation will use the same DCAS primitive used in the Cache Kernel and in Synthesis. This primitive can be emulated using architecture extensions to the popular LL/SC operations, or in software with OS support using Bershad's technique [GC96, Ber93].

5.1 Synchronization in the language

The Emerald system [BHJL86, JS91] introduced *monitored objects* for synchronization. Emerald code to implement a simple directory object is shown in Figure 9. Each object is associated with Hoare-style monitor, which provides mutual exclusion and process signalling. Each Emerald object is divided into a monitored part and a non-monitored part. Variables declared in the monitored part are shared, and access to them from methods in the non-monitored part is prohibited—although non-monitored methods may call monitored methods to effect the access. Methods in the monitored part acquire the monitor lock associated with the receiver object before entry and release it on exit, providing for mutual exclusion and safe update of the shared variables. Monitored objects naturally integrate synchronization into the object model.

Unlike Emerald monitored objects, where methods can only acquire the monitor of their receiver and where restricted access to shared variables is enforced by the compiler, Java implements a loose variant where any monitor may be explicitly acquired and no shared variable protection exists. As a default, however, Java methods declared with the `synchronized` keyword behave like Emerald monitored methods, ensuring that the monitor lock of their receiver is held during execution.

Java's synchronization primitives arguably allow for more efficient concurrent code than Emerald's—for example, Java objects can use multiple locks to protect disjoint sets of fields, and coarse-grain locks can be used which protect multiple objects—but Java is also more prone to programmer error. However, even Emerald's restrictive monitored objects are not sufficient to prevent data races. As a simple example, imagine that an object provided two monitored methods `read` and `write` which accessed a shared variable. Non-monitored code can call `read`, increment the value returned, and then call `write`, creating a classic race condition scenario. The atomicity of the parts is not sufficient to guarantee atomicity of the whole [FQ03].

This suggests that a better model for synchronization in object-oriented systems is *atomicity*. Figure 10 shows Java extended with an `atomic` keyword to implement an object representing a bank account. Rather than explicitly synchronizing on locks, we simply require that the methods marked `atomic` execute atomically with respect to other threads in the system; that is, that every execution of the program computes the same result as some execution where all atomic methods were run *in isolation* at a certain point in time between their invocation and return. This point is called the *linearization point*. Note that atomic methods invoked directly or indirectly from an atomic method are subsumed by it: if the outermost method appears atomic, then by definition all inner method invocations will also appear atomic. Flanagan and Qadeer provide a more formal semantics in [FQ03]. Atomic methods can be analyzed using sequential reasoning techniques, which significantly simplifies reasoning about program correctness.

Atomic methods can be implemented using locks. A simple if deadlock-prone implementation would simply acquire a single global lock during the execution of every atomic method. Flanagan and Qadeer [FQ03] present a more sophisticated technique which proves that a given implementation using standard Java monitors correctly guarantees method atomicity. We will use non-blocking synchronization to implement atomic methods.

5.2 A simple implementation of functional arrays

Our atomic method implementation will use *functional arrays* as a building block. Functional arrays are *persistent*; that is, after an element is updated both the new and the old contents of the array are available for use. Since arrays are simply maps from integers (indexes) to values; any functional map datatype (for example, a functional balanced tree) can be used to implement functional arrays.

However, the distinguishing characteristic of an imperative array is its theoretical complexity: $O(1)$ access or update of any element. Implementing functional arrays with a functional balanced tree yields $O(\lg n)$ worst-case access or update.

For concreteness, functional arrays have the following three operations defined:

- FA-CREATE(n): Return an array of size n. The contents of the array are initialized to zero.
- FA-UPDATE(A_j, i, v): Return an array $A_{j'}$ which is functionally identical to array A_j except that $A_{j'}(i) = v$. Array A_j is not destroyed and can be accessed further.
- FA-READ(A_j, i): Return $A_j(i)$.

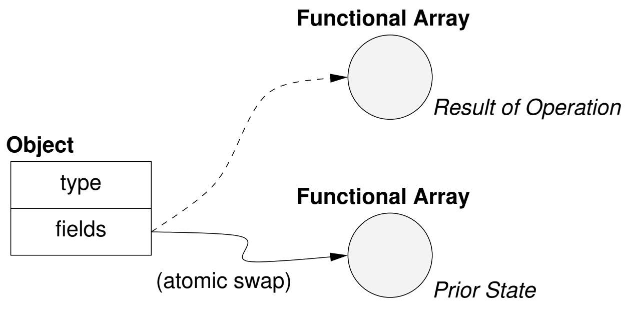


Figure 11: Implementing non-blocking single-object concurrent operations with functional arrays.

