A Methodology for Implementing Highly Concurrent Data Objects

Maurice Herlihy

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/10 October 2, 1991

Abstract

A *concurrent object* is a data structure shared by concurrent processes. Conventional techniques for implementing concurrent objects typically rely on *critical sections*: ensuring that only one process at a time can operate on the object. Nevertheless, critical sections are poorly suited for asynchronous systems: if one process is halted or delayed in a critical section, other, non-faulty processes will be unable to progress. By contrast, a concurrent object implementation is *non-blocking* if it always guarantees that some process will complete an operation in a finite number of steps, and it is *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps. This paper proposes a new methodology for constructing non-blocking and wait-free implementations of concurrent objects. The object's representation and operations are written as stylized sequential programs, with no explicit synchronization. Each sequential operation is automatically transformed into a non-blocking or wait-free operation using novel synchronization and memory management algorithms. These algorithms are presented for a multiple instruction/multiple data (MIMD) architecture in which n processes communicate by applying *read*, *write*, *load_linked*, and *store_conditional* operations to a shared memory.

1 Introduction

A *concurrent object* is a data structure shared by concurrent processes. Conventional techniques for implementing concurrent objects typically rely on *critical sections* to ensure that only one process at a time is allowed access to the object. Nevertheless, critical sections are poorly suited for asynchronous systems; if one process is halted or delayed in a critical section, other, faster processes will be unable to progress. Possible sources of unexpected delay include page faults, cache misses, scheduling preemption, and perhaps even processor failure.

By contrast, a concurrent object implementation is *non-blocking* if some process must complete an operation after the system as a whole takes a finite number of steps, and it is *wait-free* if each process must complete an operation after taking a finite number of steps. The non-blocking condition guarantees that some process will always make progress despite arbitrary halting failures or delays by other processes, while the wait-free condition guarantees that all non-halted processes make progress. Either condition rules out the use of critical sections, since a process that halts in a critical section can force other processes trying to enter that critical section to run forever without making progress. The non-blocking condition is appropriate for systems where starvation is unlikely, while the (strictly stronger) wait-free condition may be appropriate when some processes are inherently slower than others, as in certain heterogeneous architectures.

The theoretical issues surrounding non-blocking synchronization protocols have received a fair amount of attention, but the practical issues have not. In this paper, we make a first step toward addressing these practical aspects by proposing a new methodology for constructing non-blocking and wait-free implementations of concurrent objects. Our approach focuses on two distinct issues: ease of reasoning, and performance.

- It is no secret that reasoning about concurrent programs is difficult. A practical methodology should permit a programmer to design, say, a correct non-blocking priority queue, without ending up with a publishable result.

- The non-blocking and wait-free properties, like most kinds of fault-tolerance, incur a cost, especially in the absence of failures or delays. A methodology can be considered practical only if (1) we understand the inherent costs of the resulting programs, (2) this cost can be kept to acceptable levels, and (3) the programmer has some ability to influence these costs.

We address the reasoning issue by having programmers implement data objects as stylized sequential programs, with no explicit synchronization. Each sequential implementation is automatically transformed into a non-blocking or wait-free implementation via a collection of novel synchronization and memory management techniques introduced in this paper. If the sequential implementation is a correct sequential program, and if it follows certain simple conventions described below, then the transformed program will be a correct concurrent implementation. The advantage of starting with sequential programs is clear: the formidable problem of reasoning about concurrent programs and data structures is reduced to the more familiar sequential domain. (Because programmers are required to follow certain conventions, this methodology is not intended to parallelize arbitrary sequential programs after the fact.)

To address the performance issue, we built and tested prototype implementations of several concurrent objects on a multiprocessor. We show that a naive implementation of our methodology performs poorly because of excessive memory contention, but simple techniques from the literature (such as exponential backoff) have a dramatic effect on performance. We also compare our implementations with more conventional implementations based on spin locks. Even in the absence of timing anomalies, our example implementations sometimes outperform conventional spin-lock techniques, and lie within a factor of two of more sophisticated spin-lock techniques.

We focus on a multiple instruction/multiple data (MIMD) architecture in which n asynchronous processes communicate by applying *read*, *write*, *load_linked*, and *store_conditional* operations to a shared memory. The *load_linked* operation copies the value of a shared variable to a local variable. A subsequent *store_conditional* to the shared variable will change its value only if no other process has modified that variable in the interim. Either way, the *store_conditional* returns an indication of success or failure. (Note that a *store_conditional* is permitted to fail even if the variable has not changed. We assume that such *spurious* failures are rare, though possible.)

We chose to focus on the *load_linked* and *store_conditional* synchronization primitives for three reasons. First, they can be implemented efficiently in a cache-coherent architectures [9, 25], since *store_conditional* need only check whether the cached copy of the shared variable has been invalidated. Second, many other "classical" synchronization primitives are provably inadequate — we have shown elsewhere [22] that it is impossible [1] to construct non-blocking or wait-free implementations of many simple and useful data types using any combination of *read*, *write*, *test&set*, *fetch&add* [18], and memory-to-register *swap*. The *load_linked* and *store_conditional* operations, however, are *universal* — at least in principle, they are powerful enough to transform any sequential object implementation into a non-blocking or wait-free implementation. Finally, we have found *load_linked* and *store_conditional* easy to use. Elsewhere [23], we present a collection of synchronization and memory management algorithms based on *compare&swap* [24]. Although these algorithms have the same functionality as those given here, they are less efficient, and conceptually more complex.

In our prototype implementations, we used the C language [27] on an Encore Multimax [11] with eighteen NS32532 processors. This architecture does not provide *load_linked* or *store_conditional* primitives, so we simulated them using short critical sections. Naturally, our simulation is less efficient than direct hardware support. For example, a successful *store_conditional* requires twelve machine instructions rather than one. Nevertheless, these prototype implementations are instructive because they allow us to compare the relative efficiency of different implementations using *load_linked* and *store_conditional*, and because they still permit an approximate comparison of the relative efficiency of waiting versus non-waiting techniques. We assume readers have some knowledge of the syntax and semantics of C.

In Section 2, we give a brief survey of related work. Section 3 describes our model. In Section 4, we present protocols for transforming sequential implementations of small objects into non-blocking and wait-free implementations, together with experimental results showing that our techniques can be made to perform well even when each process has a dedicated processor. In Section 5, we extend this methodology to encompass large objects. Section 6 summarizes our results, and concludes with a discussion.

