acmqueue

The Balancing Act of Choosing Nonblocking Features

Design requirements of nonblocking systems

Maged M. Michael

What is nonblocking progress? Consider the simple example of incrementing a counter `C` shared among multiple threads. One way to do so is by protecting the steps of incrementing `C` by a mutual exclusion lock `L` (i.e., `acquire(L); old := C ; C := old+1; release(L);`). If a thread `P` is holding `L`, then a different thread `Q` must wait for `P` to release `L` before `Q` can proceed to operate on `C`. That is, `Q` is blocked by `P`.

Now consider an implementation of the increment operation using the Compare-and-Swap (CAS) atomic primitive. CAS atomically reads a shared location and compares the read value with an expected value. If they are equal, it writes a new value to the location and returns an indicator of equality with the expected value. In the following implementation `&C` is the address of `C`:

```
do {old := C; } until CAS(&C, old, old+1);
```

This implementation is nonblocking because no thread can be blocked by the inaction of other threads (caused by, for example, preemption, a page fault, or even termination), regardless of where the other threads stop.

There are three established levels of nonblocking progress: obstruction-free, lock-free, and wait-free. The weakest is obstruction freedom. An operation is *obstruction-free* if it is guaranteed to complete in a finite number of steps when running alone.[9] An obstruction-free operation does not need to wait for actions by other threads, regardless of where they have stopped. An obstruction-free operation, however, may starve or end up in a livelock scenario with one or more concurrent operations where the actions of the threads prevent them all from making progress.

Lock-free progress combines obstruction-free progress with livelock freedom. That is, in a system with only lock-free operations, whenever an operation takes a finite number of steps, some operation (maybe a different one) must have completed during that period.

Finally, *wait-free* progress[7] combines lock-free progress with starvation-freedom. That is, a wait-free operation is guaranteed to complete in a finite number of its own steps, regardless of the actions or inaction of other operations.

It is worth noting that these levels of progress guarantees require that primitive memory-access steps have at least the same level of progress guarantee at the hardware-arbitration and cache-coherence levels. For example, a cache-coherence protocol that may cause some primitive memory accesses to be retried indefinitely cannot support a wait-free algorithm that uses such primitives.

USES OF NONBLOCKING OPERATIONS

Nonblocking operations are often used for systems or interthread interactions where having threads wait for actions by other threads makes the system vulnerable to deadlock, livelock, and/or prolonged delays. Examples of such uses are:

- **Async signal safety.** This allows signal handlers of asynchronous signals to share data or services with the interrupted thread, with a guarantee that the signal handler never needs to wait for actions by the interrupted thread—which, as a result of being interrupted, cannot run until the signal handler completes. Nonblocking operations used in particular by the signal handlers can guarantee the absence of such waiting and provide async signal safety.
- **Kill-safe systems.** Such systems guarantee availability even if processes may be terminated at arbitrary points. This requirement arises in server systems, where the ability to terminate processes representing client requests allows high server throughput. Such systems must guarantee availability even when processes may be terminated at arbitrary points while operating on shared structures or services. When using nonblocking operations in such a system, the remaining processes never have to wait for action—which will never happen—by a terminated process.
- **Soft realtime applications on commodity systems.** Using nonblocking operations on such systems (e.g., media players) helps avoid priority inversion and provides preemption tolerance. That is, it eliminates the possibility that an active thread—potentially executing a high-priority task—is blocked awaiting action by other threads that are delayed for a long time (e.g., because of page faults). This helps make long noticeable delays highly unlikely in such systems.

SELECTING NONBLOCKING FEATURES

Unlike the example of the shared counter, almost any nonblocking operation that involves more than one location presents multiple considerations that are often at odds. This article examines trade-offs and compromises that have to be considered in selecting features of nonblocking operations. These tradeoffs are seldom straightforward and sometimes require compromises in order to achieve the bare minimum of system requirements. Balancing these tradeoffs can be daunting if you start unaware of all the potential pitfalls.

The goals of this article are to walk the reader through the various issues and features of nonblocking operations and the various choices to be made; to understand the interactions among these options; and to develop a sense of the best path to take to disentangle the interdependencies among these choices and quickly prune options that are incompatible with the main objectives of using nonblocking operations in the first place.

For some uses of nonblocking operations, such as kill-safety, the need for nonblocking progress likely involves operations on multiple data structures and services. In these cases, one has to consider the totality of the interdependence of the features of the various nonblocking components to reach an acceptable solution.