We allow any of these operations to *fail*. Failed operations can be safely retried, as all operations are idempotent by definition.

For the moment, consider the following naïve implementation:

- FA-CREATE(n): Return an ordinary imperative array of size n.
- FA-UPDATE(A_j, i, v): Create a new imperative array $A_{j'}$ and copy the contents of A_j to $A_{j'}$. Return $A_{j'}$.
- FA-READ(A_j, i): Return $A_j[i]$.

This implementation has $O(1)$ read and $O(n)$ update, so it matches the performance of imperative arrays only when $n = O(1)$. We will therefore call these *small object functional arrays*. Operations in this implementation never fail. Every operation is non-blocking and no synchronization is necessary, since the imperative arrays are never mutated after they are created.

5.3 A single-object protocol

Given a non-blocking implementation of functional arrays, we can construct an implementation of `atomic` for single objects. In this implementation, fields of at most one object may be referenced during the execution of the atomic method.

We will consider the following two operations on objects:

- READ(o, f): Read field f of o. We will assume that there is a constant mapping function which given a field name returns an integer index. We will write the result of mapping f as f`.index`. For simplicity, and without loss of generality, we will assume all fields are of equal size.
- WRITE(o, f, v): Write value v to field f of o.

All other operations on Java objects, such as method dispatch and type interrogation, can be performed using the immutable `type` field in the object. Because the `type` field is never changed after object creation, non-blocking implementations of operations on the `type` field are trivial.

As Figure 11 shows, our single-object implementation of `atomic` represents objects as a pair, combining `type` and a reference to a functional array. When not inside an atomic method, object reads and writes are implemented using the corresponding functional array operation, with the array reference in the object being updated appropriately:

- READ(o, f): Return FA-READ(o.`fields`, f.`index`).
- WRITE(o, f, v): Replace o.`fields` with the result of FA-UPDATE(o.`fields`, f.`index`, v).

The interesting cases are reads and writes inside an atomic method. At entry to our atomic method which will access (only) object o, we store o.`fields` in a local variable u. We create another local variable u' which we initialize to u. Then our read and write operations are implemented as:

- READATOMIC(o, f): Return FA-READ(u', f.`index`).
- WRITEATOMIC(o, f, v): Update variable u' to the result of FA-UPDATE(u', f.`index`, v).

At the end of the atomic method, we use Compare-And-Swap to atomically set o.`fields` to u' iff it contained u. If the CAS fails, we back-off and retry.

With our naïve "small object" functional arrays, this implementation is exactly the "small object protocol" of Herlihy [Her93]. Herlihy's protocol is rightly criticized for an excessive amount of copying. We will address this with a better implementation of functional arrays in Section 5.5. However, the restriction that only one object may be referenced within an atomic method is overly limiting.

5.4 Extension to multiple objects

We now extend the implementation to allow the fields of any number of objects to be accessed during the atomic method. Figure 12 shows our new object representation. Objects consist of two slots, and the first represents the immutable `type`, as before. The second field, `versions`, points to a linked list of `Version` structures. The `Version` structures contain a pointer `fields` to a functional array, and a pointer `owner` to an *operation identifier*. The operation identifier contains a single field, `status`, which can be set to one of three values: *COMPLETE*, *IN-PROGRESS*, or *DISCARDED*. When the operation identifier is created, the status field is initialized to *IN-PROGRESS*, and it will be updated exactly once thereafter, to either *COMPLETE* or *DISCARDED*. A *COMPLETE* operation identifier never later becomes *IN-PROGRESS* or *DISCARDED*, and a *DISCARDED* operation identifier never becomes *COMPLETE* or *IN-PROGRESS*.

We create an operation identifier when we begin or restart an atomic method and place it in a local variable *oid*. At the end of the atomic method, we use CAS to set *oid*.`status` to *COMPLETE* iff it was *IN-PROGRESS*. If the CAS is successful, the atomic method has also executed successfully; otherwise *oid*.`status` = *DISCARDED* and we must back-off and retry the atomic method. All `Version` structures created while in the atomic method will reference *oid* in in their `owner` field.