2 Related Work

Early work on non-blocking protocols focused on impossibility results [8, 12, 13, 14, 16, 22], showing that certain problems cannot be solved in asynchronous systems using certain primitives. By contrast, a synchronization primitive is *universal* if it can be used to transform any sequential object im-

[1] Although our impossibility results were presented in terms of wait-free implementations, they hold for non-blocking implementations as well.

plementation into a wait-free concurrent implementation. The author [22] gives a necessary and sufficient condition for universality: a synchronization primitive is universal in an n-process system if and only if it solves the well-known *consensus* problem [16] for n processes. Although this result established that wait-free (and non-blocking) implementations are possible in principle, the construction given was too inefficient to be practical. Plotkin [40] gives a detailed universal construction for a *sticky-bit* primitive. This construction is also of theoretical rather than practical interest. Elsewhere [23], the author gives a simple and relatively efficient technique for transforming stylized sequential object implementations into non-blocking and wait-free implementations using the *compare&swap* synchronization primitive. Although the overall approach is similar to the one presented here, the details are quite different. In particular, the constructions presented in this paper are simpler and more efficient, for reasons discussed below.

Many researchers have studied the problem of constructing wait-free *atomic registers* from simpler primitives [6, 7, 28, 31, 36, 38, 39, 43]. Atomic registers, however, have few if any interesting applications for concurrent data structures, since they cannot be combined to construct non-blocking or wait-free implementations of most common data types [22]. There exists an extensive literature on concurrent data structures constructed from more powerful primitives. Gottlieb et al. [19] give a highly concurrent queue implementation based on the *replace-add* operation, a variant of *fetch&add*. This implementation permits concurrent enqueuing and dequeuing processes, but it is blocking, since it uses critical sections to synchronize access to individual queue elements. Lamport [30] gives a wait-free queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. Herlihy and Wing [21] give a non-blocking queue implementation, employing *fetch&add* and *swap*, that permits an arbitrary number of enqueuing and dequeuing processes. Lanin and Shasha [32] give a non-blocking set implementation that uses operations similar to *compare&swap*. There exists an extensive literature on locking algorithms for concurrent B-trees [4, 33, 42] and for related search structures [5, 15, 17, 20, 26]. Anderson and Woll [1] give efficient wait-free solutions to the union-find problem in a shared-memory architecture.

The *load_linked* and *store_conditional* synchronization primitives were first proposed as part of the S-1 project [25] at Lawrence Livermore Laboratories, and they are currently supported in the MIPS-II architecture [9]. They are closely related to the *compare&swap* operation first introduced by the IBM 370 architecture [24].

3 Overview

A *concurrent system* consists of a collection of n sequential *processes* that communicate through shared typed *objects*. Processes are sequential — each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. We make *no* fairness assumptions about processes. A process can halt, or display arbitrary variations in speed. In particular, one process cannot tell whether another has halted or is just running very slowly.

Objects are data structures in memory. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to manipulate that object. Each object has a *sequential specification* that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that *enqueue* insert an item in the queue, and that *dequeue* remove the oldest item present in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions.

An object is *linearizable* [21] if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of non-overlapping operations is preserved. As discussed in more detail elsewhere [21], the notion of linearizability generalizes and unifies a number of *ad-hoc* correctness conditions in the literature, and it is related to (but not identical with) correctness criteria such as sequential consistency [29] and strict serializability [37]. We use linearizability as the basic correctness condition for the concurrent objects constructed in this paper.

Our methodology is the following.

1. The programmer provides a *sequential implementation* of the object, choosing a representation and implementing the operations. This program is written in a conventional sequential language, subject to certain restrictions given below. This implementation performs no explicit synchronization.

2. Using the synchronization and memory management algorithms described in this paper, this sequential implementation is transformed

into a non-blocking (or wait-free) *concurrent implementation*. Although we do not address the issue here, this transformation is simple enough to be performed by a compiler or preprocessor.

We refer to data structures and operations implemented by the programmer as *sequential objects* and *sequential operations*, and we refer to transformed data structures and operations as *concurrent objects* and *concurrent operations*. By convention, names of sequential data types and operations are in lower-case, while names of concurrent types and operations are capitalized. (Compile-time constants typically appear in upper-case.)

4 Small Objects

A *small object* is one that is small enough to be copied efficiently. In this section we discuss how to construct non-blocking and wait-free implementations of small objects. In a later section, we present a slightly different methodology for large objects, which are too large to be copied all at once.

A sequential object is a data structure that occupies a fixed-size contiguous region of memory called a *block*. Each sequential operation is a stylized sequential program subject to the following simple constraints:

- An sequential operation may not have any side-effects other than modifying the block occupied by the object.
- A sequential operation must be *total*, meaning that it is well-defined for every legal state of the object. (For example, the dequeue operation may return an error code or signal an exception when applied to an empty queue, but it may not provoke a core dump.)

The motivation for these restrictions will become clear when we discuss how sequential operations are transformed into concurrent operations.

Throughout this paper, we use the following extended example. A *priority queue* (`pqueue_type`) is a set of items taken from a totally-ordered domain (our examples use integers). It provides two operations: *enqueue* (`pqueue_enq`) inserts an item into the queue, and *dequeue* (`pqueue_deq`) removes and returns the least item in the queue. A well-known technique for implementing a priority queue is to use a *heap*, a binary tree in which each node has a higher priority than its children. Figure 1 shows a sequential implementation of a priority queue that satisfies our conditions. [2].

[2] This code is adapted from [10].

```
#define PARENT(i) ((i - 1) >> 1)
#define LEFT(i) ((i << 1) + 1)
#define RIGHT(i) ((i + 1) << 1)

void pqueue_heapify(pqueue_type *p, int i){
  int l, r, best, swap;

  l = LEFT(i);
  r = RIGHT(i);
  best = (l <= p->size && p->elements[l] > p->elements[i]) ? l : i;
  best = (r <= p->size && p->elements[r] > p->elements[best]) ? r : best;
  if (best != i) {
    swap = p->elements[i];
    p->elements[i] = p->elements[best];
    p->elements[best] = swap;
    pqueue_heapify(p, best);
  }
}

int pqueue_enq(pqueue_type *p, int x){
  int i;

  if (p->size == PQUEUE_SIZE) return PQUEUE_FULL;
  i = p->size++;
  while (i > 0 && p->elements[PARENT(i)] < x) {
    p->elements[i] = p->elements[PARENT(i)];
    i = PARENT(i);
  }
  p->elements[i] = x;
  return PQUEUE_OK;
}

int pqueue_deq(pqueue_type *p){
  int best;

  if (!p->size) return PQUEUE_EMPTY;
  best = p->elements[0];
  p->elements[0] = p->elements[--p->size];
  pqueue_heapify(p, 0);
  return best;
}
```

Figure 1: A Sequential Priority Queue Implementation

4.1 The Non-Blocking Transformation

We first discuss how to transform a sequential object into a non-blocking concurrent object. In this section we present a protocol that guarantees correctness, and in the next section we extend the protocol to enhance performance.

