A Methodology for Implementing Highly Concurrent Data Structures

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Center
One Kendall Square
Cambridge, MA 02139

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*, and *compare&swap* operations to a shared memory.

1 Introduction

A *concurrent object* is a data structure shared by concurrent processes. Algorithms for implementing concurrent objects lie at the heart of many important problems in concurrent systems. 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, faster processes will be unable to progress. Possible sources of unexpected delay include page faults, exhausting one's scheduling quantum, preemption, and halting failures.

By contrast, a concurrent object implementation is *non-blocking* if it guarantees that some process will complete an operation in a finite number of steps, regardless of the relative execution speeds of the processes. An implementation is *wait-free* if it guarantees that *each* process will complete an operation in a finite number of steps. The non-blocking condition permits individual processes to starve, but it guarantees that the system as a whole will make progress despite individual halting failures or delays. The wait-free condition does not permit starvation; it guarantees that all non-halted processes make progress. The non-blocking condition is appropriate for systems where starvation is unlikely, while the (stronger) wait-free condition is appropriate when some processes are inherently slower than others, as in some heterogeneous architectures. 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.

In this paper, we propose 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 via a collection of novel synchronization and memory management algorithms introduced in this paper. We focus on a multiple instruction/multiple data (MIMD) architecture in which n processes communicate by applying *read*, *write*, and *compare&swap* operations to a shared memory. The

compare&swap operation is shown in Figure 1. We chose *compare&swap* for two reasons. First, it has been successfully implemented, having first appeared in the IBM System/370 architecture [20] [1]. Second, most other "classical" primitives are provably inadequate — we have shown elsewhere [18] that it is impossible [2] 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* [13], and memory-to-register *swap*. The *compare&swap* operation, however, is *universal* — at least in principle, it is powerful enough to transform any sequential object implementation into a non-blocking or wait-free implementation.

```
compare&swap(w: word, old, new: value)
    returns(boolean)
    if w = old
      then w := new
           return true
      else return false
    end if
end compare&swap
```

Figure 1: The Compare&Swap Operation

Although we do not present specific language proposals in this paper, we believe the methodology introduced here lays the foundation for a new approach to programming languages for shared-memory multiprocessors. As illustrated by Andrews and Schneider's comprehensive survey [1], most language constructs for shared memory architectures focus on techniques for mutual exclusion and scheduling. Only recently has attention started to shift to models that permit a higher degree of concurrency [18, 19, 32]. Because our methodology is based on automatic transformation of sequential programs, the formidable problem of reasoning about concurrent data structures is reduced to the more familiar domain of sequential reasoning. As discussed below in the section on related work, the transformed implementations are simpler and more efficient, both in time and space, than earlier constructions of this kind. For example, the concurrent priority queue example in Section 4.3 is an interesting algorithm in its own right.

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 into non-blocking implementations. To illustrate our methodology, we present a novel non-blocking implementation of a *skew heap* [35], an approximately-balanced binary tree used to implement a priority queue. In Section 5, we show how to transform our non-blocking protocols into wait-free protocols. Section 6 summarizes our results, and concludes with a discussion. Since rigorous models and proofs are beyond the scope of a paper of this length, our presentation here is deliberately informal, emphasizing the intuition and motivation underlying our algorithms.

[1] The System/370's *compare&swap* returns the register's previous value in addition to the boolean condition.

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

2 Related Work

Early work on non-blocking protocols focused on impossibility results [6, 7, 8, 9, 11, 18], 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 implementation into a wait-free concurrent implementation. The author [18] gave a necessary and sufficient condition for universality: a synchronization primitive is universal in an n-process system if and only if it solves asynchronous consensus [11] for n processes. Although this result showed that wait-free (and non-blocking) implementations are possible in principle, the universal construction was too inefficient to be practical. Plotkin [32] gives a detailed universal construction for a *sticky-bit* primitive. This construction, while more efficient than the consensus-based construction, is still not entirely practical, as each operation may require multiple scans of all of memory. The universal constructions presented here are simpler and more efficient than earlier constructions, primarily because *compare&swap* seems to be a "higher-level" primitive than sticky-bits.

Many researchers have studied the problem of constructing wait-free *atomic read/write registers* from simpler primitives [4, 5, 22, 25, 28, 30, 31, 34]. 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 elementary data types such as queues, directories, or sets [18]. There exists an extensive literature on concurrent data structures constructed from more powerful primitives. Gottlieb et al. [14] 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 [24] gives a wait-free queue implementation that permits one enqueuing process to execute concurrently with one dequeuing process. Herlihy and Wing [17] give a non-blocking queue implementation, employing *fetch&add* and *swap*, that permits an arbitrary number of enqueuing and dequeuing

processes. Lanin and Shasha [26] 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 [2, 27, 33] and for related search structures [3, 10, 12, 15, 21]. Our concurrent skew heap implementation in Section 4.3 uses *futures*, a form of lazy evaluation used in MultiLisp [16].

3 Model

In this section we give an informal presentation of our model, focusing on the intuition behind our definitions. A more formal treatment appears elsewhere [17].

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 *enq* insert an item in the queue, and that *deq* 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* [17] 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 [17], 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 [23] and strict serializability [29]. We use linearizability as the basic correctness condition for all concurrent objects constructed in this paper.

A natural way to measure the time complexity of a non-blocking implementation is the *system latency*, defined to be the largest number of steps the system can take without completing an operation. For a wait-free implementation, the *process latency* is the largest number of steps a process can take without completing an operation. Both kinds of latency measure worst-case performance. We are usually interested in latency as a function of the system size. For brevity, we say that an implementation is $O(n)$ *non-blocking* (*wait-free*) if it has $O(n)$ system (process) latency. Note that an implementation using critical sections has infinite system and process latencies, since the system can take an arbitrary number of steps without completing an operation if some process is delayed in a critical section.

Our methodology is based on the following steps.

1. The *programmer* chooses a representation for the object, and implements a set of *sequential operations*. The sequential operations are written in a conventional sequential language, with no explicit synchronization. They are subject to the following important restriction: they must written in a *functional* style — an operation that changes the object state is not allowed to modify the object in place, instead it must compute and return a (logically) distinct version of the object.

2. Using the synchronization and memory management algorithms described below, each sequential operation is transformed into a *non-blocking* (or *wait-free*) *operation*. (The transformed operations typically appear to update the object in place.)

This transformation could be done by a compiler.

For example, to implement an operation with the following signature:

```
operation(x: object, a: value)
        returns(value)
```

the programmer would implement a sequential operation with the following signature:

```
OPERATION(x: object, a: value)
        returns(object, value)
```

By convention, names of sequential operations appear in small capitals, and names of non-blocking operations in lower-case.

4 Non-Blocking Protocols

4.1 Single Word Objects

We first consider objects whose values fit in a single word of memory. The sequential operation is transformed into a non-blocking operation by the *Single Word Protocol*, shown in Figure 4.1. Here, we show a sequential *fetch&add* operation, together with its non-blocking transformation. Each process reads the object's current value into a local variable, calls the sequential operation to compute a new value, and then attempts to reset the object to that new value using