Semantically, the current field values for the object will be given by the first version in the versions list whose operation identifier is *COMPLETE*. This allows us to link *IN-PROGRESS* versions in at the head of multiple objects' versions lists and atomically change the values of all these objects by setting the one common operation identifier to *COMPLETE*. We only allow one *IN-PROGRESS* version

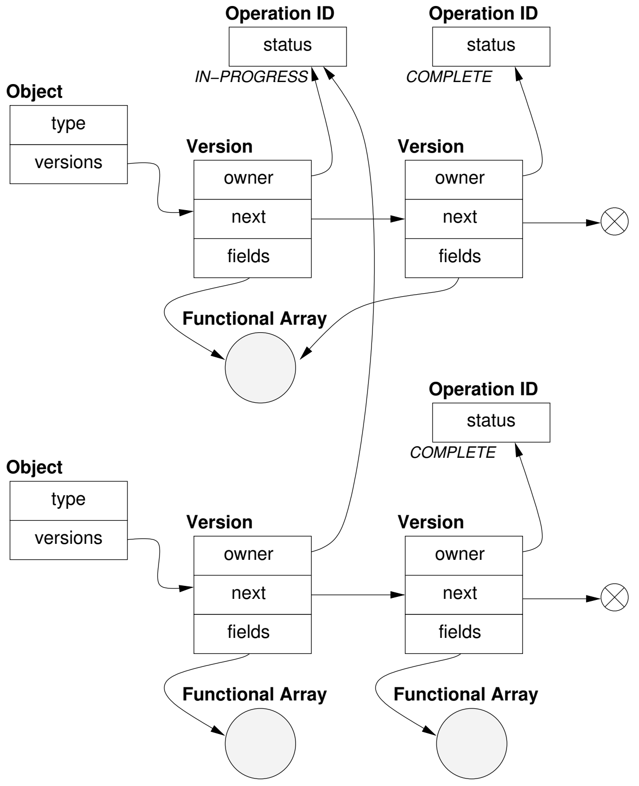


Figure 12: Data structures to support non-blocking multi-object concurrent operations. Objects point to a linked list of versions, which reference operation identifiers. Versions created within the same execution of an atomic method share the same operation identifier. Version structure also contain pointers to functional arrays, which record the values for the fields of the object. If no modifications have been made to the object, multiple versions in the list may share the same functional array.

on the versions list, and it must be at the head, so Therefore, before we can link a new version at the head, we must ensure that every other version on the list is *DISCARDED* or *COMPLETE*.

Since we will never look past the first *COMPLETE* version in the versions list, we can free all versions past that point. In our presentation of the algorithm, we do this by explicitly setting the `next` field of every *COMPLETE* version we see to **null**; this allows the versions past that point to be garbage collected. An optimization would be to have the garbage collector do the list trimming for us when it does a collection.

We don't want to inadvertently chase the null `next` pointer of a *COMPLETE* version, so we always load the `next` field of a version *before* we load `owner.status`. Since the writes occur in the reverse order (*COMPLETE* to `owner.status`, then **null** to `next`) we have ensured that our `next` pointer is valid whenever the status is not *COMPLETE*.

We begin an atomic method with ATOMICENTRY and attempt to complete an atomic method with ATOMICEXIT. They are defined as follows:

- ATOMICENTRY: create a new operation identifier, with

```
READ$(o, f)$:
begin
retry:
  $u \leftarrow o$.versions
  $u' \leftarrow u$.next
  $s \leftarrow u$.owner.status
  if ($s$ = DISCARDED)                      [Delete DISCARDED?]
    CAS($u, u'$, &($o$.versions))
    goto retry
  else if ($s$ = COMPLETE)
    $a \leftarrow u$.fields                        [$u$ is COMPLETE]
    $u$.next $\leftarrow$ null                     [Trim version list]
  else
    $a \leftarrow u'$.fields                       [$u'$ is COMPLETE]
  return FA-READ($a, f$.index)                  [Do the read]
end

READATOMIC$(o, f)$:
begin
  $u \leftarrow o$.versions
  if (oid = $u$.owner)                      [My OID should be first]
    return FA-READ($u$.fields, $f$.index)   [Do the read]
  else                                          [Make me first!]
    $u' \leftarrow u$.next
    $s \leftarrow u$.owner.status
    if ($s$ = DISCARDED)                    [Delete DISCARDED?]
      CAS($u, u'$, &($o$.versions))
    else if (oid.status = DISCARDED)        [Am I alive?]
      fail
    else if ($s$ = IN-PROGRESS) [Abort IN-PROGRESS?]
      CAS($s$, DISCARDED, &($u$.owner.status))
    else                                    [Link new version in:]
      $u$.next $\leftarrow$ null                   [Trim version list]
      $u' \leftarrow$ new Version(oid, $u$, null) [Create new version]
      if (CAS($u, u'$, &($o$.versions)) $\neq$ FAIL)
        $u'$.fields $\leftarrow u$.fields           [Copy old fields]
    goto retry
end
```