Omitting certain important details, the basic technique is the following. The objects share a variable that holds a pointer to the object's current version. Each process (1) reads the pointer using *load_linked*, (2) copies the indicated version into another block, (3) applies the sequential operation to the copy, and (4) calls *store_conditional* to swing the pointer from the old version to the new. If the last step fails, the process restarts at Step 1. Each execution of these four steps is called an *attempt.* Linearizability is straightforward, since the order in which operations appear to happen is the order of their final calls to *store_conditional*. Barring spurious failures of the *store_conditional* primitive, this protocol is non-blocking because at least one out of every n attempts must succeed.

Memory management for small objects is almost trivial. Each process *owns* single block of unused memory. In Step 2, the process copies the object's current version into its own block. When it succeeds in swinging the pointer from the old version to the new, it gives up ownership of the new version's block, and acquires ownership of the old version's block. Since the process that replaces a particular version is uniquely determined, each block has a unique and well-defined owner at all times. If all blocks are the same size, then support for m small objects requires $m + n + 1$ blocks.

A slow process may observe the object in an inconsistent state. For example, processes P and Q may read a pointer to a block b, Q may swing the pointer to block b' and then start a new operation. If P copies b while Q is copying b' to b, then P's copy may not be a valid state of the sequential object. This race condition raises an important software engineering issue. Although P's subsequent *store_conditional* is certain to fail, it may be difficult to ensure that the sequential operation does not store into an out-of-range location, divide by zero, or perform some other illegal action. It would be imprudent to require programmers to write sequential operations that avoid such actions when presented with arbitrary bit strings. Instead, we insert a consistency check after copying the old version, but before applying the sequential operation. Consistency can be checked either by hardware or by software. A simple hardware solution is to include a *validate* instruction that checks whether a variable read by a *load_linked* instruction has been modified. Implementing such a primitive in an architecture that already supports *store_conditional* should be straightforward, since they have similar functionalities. In our examples, however, we use a software solution. Each version has two associated counters, `check[0]` and `check[1]`. If the counters are equal, the version is consistent. To modify a version, a process increments `check[0]`, makes the modifications, and then increments `check[1]`. When copying, a process reads `check[1]`, copies the version, and then reads `check[0]`. Incrementing the counters in one order and reading them in the other ensures that if the counters match, then the copy is consistent. [3]

This protocol does not work if *compare&swap* replaces *store_conditional*. Consider the following execution: P and Q each reads a pointer to a block b, Q completes its operation, replacing b with b' and acquiring ownership of b. Q then completes a second operation, replacing b' with b. If P now does a *compare&swap*, then it will erroneously install an out-of-sequence version. Elsewhere [23], we describe a more complex protocol in which P "freezes" a block before reading it, ensuring that the block will not be recycled while the attempt is in progress. As mentioned above, the resulting protocols are more complex and less efficient than the ones described here for *store_conditional*.

Several optimizations are possible. If the hardware provides a *validate* operation, then read-only operations can complete with a successful *validate* instead of a *store_conditional*. An object may be significantly smaller than a full block. If programmers follow a convention where the object's true size is kept in a fixed location within the block, then the concurrent operation can avoid unnecessary copying. (Our prototypes make use of this optimization).

We are now ready to review the protocol in more detail (Figure 2). A concurrent object is a shared variable that holds a pointer to a structure with two fields: (1) `version` is a sequential object, and (2) `check` is a two-element array of unsigned (large) integers. Each process keeps a pointer (`new`) that points to the block it owns. The process enters a loop. It reads the pointer using *load_linked*, and marks the new version as inconsistent by setting `check[0]` to `check[1] + 1`. It then reads the old version's `check[1]` field, copies the `version` field, and then reads the `check[0]` field. If the two counters fail to match, then the copy is inconsistent, and the process restarts the loop. Otherwise, the process applies the sequential operation to the

[3] Counters are bounded, so there is a remote chance that a consistency check will succeed incorrectly if a counter cycles all the way around during a single attempt. As a practical matter, this problem is avoided simply by using a large enough (e.g., 32 bit) counter.

```
typedef struct {
  pqueue_type version;
  unsigned check[2];
} Pqueue_type;

static Pqueue_type *new_pqueue;

int Pqueue_deq(Pqueue_type **Q){
  Pqueue_type *old_pqueue;      /* concurrent object */
  pqueue_type *old_version, *new_version; /* seq object */
  int result;
  unsigned first, last;

  while (1) {
    old_pqueue = load_linked(Q);
    old_version = &old_pqueue->version;
    new_version = &new_pqueue->version;
    first = old_pqueue->check[1];
    copy(old_version, new_version);
    last = old_pqueue->check[0];
    if (first == last) {
      result = pqueue_deq(new_version);
      if (store_conditional(Q, new_version)) break;
    }                           /* if */
  }                             /* while */
  new_pqueue = old_pqueue;
  return result;
}                               /* Pqueue_deq */
```

Figure 2: Simple Non-Blocking Protocol

`version` field, and then increments `check[1]`, indicating that the version is consistent. It then attempts to reset the pointer using *store_conditional*. If it succeeds, the operation returns; otherwise the loop is resumed.

4.2 Experimental Results

The non-blocking property is best thought of as a kind of fault-tolerance. In return for extra work (updating a copy instead of updating in place), the program acquires the ability to withstand certain failures (unexpected process failure or delay). In this section, we present experimental results that provide a rough measure of this additional overhead, and that allow us to identify and evaluate certain additional techniques that substantially enhance performance. We will show that a naive implementation of the non-blocking transformation performs poorly, even allowing for the cost of simulated *load_linked* and *store_conditional*, but that adding a simple exponential backoff dramatically increases throughput.

As described above, we constructed a prototype implementation of a small priority queue on an Encore Multimax, in C, using simulated *load_linked* and *store_conditional* primitives. As a benchmark, we measure the elapsed time needed for n processes to enqueue and then dequeue $2^{20}/n$ items from a shared 16-element priority queue (Figure 3), where n ranges from 1 to 16. As a control, we also ran the same benchmark using the same heap implementation of the priority queue, except that updates were done in place, using an in-line compiled *test-and-test-and-set* [4] spin lock to achieve mutual exclusion. This *test-and-test-and-set* spin lock is a built-in feature of Encore's C compiler, and it represents how most current systems synchronize access to shared data structures.

When evaluating the performance of these benchmarks, it is important to understand that they were run under circumstances where timing anomalies and delays almost never occur. Each process ran on its own dedicated processor, and the machine was otherwise idle, ensuring that processes were likely to run uninterruptedly. The processes repeatedly accessed a small region of memory, making page faults unlikely. Under these circumstances, the costs of avoiding waiting are visible, although the benefits are not. Nevertheless, we chose these circumstances because they best highlight the inherent costs of our proposal.

