Distrib. Comput. (2002) 15: 255–271
Digital Object Identifier (DOI) 10.1007/s00446-002-0079-z

DISTRIBUTED COMPUTING

Lock-free reference counting

David L. Detlefs, Paul A. Martin, Mark Moir, Guy L. Steele Jr.

Sun Microsystem Laboratories, 1 Network Drive, Burlington, MA 01803, USA (e-mail: Mark.Moir@sun.com)

Received: January 2002 / Accepted: March 2002

Abstract. Assuming the existence of garbage collection makes it easier to design implementations of dynamic-sized concurrent data structures. However, this assumption limits their applicability. We present a methodology that, for a significant class of data structures, allows designers to first tackle the easier problem of designing a garbage-collection-dependent implementation, and then apply our methodology to achieve a garbage-collection-independent one. Our methodology is based on the well-known reference counting technique, and employs the double compare-and-swap operation.

Keywords: Lockfree synchronization – Reference counting – Memory management – Dynamic data structures

1 Introduction

We present a methodology that can be used to simplify the design of dynamic-sized nonblocking shared data structures. The considerable amount of research attention that has been paid in recent years to the implementation of nonblocking data structures has mostly focused on lock-free and wait-free implementations. An implementation is *wait-free* if it guarantees that after a finite number of steps of any operation, the operation completes (regardless of the timing behaviour of threads executing other operations concurrently). An implementation is *lock-free* if it guarantees that after a finite number of steps of any operation, *some* operation completes. Both properties preclude the use of mutually exclusive locks for synchronization because if some thread is delayed while holding a lock, then no other operation that requires the lock can complete – no matter how many steps it takes – until the lock holder releases the lock. Lock-free programming is increasingly important for overcoming this and other problems associated with locking, including performance bottlenecks, susceptibility to delays and failures, design complications, and, in real-time systems, priority inversion.

Lock-freedom is weaker than wait-freedom, as it guarantees progress only on a system-wide basis, while wait-freedom guarantees per-operation progress. Research experience shows, however, that the stronger wait-freedom property is much more difficult to achieve, and usually results in more costly implementations. Furthermore, lock-freedom is likely to be sufficient in practice because in a well-designed system, contention should be low. In this paper, we concentrate on the design of lock-free data structures, and specifically *dynamic-sized* ones, that is, data structures whose size can grow and shrink over time.

It is well known that garbage collection (GC) can simplify the design of sequential implementations of dynamic-sized data structures. Furthermore, in addition to providing storage management benefits, GC can also significantly simplify various *synchronization* issues in the design of *concurrent* data structures [3,15,22]. Unfortunately, however, simply designing implementations for garbage-collected environments is not the whole solution. First, not all programming environments support GC. Second, almost all of those that do employ excessive synchronization, such as locking and/or stop-the-world mechanisms, which brings into question their scalability. Finally, an obvious restriction of particular interest to our group is that GC-dependent implementations cannot be used in the implementation of the garbage collector itself!

The goal of this work, therefore, is to allow programmers to exploit the advantages of GC in designing their lock-free data structure implementations, while avoiding its drawbacks. To this end, we provide a methodology that allows programmers to first solve the easier problem of designing a GC-dependent implementation, and to then apply our methodology in order to achieve a GC-independent one.

We have designed our methodology to preserve lock-freedom. That is, if the GC-dependent implementation is lock-free so too will be the GC-independent one derived using our methodology. Two clarifications are needed here. First, some readers may be concerned about our reference to lock-free, GC-dependent implementations, given that most garbage collectors stop threads from executing while they perform their work. However, this does not mean that the *data structure implementation* is itself not lock-free: the definition of lock-freedom does not require progress if the system (say, the GC or the OS) prevents threads from taking any steps.

Second, we say that we achieve lock-free, GC-independent implementations of dynamic-sized data structure by applying our methodology, but we do not specify how objects are cre-

ated and destroyed; usually malloc and free are not lock-free, so implementations based on them are also not lock-free. However, most production-quality memory allocators do attempt to avoid contention for locks – for example by maintaining separate allocation buffers for each thread – and therefore avoid most of the problems associated with locks. In fact, our colleagues Dice and Garthwaite have recently proposed a new memory allocator [5] that is designed to be "multiprocessor-aware" in order to improve locality and reduce synchronization overhead. As a result, their allocator resorts to locking only very occasionally, and therefore scales very well. While their allocator does still use locks, variations on straightforward and well-known nonblocking techniques could be applied to make it completely nonblocking.