Figure 13: READ and READATOMIC implementations for the multi-object protocol.

```
WRITE$(o, f, v)$:
begin
retry:
  $u \leftarrow o$.versions
  $u' \leftarrow u$.next
  $s \leftarrow u$.owner.status
  if ($s$ = DISCARDED)                          [Delete DISCARDED?]
    CAS($u, u'$, &($o$.versions))
  else if ($s$ = IN-PROGRESS)                   [Abort IN-PROGRESS?]
    CAS($s$, DISCARDED, &($u$.owner.status))
  else                                              [$u$ is COMPLETE]
    $u$.next $\leftarrow$ null                         [Trim version list]
    $a \leftarrow u$.fields
    $a' \leftarrow$ FA-UPDATE($a, f$.index, $v$)
    if (CAS($a, a'$, &($u$.fields)) $\neq$ FAIL)       [Do the write]
      return                                        [Success!]
  goto retry
end

WRITEATOMIC$(o, f, v)$:
begin
  $u \leftarrow o$.versions
  if (oid = $u$.owner)                          [My OID should be first]
    $u$.fields $\leftarrow$ FA-UPDATE($u$.fields, $f$.index, $v$)[Do write]
  else                                              [Make me first!]
    $u' \leftarrow u$.next
    $s \leftarrow u$.owner.status
    if ($s$ = DISCARDED)                        [Delete DISCARDED?]
      CAS($u, u'$, &($o$.versions))
    else if (oid.status = DISCARDED)            [Am I alive?]
      fail
    else if ($s$ = IN-PROGRESS)                 [Abort IN-PROGRESS?]
      CAS($s$, DISCARDED, &($u$.owner.status))
    else                                        [Link new version in:]
      $u$.next $\leftarrow$ null                       [Trim version list]
      $u' \leftarrow$ new Version(oid, $u$, null)      [Create new version]
      if (CAS($u, u'$, &($o$.versions)) $\neq$ FAIL)
        $u'$.fields $\leftarrow u$.fields               [Copy old fields]
    goto retry
end
```

Figure 14: WRITE and WRITEATOMIC implementations for the multi-object protocol.

its status initialized to *IN-PROGRESS*. Assign it to the thread-local variable *oid*.

- AtomicExit: If

  CAS(*IN-PROGRESS*, *COMPLETE*, `&`(*oid*.`status`))

  is successful, the atomic method as a whole has completed successfully, and can be linearized at the location of the CAS. Otherwise, the method has failed. Back-off and retry from AtomicEntry.

Pseudo-code describing Read, Write, AtomicRead, and AtomicWrite is presented in Figures 13 and 14. In the absence of contention, all operations take constant time plus an invocation of FA-Read or FA-Update.

5.5 Lock-free functional arrays

In this section we will present a lock-free implementation of functional arrays with $O(1)$ performance in the absence of contention. This will complete our implementation of non-blocking `atomic` methods for Java.

There have been a number of proposed implementations of functional arrays, starting from the "classical" functional binary tree implementation. O'Neill and Burton [OB97] give a fairly inclusive overview. Functional array implementations fall generally into one of three categories: *tree-based*, *fat-elements*, or *shallow-binding*.

Tree-based implementations typically have a logarithmic term in their complexity. The simplest is the persistent binary tree with $O(\ln n)$ look-up time; Chris Okasaki [Oka95] has implemented a purely-functional random-access list with $O(\ln i)$ expected lookup time, where i is the index of the desired element.

Fat-elements implementations have per-element data structures indexed by a master array. Cohen [Coh84] hangs a list of versions from each element in the master array. O'Neill and Burton [OB97], in a more sophisticated technique, hang a splay tree off each element and achieve $O(1)$ operations for single-threaded use, $O(1)$ amortized cost when accesses to the array are "uniform", and $O(\ln n)$ amortized worst case time.

Shallow binding was introduced by Baker [Bak78] as a method to achieve fast variable lookup in Lisp environments. Baker clarified the relationship to functional arrays in [Bak91]. Shallow binding is also called *version tree arrays*, *trailer arrays*, or *reversible differential lists*. A typical drawback of shallow binding is that reads may take $O(u)$ worst-case time, where u is the number of updates made to the array. Tyng-Ruey Chuang [Chu94] uses randomized cuts to the version tree to limit the cost of a read to $O(n)$ in the worst case. Single-threaded accesses are $O(1)$.