[4] A test-and-test-and-set [41] loop repeatedly reads the lock until it observes the lock is free, and then tries the *test&set* operation.

```
#define million 1024 * 1024

shared Pqueue_type *object;
int N;                    /* number of processes */

process(){
  int work = million / N;
  int i;
  for (i = 0; i < work; i++)
    {
      Pqueue_enq(object, random());
      Pqueue_deq(object);
    }
}
```

Figure 3: Concurrent Heap Benchmark

In Figure 4, the horizontal axis represents the number of concurrent processes executing the benchmark, and the vertical axis represents the time taken (in seconds). The top curve is the time taken using the non-blocking protocol, and the lower curve is the time taken by the spin lock. When reading this graph, it is important to bear in mind that each point represents approximately the same amount of work – enqueuing and dequeuing 2^{20} (about a million) randomly-generated numbers. In the absence of memory contention, both curves would be nearly flat [5].

The simple non-blocking protocol performs much worse than the spin-lock protocol, even allowing for the inherent inefficiency of the simulated *load_linked* and *store_conditional* primitives. The poor performance of the non-blocking protocol is primarily a result of memory contention. In each protocol, only one of the n processes is making progress at any given time. In the spin lock protocol, it is the process in the critical section, while in the non-blocking protocol, it is the process whose *store_conditional* will eventually succeed. In the spin-lock protocol, however, the processes outside the critical section are spinning on cached copies of the lock, and are therefore not generating any bus traffic. In the non-blocking protocol, by contrast, all

[5] Concurrent executions are slightly less efficient because the heap's maximum possible size is a function of the level of concurrency.

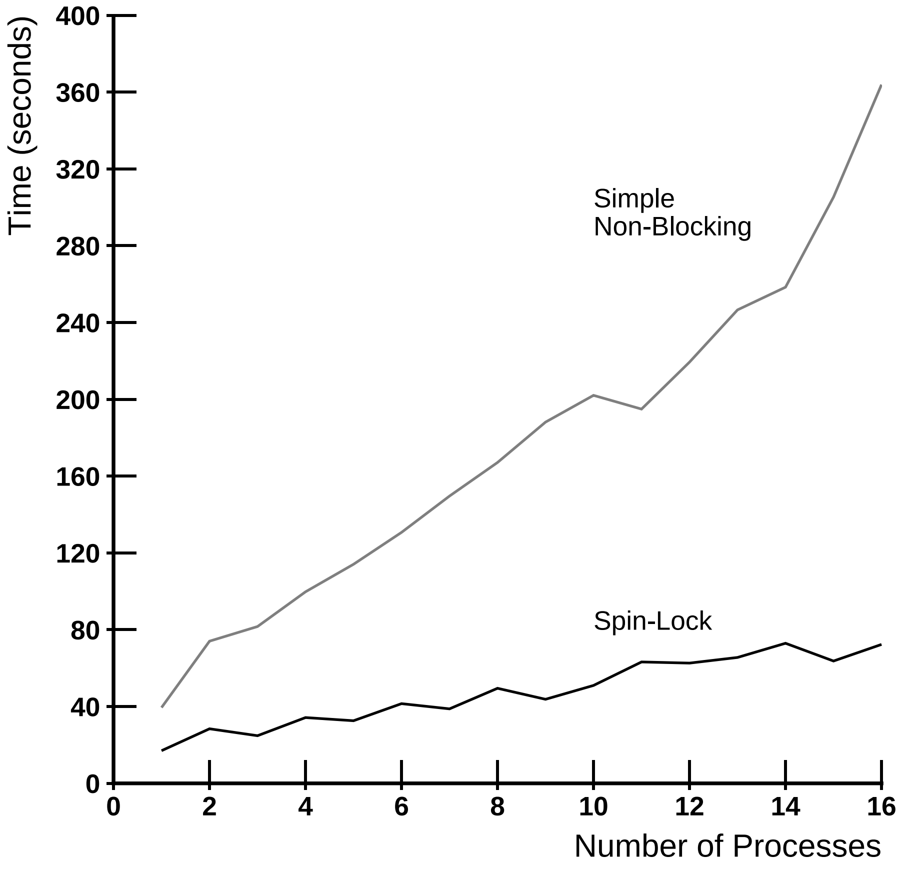

Figure 4: Simple Non-Blocking vs. Spin-Lock

	2	4	6	8	10	12	14	16
Dequeue Average	1.04	2.15	2.86	3.29	3.70	4.01	4.44	5.23
Enq Average	2.89	4.75	4.79	4.84	5.00	5.19	5.50	5.93
Deq Maximum	5	124	73	83	83	150	98	73
Enq Maximum	2046	3090	1596	2789	5207	4881	2592	178

Figure 5: Simple Non-Blocking Protocol: Number of Attempts

	2	4	6	8	10	12	14	16
Dequeue Average	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Enq Average	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Deq Maximum	1	30	60	80	60	27	48	58
Enq Maximum	1	122	138	159	216	244	254	282

Figure 6: Non-Blocking with Backoff: Number of Attempts

processes are generating bus traffic, so only a fraction of the bus bandwidth is dedicated to useful work.

The simple non-blocking protocol has a second weakness: starvation. The enqueue operation is about 10% slower than the dequeue operation. If we look at the average number of attempts associated with each process (Figure 4.2), we can see that enqueues make slightly more unsuccessful attempts than dequeues, but that each makes an average of fewer than six attempts. If we look at the *maximum* number of attempts, however, a dramatically different story emerges. The maximum number of unsuccessful dequeue attempts is in the high thousands, while the maximum number of enqueue hovers around one hundred. This table shows that starvation is indeed a problem, since a longer operation may have difficulty completing if it competes with shorter operations.

These performance problems have a simple solution. We introduce an exponential backoff [2, 34, 35] between successive attempts (Figure 7). Each process keeps a dynamically-adjusted maximum delay. When an operation starts, it halves its current maximum delay. Each time an attempt fails, the process waits for a random duration less than the maximum delay, and then doubles the maximum delay, up to a fixed limit [6].

Exponential backoff has a striking effect on performance. As illustrated in Figure 8, the throughput of the non-blocking protocol soon overtakes that of the standard spin lock implementation. Moreover, starvation is no longer a threat. In the typical execution shown in Figure 4.2, the average number of attempts is 1.00 (out of 2^{20} operations), and the maximum for enqueues is reduced by an order of magnitude.

As an aside, we point out that it is well-known that spin-locks also benefit from exponential backoff [2, 34]. We replaced the in-line compiled *test-and-test-and-set* spin lock with a hand-coded spin lock that itself employs exponential backoff. Not surprisingly, this protocol has the best throughput of all when run with dedicated processors, almost twice that of the non-blocking protocol.

In summary, using exponential backoff, the non-blocking protocol significantly outperforms a straightforward spin-lock protocol (the default provided by the Encore C compiler), and lies within a factor of two of a sophisticated spin-lock implementation.