An important feature of our methodology – and data structure implementations based on it – is that it allows the memory consumption of the implementation to grow and shrink over time, without imposing any restrictions on the underlying memory allocation mechanisms. In contrast, lock-free implementations of dynamic data structures often either require maintenance of a special "freelist", whose storage cannot in general be reused for other purposes (e.g. [19,25]), or require special system support such as type stable memory [7].

Our methodology is based on the well-known garbage collection technique of *reference counting*. We refer to our methodology as LFRC (Lock-Free Reference Counting). In this approach, each object contains a count of the number of pointers that point to it, and can be freed if and only if this count is zero. The reference counting approach to memory management has the limitation that it does not detect cycles in garbage (imagine a ring of unreachable objects in which each has a pointer to the next – all reference counts are at least one, so none of the objects are ever identified as garbage). LFRC inherits this limitation. In many cases, either cycles do not exist in garbage, or the implementation can be easily modified to eliminate this possibility. In other cases, it is more difficult or even impossible to do so; in such cases our methodology cannot guarantee that all garbage will be reclaimed.

In order to maintain accurate reference counts, we would like to be able to atomically create a pointer to an object and increment that object's reference count, and to atomically destroy a pointer to an object and decrement its reference count. This way, by freeing an object when and only when its reference count becomes zero, we can ensure that objects are not freed prematurely, but that they are eventually freed when no pointers to the object remain.

The main difficulty that arises in the above-described approach is the need to atomically modify two separate memory locations: a pointer and the reference count of the object to which it points. This can be achieved either by using hardware synchronization primitives to enforce this atomicity, or by using lock-free software mechanisms to achieve the appearance of atomicity. Work on the latter approach has yielded various lock-free and wait-free implementations of multi-variable synchronization operations (e.g. [20,23]). However, all of these results are complicated, introduce significant overhead, and/or work only on locations in statically-allocated memory, rendering them unsuitable for our goal of supporting dynamic-sized nonblocking data structures. Therefore, in this paper, we explore the former approach. In particular, we assume the availability of a double compare-and-swap (DCAS) instruction that can atomically access two independently-chosen memory locations. (DCAS is precisely defined in Sect. 2.) While such an instruction is not widely available, it has been implemented in hardware in the past (e.g. [21]). Furthermore, it is important to investigate the role of hardware synchronization mechanisms in determining the feasibility of scalable nonblocking synchronization. Thus it is appropriate to study what can be achieved with alternative hardware mechanisms.

Even DCAS does not allow us to maintain reference counts that are accurate at all times. For example, if a pointer in shared memory points to an object v, and we change the pointer to point to another object w, then we have to atomically modify the pointer, increment w's reference count, *and* decrement v's reference count. However, it turns out that a weaker requirement on the reference counts suffices, and that this requirement *can* be achieved using DCAS. This weakening is based on the observation that reference counts do not always need to be accurate. The important requirements are that if the number of pointers to an object is non-zero, then so too is its reference count, and that if the number of pointers is zero, then the reference count eventually becomes zero. These two requirements respectively guarantee that an object is never freed prematurely, and that the reference count of an object that has no pointers to it eventually become zero, so that it can be freed.[1]

These observations imply that it is safe for a thread to increment an object's reference count *before* creating a new pointer to it, provided that the thread eventually either creates the pointer, or decrements the reference count to compensate for the previous increment. This is the approach used by our methodology.

The observation that a weaker property for reference counts is sufficient entices us to consider using a single-location compare-and-swap (CAS) instruction, which is widely available, to maintain reference counts. However, when we load a pointer from a shared memory location, we need to increment the reference count of the object to which the loaded value points. If we can access this reference count only with a single-location CAS, then there is a risk that the object will be freed before we increment the reference count, and that the subsequent attempt to increment the reference count will corrupt memory that has been freed, and potentially reallocated for another purpose. By using DCAS, we can increment the reference count while atomically ensuring that a pointer still exists to this object. While this scenario explains why a naive approach to using CAS for maintaining reference counts does not work, it is by no means a formal impossibility argument. In fact, our subsequent work on constructing such an argument resulted in an effective CAS-based approach for handling memory management in nonblocking dynamic-sized shared data structures [8–10]. This development adds more weight to our argument that research based on synchronization primitives that are not widely available is appropriate and important.

To demonstrate the utility of the LFRC methodology, we show how it can be used to convert a GC-dependent algorithm published recently in [3] into a GC-independent one. The algorithm presented in [3] is a lock-free implementation of a

[1] To be precise, we should point out that it is possible for garbage to exist and to never be freed in the case where a thread fails permanently.