Our use of functional arrays is single-threaded in the common case, when transactions do not abort. Chuang's scheme is attractive because it limits the worst-case cost of an abort, with very little added complexity. In this section we will present a lock-free version of Chuang's randomized algorithm.

In shallow binding, only one version of the functional array (the *root*) keeps its contents in an imperative array (the *cache*). Each of the other versions is represented as a path of *differential nodes*, where each node describes the differences between the current array and the previous array. The difference is represented as a pair $\langle index, value\rangle$, representing the new value to be stored at the specified index. All paths lead to the root. An update to the functional array is simply implemented by adding a differential node pointing to the array it is updating.

The key to constant-time access for single-threaded use is provided by the read operation. A read to the root simply reads the appropriate value from the cache. However, a read to a differential node triggers a series of rotations which swap the direction of differential nodes and result in the current array acquiring the cache and becoming the new root. This sequence of rotations is called *re-rooting*, and is illustrated in Figure 15. Each rotation exchanges the root nodes for a differential node pointing to it, after which the differential node becomes the new root and the root becomes a differential node pointing to the new root. The cost of a read is proportional to its re-rooting length, but after the first read accesses to the same version are $O(1)$ until the array is re-rooted again.

Shallow binding performs badly if read operations ping-pong between two widely separated versions of the array, as we will continually re-root the array from one version to the other. Chuang's contribution is to provide for *cuts* to the chain of differential nodes: once in a while we clone the cache and create a new root instead of performing a rotation. This operation takes $O(n)$ time, so we amortize it over n operations by randomly choosing to perform a cut with probability $1/n$.

Figure 16 shows the data structures used for the functional array implementation, and the series of atomic steps used to implement a rotation. The `Array` class represents a functional array; it consists of a `size` for the array and a pointer to a `Node`. There are two types of nodes: a `CacheNode` stores a value for every index in the array, and a `DiffNode` stores a single change to an array. `Array` objects which point to `CacheNode`s are roots.

In step 1 of the figure, we have a root array A and an array B whose differential node d_B points to A. The functional arrays A and B differ in one element: element x of A is z, while element x of B is y. We are about to rotate B to give it the cache, while linking a differential node to A.

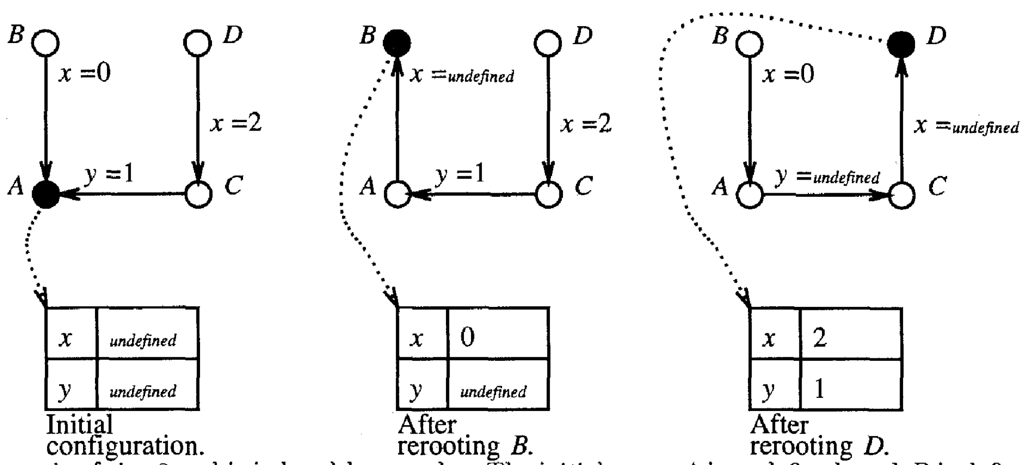
Step 2 shows our first atomic action. We have created a new `DiffNode` d_A and a new `Array` C and linked them between A and its cache. The `DiffNode` d_A contains the value for element x contained in the cache, z, so there is no change in the value of A.

We continue swinging pointers until step 5, when can finally set the element x in the cache to y. We perform this operation with a DCAS operation which checks that C.`node` is still pointing to the cache as we expect. Note that a concurrent rotation would swing C.`node` in its step 1. In general, therefore, the location pointing to the cache serves as a reservation on the cache.

Thus in step 6 we need to again use DCAS to simultaneously swing C.`node` away from the cache as we swing B.`node` to point to the cache.