4.3 A Wait-Free Protocol

This protocol can be made wait-free by a technique we call *operation combining*. When a process starts an operation, it records the call in an *invocation* structure (`inv_type`) whose fields include the operation name (`op_name`), argument value (`arg`), and a toggle bit (`toggle`) used to distinguish old and new invocations. When it completes an operation, it records the result in a `response` (`res_type`) structure, whose fields include the result (`value`) and toggle bit. Each concurrent object has an additional field: `responses` is an n-element array of responses, whose P^{th} element is the result of P's last completed operation. The processes share an n-element array `announce` of invocations. When P starts an operation, it records the operation name and argument in `announce[P]`. Each time a process records a new invocation, it complements the invocation's toggle bit.

A wait-free enqueue operation appears in Figure 10. After performing the consistency check, the `apply` procedure (Figure 9) scans the `responses` and `announce` arrays, comparing the *toggle* fields of corresponding invocations and responses. If the bits disagree, then it applies that invocation to

[6] For speed, each process in our prototype uses a precomputed table of random numbers, and certain arithmetic operations are performed by equivalent bit-wise logical operations.

```
static int max_delay;

int Pqueue_deq(Pqueue_type **Q)
{
  Pqueue_type *old_pqueue;
  pqueue_type *old_version, *new_version;
  int i, delay, result;
  unsigned first, last;

  if (max_delay > 1) max_delay = max_delay / 2;
  while (1) {
    old_pqueue = load_linked(Q);
    old_version = &old_pqueue->version;
    new_version = &new_pqueue->version;
    first = old_pqueue->check[1];
    copy(old_version, new_version);
    last = old_pqueue->check[0];
    if (first == last) {
      result = pqueue_deq(new_version);
      if (store_conditional(Q, new_version)) break;
    }                           /* if */
    /* backoff */
    if (max_delay < DELAY_LIMIT) max_delay = 2 * max_delay;
    delay = random() % max_delay;
    for (i = 0; i < delay; i++);
  }                             /* while */
  new_pqueue = old_pqueue;
  return result;
}
```

Figure 7: Non-Blocking Protocol with Exponential Backoff

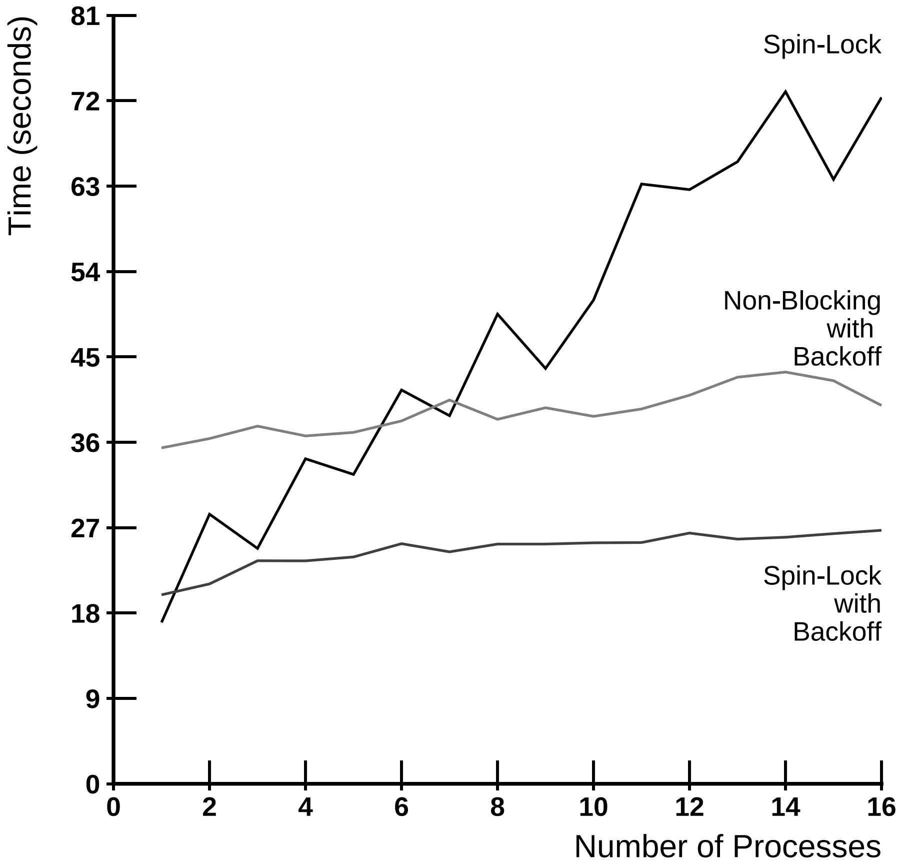

Figure 8: The Effect of Exponential Backoff

the new version, records the result in the matching position in the `responses` array, and complements the response's toggle bit. After calling the `apply` procedure to apply the pending operations to the new version, the process calls *store_conditional* to replace the old version, just as before. To determine when its own operation is complete, P compares the toggle bits of its invocation with the object's matching response. It performs this comparison *twice*; if both comparisons match, the operation is complete. This comparison must be done twice to avoid the following race condition: (1) P reads a pointer to version v. (2) Q replaces v with v'. (3) Q starts another operation, scans `announce`, applies P's operation to the new value of v, and stores the tentative result in v's `responses` array. (4) P observes that the toggle bits match and returns. (5) Q fails to install v as the next version, ensuring that P has returned the wrong result.

This protocol guarantees that as long as *store_conditional* has no spurious failures, each operation will complete after at most two loop iterations [7]. If P's first or second *store_conditional* succeeds, the operation is complete. Suppose the first *store_conditional* fails because process Q executed an earlier *store_conditional*, and the second *store_conditional* fails because process Q' executed an earlier *store_conditional*. Q' must have scanned the `announce` array after Q performed its *store_conditional*, but Q performed its *store_conditional* after P updated its invocation structure, and therefore Q' must have carried out P's operation and set the toggle bits to agree. The process applies the termination test repeatedly during any backoff.

We are now ready to explain why sequential operations must be total. Notice that in the benchmark program (Figure 3), each process enqueues an item before dequeuing. One might assume, therefore, that no dequeue operation will ever observe an empty queue. This assumption is wrong. Each process reads the object version and the `announce` array as two distinct steps, and the two data structures may be mutually inconsistent. A slow process executing an enqueue might observe an empty queue, and then observe an `announce` array in which dequeue operations outnumber enqueue operations. This process's subsequent *store_conditional* will fail, but not until the sequential dequeue operation has been applied to an empty queue. This issue does not arise in the non-blocking protocol.

Figure 11 shows the time needed to complete the benchmark program for the wait-free protocol. The throughput increases along with concurrency

[7] Because spurious failures are possible, this loop requires an explicit termination test; it cannot simply count to two.

```
void apply(inv_type announce[MAX_PROCS], Pqueue_type *object){
  int i;
  for (i = 0; i < MAX_PROCS; i++) {
    if (announce[i].toggle != object->res_types[i].toggle) {
      switch (announce[i].op_name) {
      case ENQ_CODE:
        object->res_types[i].value =
          pqueue_enq(&object->version, announce[i].arg);
        break;
      case DEQ_CODE:
        object->res_types[i].value = pqueue_deq(&object->version);
        break;
      default:
        fprintf(stderr, "Unknown operation code\n");
        exit(1);
      };                      /* switch */
      object->res_types[i].toggle = announce[i].toggle;
    }                           /* if */
  }                             /* for i */
}
```

Figure 9: The Apply Operation

because the amount of copying per operation is reduced. Nevertheless, there is a substantial overhead imposed by scanning the announce array, and, more importantly, copying the version's responses array with each operation. As a practical matter, the probabilistic guarantee against starvation provided by exponential backoff may be preferable to the deterministic guarantee provided by operation combining.

5 Large Objects

In this section, we show how to extend the previous section's protocols to objects that are too large to be copied all at once. For large objects, copying is likely to be the major performance bottleneck. Our basic premise is that copying should therefore be under the explicit control of the programmer, since the programmer is in a position to exploit the semantics of the application.

```
static Pqueue_type *new_pqueue;
static int max_delay;
static invocation announce[MAX_PROCS];
static int P;                   /* current process id */

int Pqueue_deq(Pqueue_type **Q){
  Pqueue_type *old_pqueue;
  pqueue_type *old_version, *new_version;
  int i, delay, result, new_toggle;
  unsigned first, last;

  announce[P].op_name = DEQ_CODE;
  new_toggle = announce[P].toggle = !announce[P].toggle;
  if (max_delay > 1) max_delay = max_delay >> 1;
  while ((*Q)->responses[P].toggle != new_toggle
         || (*Q)->responses[P].toggle != new_toggle) {
    old_pqueue = load_linked(Q);
    old_version = &old_pqueue->version;
    new_version = &new_pqueue->version;
    first = old_pqueue->check[1];
    memcpy(old_version, new_version, sizeof(pqueue_type));
    last = old_pqueue->check[0];
    if (first == last) {
      result = pqueue_deq(new_version);
      if (store_conditional(Q, new_version)) break;
    }                           /* if */
    /* backoff */
    if (max_delay < DELAY_LIMIT) max_delay = max_delay << 1;
    delay = random() % max_delay;
    for (i = 0; i < delay; i++);
  }                             /* while */
  new_pqueue = old_pqueue;
  return result;
}
```

Figure 10: A Wait-Free Operation

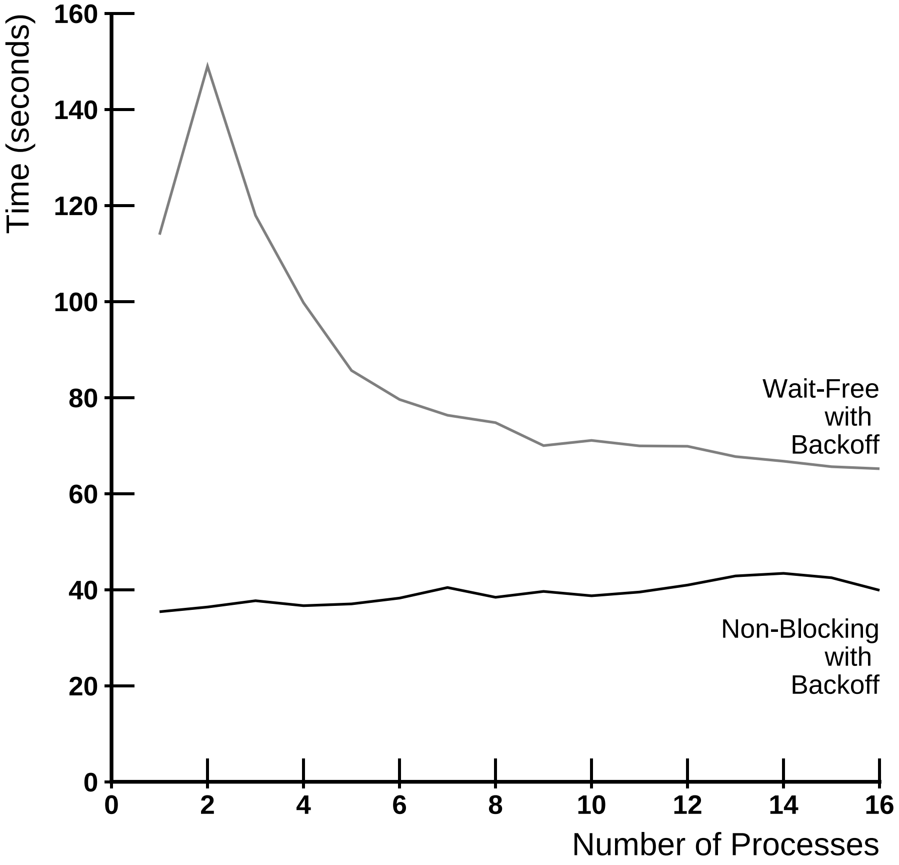

Figure 11: Non-Blocking vs. Wait-Free

A *large object* is represented by a set of blocks linked by pointers. Sequential operations of large objects are written in a *functional* style: an operation that changes the object's state does not modify the object in place. Instead, it constructs and returns a *logically* distinct version of the object. By logically distinct, we mean that the old and new versions may in fact share a substantial amount of memory. It is the programmer's responsibility to choose a sequential implementation that performs as little copying as possible.

The basic technique is the following. Each process (1) reads the pointer using *load_linked*, (2) applies the sequential operation, which returns a pointer to a new version, and (3) calls *store_conditional* to swing the pointer from the old version to the new.

Memory management is slightly more complex. Since an operation may require allocating multiple blocks of memory, each process owns its own pool of blocks. When a process creates a new version of the object, it explicitly allocates new blocks by calling *alloc*, and it explicitly frees old blocks by calling *free*. The *copy* primitive copies the contents of one block to another. If the attempt succeeds, the process acquires ownership of the blocks it freed and relinquishes ownership of the blocks it allocated.

A process keeps track of its blocks with a data structure called a *recoverable set* (`set_type`). The abstract state of a recoverable set is given by three sets of blocks: *committed*, *allocated*, and *freed*. The `set_free` operation inserts a block in *freed*, and `set_alloc` moves a block from *committed* to *allocated* and returns its address. As shown in figure 12, *alloc* calls `set_alloc` and marks the resulting block as inconsistent, while *free* simply calls `set_free`.