Figure 17 presents pseudocode for FA-Rotate, FA-Read, and FA-Update. Note that FA-Read also uses the cache pointer as a reservation, double-checking the cache pointer after it finishes its read to ensure that the cache hasn't been stolen from it.

Let us now consider cuts, where FA-Read clones the cache instead of performing a rotation. Cuts also check the cache pointer to protect against concurrent rotations. But what if the cut occurs while a rotation is mutating the cache in step 5? In this case the only array adjacent to the root is B, so the cut must be occurring during an invocation of



NOTE. The array is of size 2 and is indexed by x and y. The initial array A is undefined, and B is defined as an update to A at index x by value 0. Similarly for C and D. The dark node is the root node which has the cache. White nodes are differential nodes which must first be rerooted before be read. Note that only the root node has the cache.

Figure 15: Shallow binding scheme for functional arrays, from [Chu94, Figure 1].

FA-ROTATE(B). But then the differential node d_B will be applied after the cache is copied, which will safely overwrite the mutation we were concerned about.

Note that with hardware support for small transactions [HM93] we could cheaply perform the entire rotation atomically, instead of using this six-step approach.

6 Experimental plan

To a first approximation, one can implement the standard Java `synchronized` keyword with `atomic`, ignoring the monitor object specified by the programmer as the argument to `synchronized`. This only works for "correct" programs, where you can define "correct" as "uses the specified monitor operations to guarantee atomic operation"—but a large number of existing applications are "correct" under this interpretation. Some which aren't have race conditions which can be fixed by substituting `atomic` for the `synchronized` keyword.[9]

This similarity allows us to run most Java applications unchanged under our modified system, replacing standard Java synchronization by non-blocking atomic regions. We plan to conduct experiments with the Java operating system TOS [NB00] to compare the scalability of standard synchronization and non-blocking atomic regions. TOS is a portable Java application, so some traditional operating system functions have been omitted; in particular, TOS provides neither scheduling nor memory management services. Instead, it uses standard Java threads and Java's garbage-collected heap.

However, TOS does provide disk servers and pipe servers. I hope to get these running under the non-blocking synchronization system described here to obtain some quantitative performance metrics.

[9]See Flanagan and Qadeer [FQ03] for more discussion, including an example of "incorrect" synchronization in the Java library `StringBuffer` class which would be corrected by using `atomic`.

7 Conclusions

We examined how non-blocking synchronization is integrated into two object-oriented operating systems: Synthesis and the Cache Kernel. We briefly discussed object-oriented operating systems and the trend toward moving protection features out of the operating system and into language and software systems. We then proposed an extension to Java which would allow the use of non-blocking synchronization to implement atomic regions, and explored the use of this system to gather quantitative performance data from Java operating systems.

The intersection of language and operating system design appears to be rich in innovative possibilities. Massalin and Pu left their code synthesis language on the table, and further exploration of synchronization as a language mechanism beckons as well. Operating system designers are accustomed to using low-level implementation languages, but it seems likely that higher-level languages can provide compelling safety and protection properties and allow even smaller underlying microkernels.

At this point, non-blocking synchronization is still a black art. Efficient compiler analyses can be designed to determine safety properties of code with mutual exclusion locks, but no such techniques exist to determine the correctness of a proposed implementation of non-blocking synchronization. To underscore the point, two race conditions were found in the non-blocking data structures used by the Synthesis kernel. Universal techniques provide a possible solution, but existing methods are too heavy-weight to replace hand-coded (and possibly-correct) implementations of primitive data structures. We hope to address this lack by using functional data structures to obtain an efficient universal implementation of non-blocking synchronization for object-oriented languages.