The recoverable set type provides three additional operations, not explicitly called by the programmer. Before executing the *store_conditional*, the process calls `set_prepare` to mark the blocks in *allocated* as consistent. If the *store_conditional* succeeds, it calls `set_commit` to set *committed* to the union of *freed* and *committed*, and if it fails, it calls `set_abort` to set both *freed* and *allocated* to the empty set.

It might also be necessary for processes to share a pool of blocks. If process exhausts its local pool, it can allocate multiple blocks from the shared pool, and if it acquires too many blocks, it can return the surplus to the shared pool. The shared pool should be accessed as infrequently as possible, since otherwise it risks becoming a contention "hot-spot." Some techniques for implementing shared pools appear elsewhere [23]; we did not use a shared pool in the prototypes shown here.

As in the small object protocol, a process checks for consistency whenever it copies a block. If the copy is inconsistent, the process transfers control back to the main loop (e.g., using the Unix `longjmp`).

5.1 Experimental Results

For the examples presented in this section, it is convenient to follow some syntactic conventions. Because C procedures can return only one result value, we follow the convention that all sequential operations return a pointer to a `result_type` structure containing a ṽalue field (e.g., the result of a *dequeue*) and a `version` field (the new state of the object). Instead of treating the sequential and concurrent objects as distinct data structures, it is convenient to treat the `check` array as an additional field of the sequential object, one that is invisible to the sequential operation.

A *skew heap* [44] is an approximately-balanced binary tree in which each node stores an item, and each node's item is less than or equal to any item in the subtree rooted at that node. A skew heap implements a priority queue, and the amortized cost of enqueuing and dequeuing items in a skew heap is logarithmic in the size of the tree. For our purposes, the advantage of a skew heap over the conventional heap is that update operations leave most of the tree nodes untouched.

The `skew_meld` operation (Figure 13) merges two heaps. It chooses the heap with the lesser root, swaps its right and left children (for balance), and then melds the right child with the other heap. To insert item x in h, `skew_enq` melds h with the heap containing x alone. To remove an item from h, `skew_deq` (Figure 14) removes the item at the root and melds the root's left and right subtrees.

We modified the priority queue benchmark of Figure 3 to initialize the priority queue to hold 512 randomly generated integers.

Figure 15 shows the relative throughput of a non-blocking skew heap, a spin-lock heap with updates in place, and a spin-lock skew heap with updates in place. The non-blocking skew heap and the spin-lock heap are about the same, and the spin-lock skew heap has almost twice the throughput of the non-blocking skew heap, in agreement with our experimental results for the small object protocol.

```
typedef struct
{
  int free_ptr, alloc_ptr;      /* next full & empty slots */
  int free_count, alloc_count;  /* number of allocs & frees */
  int size;                     /* number of committed entries */
  int old_free_ptr, old_alloc_ptr; /* reset on abort */
  Skew_type *blocks[SET_SIZE]; /* pointers to blocks */
} set_type;

Object_type *set_alloc(set_type *q){
  Object_type *x;
  if (q->alloc_count == q->size) {
    fprintf(stderr, "alloc: wraparound!\n");
    exit(1);
  }
  x = q->blocks[q->alloc_ptr];
  q->alloc_ptr = (q->alloc_ptr + 1) % SET_SIZE;
  q->alloc_count++;
  return x;
}

void set_commit(set_type *q){
  q->old_alloc_ptr = q->alloc_ptr;
  q->old_free_ptr = q->free_ptr;
  q->size = q->size + q->free_count - q->alloc_count;
  q->free_count = q->alloc_count = 0;
}

void set_prepare(set_type *q){
  int i;
  for (i = 0; i < q->alloc_count; i++)
    q->blocks[q->old_alloc_ptr + i]->check[1]++;
}

Object_type *alloc(){
  Object_type *s;
  s = set_alloc(pool);
  s->check[0] = s->check[1] + 1;
  return s;
}
```

Figure 12: Part of a Recoverable Set Implementation

```
typedef struct skew_rep {
  int value;
  int toggle;                   /* left or right next? */
  struct skew_rep *child[2];    /* left and right children */
  int check[2];                 /* inserted by system */
} Skew_type;

/*
  Skew_meld assumes its first argument is already copied.
*/
Skew_type *skew_meld(Skew_type *q, *qq){
  int toggle;
  skew_type *p;

  if (!q) return (qq);          /* if one is empty, return the other */
  if (!qq) return (q);
  p = queue_alloc(pool);        /* make a copy of q  */
  copy(qq, p);
  queue_free(pool, qq);
  if (q->value > p->value) {
    toggle = q->toggle;
    q->child[toggle] = skew_meld(p, q->child[toggle]);
    q->toggle = !toggle;
    return q;
  } else {
    toggle = p->toggle;
    p->child[toggle] = skew_meld(q, p->child[toggle]);
    p->toggle = !toggle;
    return p;
  }
}
```

Figure 13: Skew Heap: The Meld Operation

```
result_type *skew_deq(Skew_type *q) {
  Skew_type *left, *new_left, *right, buffer;
  static result_type r;

  r.value = SKEW_EMPTY;
  r.version = 0;
  if (q) {
    copy(q, &buffer);
    queue_free(pool, q);
    r.value = buffer.value;
    left = buffer.child[0];
    right = buffer.child[1];
    if (! left) {
      r.version = right;
    } else {
      new_left =  alloc(pool);
      copy(left, new_left);
      queue_free(pool, left);
      r.version = skew_meld(new_left, right);
    }
  }
  return &r;
}
```

Figure 14: Skew Heap: The Dequeue Operation

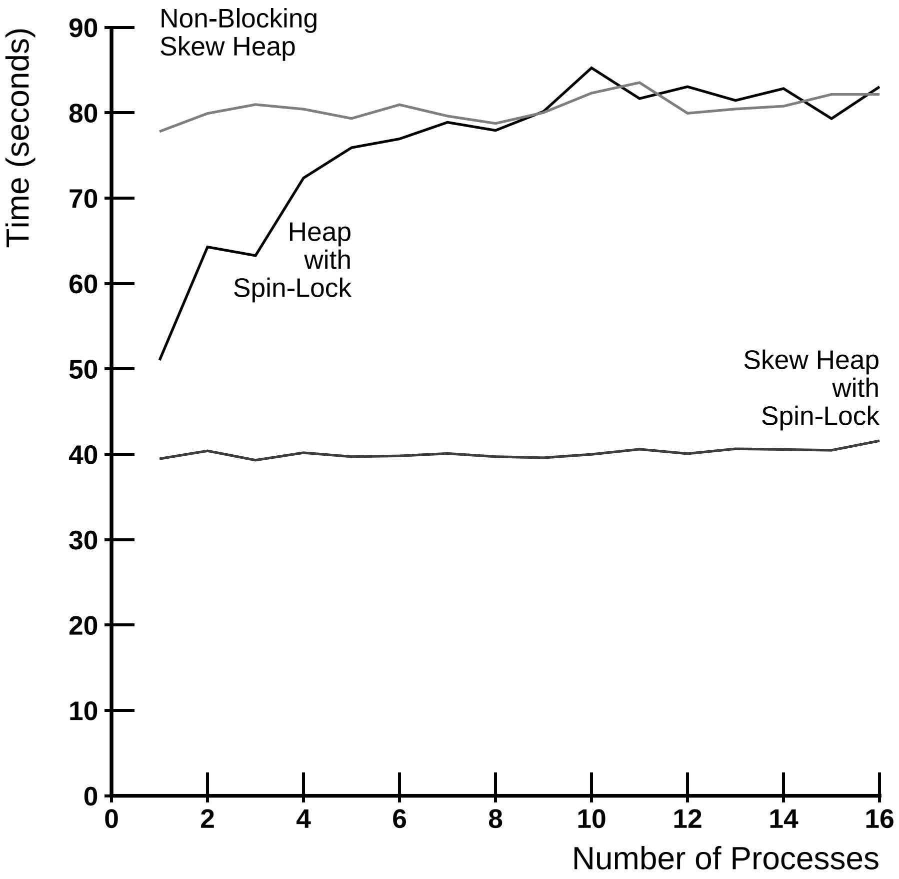

Figure 15: Large Heap Throughput

6 Conclusions

Conventional concurrency control techniques based on mutual exclusion were originally developed for single-processor machines in which the processor was multiplexed among a number of processes. To maximize throughput in a uniprocessor architecture, it suffices to keep the processor busy. In a multiprocessor architecture, however, maximizing throughput is more complex. Individual processors are often subject to unpredictable delays, and throughput will suffer if a process capable of making progress is unnecessarily forced to wait for one that is not.

To address this problem, a number of researchers have investigated *wait-free* and *non-blocking* algorithms and data structures that do not rely on waiting for synchronization. Much of this work has been theoretical. There are two obstacles to making such an approach practical: conceptual complexity, and performance. Conceptual complexity refers to the well-known difficulty of reasoning about the behavior of concurrent programs. Any practical methodology for constructing highly-concurrent data structures must include some mechanism for ensuring their correctness. Performance refers to the observation that avoiding waiting, like most other kinds of fault-tolerance, incurs a cost when it is not needed. For a methodology to be practical, this overhead must be kept to a minimum.

In the methodology proposed here, we address the issue of conceptual complexity by proposing that programmers design their data structures in a stylized sequential manner. Because these programs are sequential, both formal and informal reasoning are greatly simplified.

We address the issue of performance in several ways:

- We observe that the *load_linked* and *store_conditional* synchronization primitives permit significantly simpler and more efficient algorithms than *compare&swap*.

- We propose extremely simple and efficient memory management techniques.

- We provide experimental evidence that a naive implementation of a non-blocking protocol incurs unacceptable memory contention, but that this contention can be eliminated by applying known techniques such as exponential backoff. Our prototype implementations (using inefficient simulated synchronization primitives) outperform conventional ("test-and-test-and-set") spin-lock implementations, and lie within a factor of two of more sophisticated (exponential backoff) spin-lock implementations.

- For large objects, programmers are free to exercise their ingenuity to keep the cost of copying under control. Whenever possible, correctness should be the responsibility of the system, and performance the responsibility of the programmer.

A promising area for future research concerns how one might exploit type-specific properties to increase concurrency. Any such approach would have to sacrifice some of the simplicity of our methodology, since the programmer would have to reason explicitly about concurrency. Nevertheless, perhaps one could use our methodology to construct simple concurrent objects that could be combined to implement more complex concurrent objects, in the same way that B-link [33] trees combine a sequence of low-level atomic operations to implement a single atomic operation at the abstract level.

As illustrated by Andrews and Schneider's comprehensive survey [3], most language constructs for shared memory architectures focus on techniques for managing mutual exclusion. Because the transformations described here are simple enough to be performed by a compiler or preprocessor, it is intriguing to speculate about a programming language might support the methodology proposed here. For example, inheritance might be a convenient way to combine the object fields (e.g., check variables) used by the run-time system with those introduced by the programmer. Programming language design raises many complex issues that lie well beyond the scope of this paper, but the issue merits further attention.

References

[1] R.J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 360–370, May 1991.

[2] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] G.R. Andrews and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1983.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 1(1):1–21, 1977.

[5] J. Biswas and J.C. Browne. Simultaneous update of priority structures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 124–131, 1987.

[6] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259, 1987.

[7] J.E. Burns and G.L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

[8] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.

[9] MIPS Computer Company. The mips risc architecture.

[10] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge MA, 1990.

[11] Encore Computer Corporation. Multimax technical summary. Order Number 726-01759, Rev E.

[12] D. Dolev, C. Dwork, and L Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[13] C. Dwork, N. Lynch, and L Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):228–323, April 1988.

[14] C. Dwork, D. Shmoys, and L. Stockmeyer. Flipping persuasively in constant expected time. In *Twenty-Seventh Annual Symposium on Foundations of Computer Science*, pages 222–232, October 1986.

[15] C.S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

[16] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[17] R. Ford and J. Calhoun. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *3rd ACM Symposium on Princples of Database Systems*, pages 51–60, 1984.

[18] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer - designing an mimd parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[19] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[20] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th ACM Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[21] M. Herlihy and J. Wing. Axioms for concurrent objects. In *14th ACM Symposium on Principles of Programming Languages*, pages 13–26, January 1987.

[22] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[23] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[24] IBM. System/370 principles of operation. Order Number GA22-7000.

[25] E.H. Jensen, G.W. Hagensen, and J.M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, November 1987.

[26] D.W. Jones. Concurrent operations on priority queues. *Communications of the ACM*, 32(1):132–137, January 1989.

[27] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

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

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690, September 1979.

[30] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[31] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[32] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[33] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[34] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report 342, University of Rochester,, Rochester, NY 14627, April 1990.

[35] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[36] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[37] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[38] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[39] G.L. Peterson and J.E. Burns. Concurrent reading while writing ii: the multi-writer case. Technical Report GIT-ICS-86/26, Georgia Institute of Technology, December 1986.

[40] S.A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–176, 1989.

[41] L. Rudolph. Decentralized cache scheme for an mimd parallel processor. In *11th Annual Computing Architecture Conference*, pages 340–347, 1983.

[42] Y. Sagiv. Concurrent operations on b-trees with overtaking. In *ACM Princples of Database Systems*, pages 28–37, January 1985.

[43] A.K. Singh, J.H. Anderson, and M.G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, 1987.

[44] D.D. Sleator and R.E. Tarjan. Self adjusting binary trees. In *Proceedings of the 15th ACM Symposium on Theory of Computing*, pages 52–59, 1983.