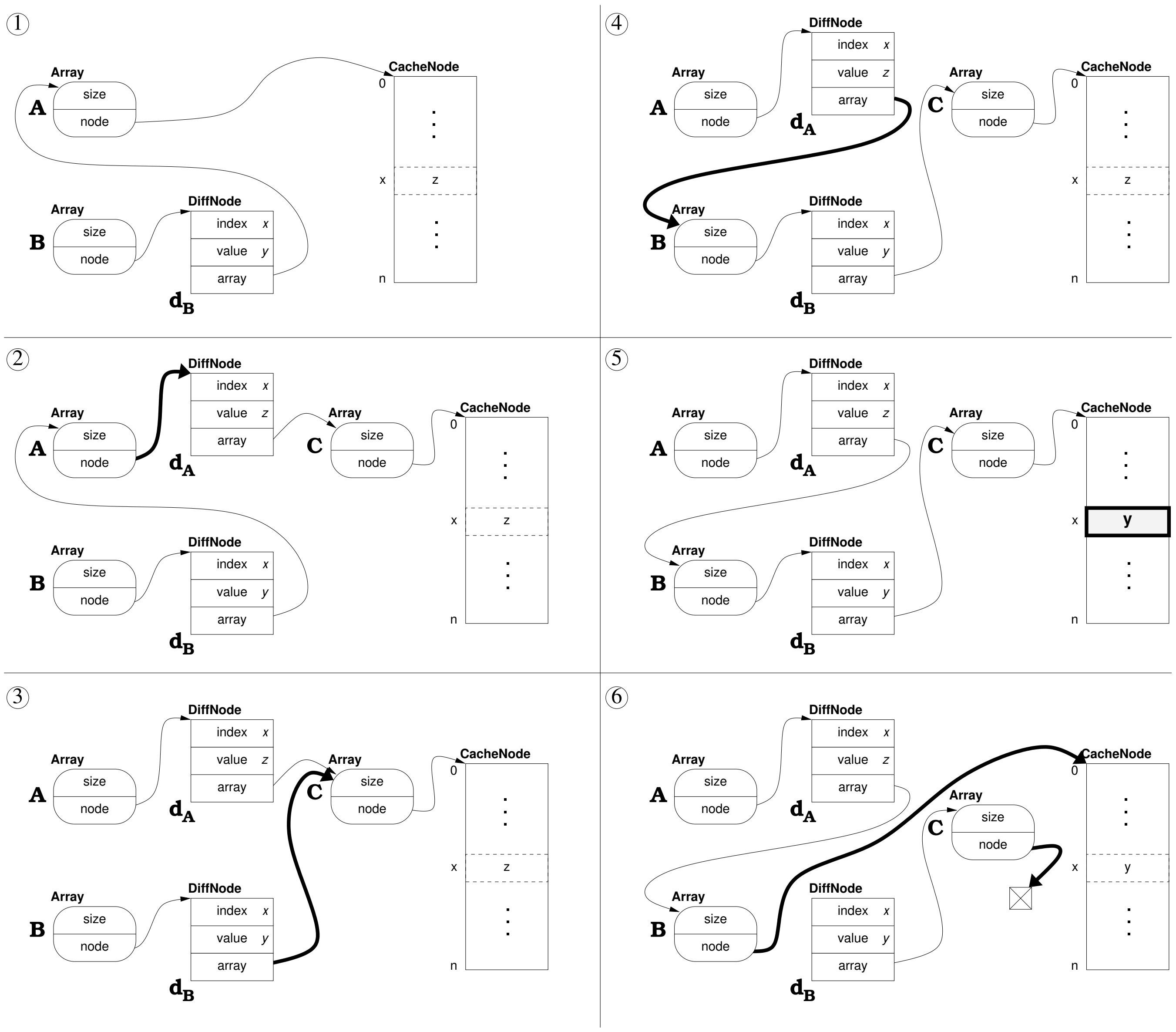


Figure 16: Atomic steps in FA-ROTATE(B). Time proceeds top-to-bottom on the left hand side, and then top-to-bottom on the right. Array A is a root node, and FA-READ(A, x) $= z$. Array B has the almost the same contents as A, but FA-READ(B, x) $= y$.

```
FA-UPDATE($A, i, v$):
begin
  $d \leftarrow$ new DiffNode($i, v, A$)
  $A' \leftarrow$ new Array($A$.size$, d$)
  return $A'$
end

FA-READ($A, i$):
begin
retry:
  $d_C \leftarrow A$.node
  if $d_C$ is a cache, then
    $v \leftarrow A$.node$[i]$
    if ($A$.node $\neq d_C$)                    [consistency check]
      goto retry
    return $v$
  else
    FA-ROTATE(A)
    goto retry
end

FA-ROTATE($B$):
begin
retry:
  $d_B \leftarrow B$.node      [step (1): assign names as per Figure 16.]
  $A \leftarrow d_B$.array
  $x \leftarrow d_B$.index
  $y \leftarrow d_B$.value
  $z \leftarrow$ FA-READ($A, x$)                 [rotates A as side effect]

  $d_C \leftarrow A$.node
  if $d_C$ is not a cache, then
    goto retry

  if ($0 =$ (random mod $A$.size))                    [random cut]
    $d'_C \leftarrow$ copy of $d_C$
    $d'_C[x] \leftarrow y$
    $s \leftarrow$ DCAS($d_C, d_C,$ &($A$.node)$, d_B, d'_C,$ &($B$.node))
    if ($s \neq$ SUCCESS) goto retry
    else return

  $C \leftarrow$ new Array($A$.size$, d_C$)
  $d_A \leftarrow$ new DiffNode($x, z, C$)

  $s \leftarrow$ CAS($d_C, d_A,$ &($A$.node))                      [step (2)]
  if ($s \neq$ SUCCESS) goto retry

  $s \leftarrow$ CAS($A, C,$ &($d_B$.array))                       [step (3)]
  if ($s \neq$ SUCCESS) goto retry

  $s \leftarrow$ CAS($C, B,$ &($d_A$.array))                       [step (4)]
  if ($s \neq$ SUCCESS) goto retry

  $s \leftarrow$ DCAS($z, y,$ &($d_C[x]$)$, d_C, d_C,$ &($C$.node))        [step (5)]
  if ($s \neq$ SUCCESS) goto retry

  $s \leftarrow$ DCAS($d_B, d_C,$ &($B$.node)$, d_C,$ nil, &($C$.node))[step (6)]
  if ($s \neq$ SUCCESS) goto retry
end
```

Figure 17: Implementation of lock-free functional array using shallow binding and randomized cuts.

References

[AF92] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134. ACM Press, August 1992.

[Bak78] Henry G. Baker, Jr. Shallow binding in Lisp 1.5. *Communications of the ACM*, 21(7):565–569, July 1978.

[Bak91] Henry G. Baker. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices*, 26(8):145–147, August 1991.

[Bar93] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270. ACM Press, June 1993.

[Ber93] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, pages 264–273, Los Alamitos, CA, May 1993. IEEE Computer Society Press.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–86. ACM Press, September 1986.

[BSP$^+$95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, David Becker, Marc Fiuczynski, and Emin Gün Sirer. Protection is a software issue. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 62–65, Orcas Island, WA, May 1995.

[BW95] Andrew P. Black and Jonathan Walpole. Objects to the rescue! or httpd: the next generation operating system. *ACM SIGOPS Operating Systems Review*, 29(1):91–95, January 1995.

[CD94] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX Association, November 1994.

[Chu94] Tyng-Ruey Chuang. A randomized implementation of multiple functional arrays. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming (LFP)*, pages 173–184. ACM Press, June 1994.

[Coh84] Shimon Cohen. Multi-version structures in Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 265–274, November 1984.

[DPM02] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys (CSUR)*, 34(4):450–468, December 2002.

[FFKF99] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or, Revenge of the son of the Lisp machine). In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 138–147. ACM Press, September 1999.

[FQ03] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349. ACM Press, June 2003.

[GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136. ACM Press, October 1996.

[HCC$^+$98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, June 1998.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290. ACM Press, August 1988.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, November 1993.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, May 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300. ACM Press, May 1993.

[Jon97] Mike Jones. What really happened on Mars? `http://research.microsoft.com/~mbj/Mars_Pathfinder/`, December 1997.

[JS91] Eric Jul and Bjarne Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 130–132, Palo Alto, CA, October 1991. IEEE.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[LaM94] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 130–140. ACM Press, August 1994.

[MP89] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 191–201, Litchfield Park, AZ, December 1989. ACM Press.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, New York, NY 10027, June 1991.

[MP92] Henry Massalin and Calton Pu. Reimplementing the Synthesis kernel on the Sony NeWS workstation. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 177–186, Seattle, WA, April 1992.

[NB00] Tyrone Nicholas and Jerzy A. Barchanski. Overview of TOS: A distributed educational operating system in Java. *ACM SIGOPS Operating Systems Review*, 34(1):2–10, January 2000.

[OB97] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–514, September 1997.

[Oka95] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 86–95. ACM Press, June 1995.

[PM91] Calton Pu and Henry Massalin. Quaject composition in the Synthesis kernel. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 29–34, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[PW93] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, Portland, OR, April 1993.

[Ree01] Jonathan Rees. Rees re: OO. `http://www.paulgraham.com/reesoo.html`, December 2001.

[SBCK03] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling Java for low-end embedded systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, pages 42–50. ACM Press, June 2003.

[SC02] Benoît Sonntag and Dominique Colnet. Lisaac: The power of simplicity at work for operating system. In *Proceedings of the 40th International Conferenece on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)*, pages 45–52. Australian Computer Society, Inc., 2002.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 214–222. ACM Press, August 1995.

[ZC96] Matthew J. Zelesko and David R. Cheriton. Specializing object-oriented RPC for functionality and performance. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, pages 175–187. IEEE Computer Society Press, May 1996.