A RUNNING EXAMPLE

Figure 1 presents a variant of the classic lock-free IBM LIFO (last-in first-out)-free list algorithm[10] as a running example. The variable `Header` consists of a packed pointer-integer pair that can be read and operated on atomically by a double-width CAS. The integer size is assumed to be large enough

FIGURE 1

Variant of the Classic Lock-free IBM LIFO Free List Algorithm

```
Structures:
Header : <pointer, integer> // Initially: <null, 0>
Push (b : pointer) : void
1: h = Header;              // single-width read of Header
2: *b = h;                  // Make the pushed block point
                            // to the old header
3: if CAS(&Header, h, n) return ; else goto 1;
                            // use single width CAS to try to make
                            // the pushed block the new header
                            // if Header is still equal to h

Pop () : pointer
1: <h,t> = Header; if h == null return null;
                            // double-width read of Header
2: n = *h;                  // read the pointer in the first block
3: if CAS(&Header, <h,t>, <n,t+1>) return n ; else goto 1;
                            // Use double-width CAS to try to make
                            // the pointer in the popped block
                            // become the new Header, if Header
                            // has not changed since it was read
                            // in line 1
```

that it never overflows. The notation `*b` is used to indicate the location pointed to by a pointer `b`. Depending on the context, reads and CAS operations on `Header` are either single- or double-width.

Ignoring for now why an integer is packed with the pointer in `Header` (explained later), the reader can notice that the inaction of any number of threads stopped anywhere in the Push and Pop operations cannot block other threads from operating on the list. In fact, since Push and Pop are not only nonblocking but also lock-free, then as long as there are active threads attempting these operations, some operation will complete. Whenever any thread takes five steps in one of these operations, some operation must have completed (by the same or a different thread) during that period.

KEY ISSUES IN SELECTING NONBLOCKING FEATURES

The following are the key issues to consider in selecting the features of nonblocking operations.

LEVELS OF PROGRESS GUARANTEE

The appropriate level of nonblocking progress (e.g., obstruction-free vs. lock-free) and the extent of

use of nonblocking operations depend on the progress guarantees required by the system.

To achieve async signal safety using nonblocking operations, only the signal handler's operations need to be nonblocking, while operations by the main threads can be blocking. For example, in a case where signal handlers may search hash tables that may be updated by threads that may be interrupted, a fully nonblocking algorithm is not necessary, whereas an algorithm with nonblocking search operations and blocking add and remove operations (as described by Steve Heller et al.[6]) can be sufficient.

If the only concern is the interaction between the signal handler and the interrupted thread, then using obstruction-free operations is sufficient. Kill-safety tends to require broader use of nonblocking operations to access objects shared by processes that are subject to arbitrary termination. Similar to async signal safety, the use of obstruction-free operations suffices to provide availability. Using lock-free or wait-free operations guarantees livelock freedom or starvation freedom, respectively.

For avoiding priority inversion in soft realtime applications, it may suffice to make the high-priority operation obstruction-free; however, stronger progress guarantees (lock-free or wait-free) are clearly desirable in this context.

Wait-free algorithms tend to entail space at least linear in the number of threads that may operate concurrently.[1] Aside from the space overheads, some recent wait-free algorithms show reasonably competitive performance.[5,19] Lock-free algorithms can achieve competitive performance with blocking algorithms and do not have inherently high space requirements. There are hardly any custom obstruction-free algorithms for specific data structures that are not lock-free; however, obstruction-free algorithms for arbitrary atomic sections (or transactional memory) are evidently simpler than corresponding lock-free algorithms.

Accordingly, in choosing the appropriate level of nonblocking progress to use, you must take into account what is absolutely needed to achieve the primary requirement (e.g., kill-safety) vs. what is desirable (e.g., wait-free progress) and what this entails for the other issues.

If the number of threads that can concurrently execute certain operations is limited, then strong progress guarantees can be achieved by using simpler algorithms than when higher levels of concurrency are allowed (e.g., single-producer or single-consumer queues vs. multiproducer multiconsumer queues).

An issue that is sometimes missed is the effect of read operations on update operations. Consider two buffer implementations. Both support *check* (if empty) and *add* operations. In both implementations, operations are lock-free. The first implementation guarantees that check operations never prevent an add operation from completing. In the other, check operations may interfere with and prevent add operations from completing (e.g., as a result of using reference counting, as discussed later).

You can see how problematic the latter implementation is when the act of checking if the buffer is not empty can prevent items from ever being added to the buffer. Therefore, in selecting the appropriate level of progress for a nonblocking operation, it is not enough just to require the implementation, for example, to be lock-free. It is also important to decide which operations must be immune from interference by others. In the buffer example, while add operations are lock-free with respect to each other in both implementations, it is desirable that an individual add operation be wait-free with respect to any number of concurrent check operations.

CHOICE OF DATA STRUCTURES

The choice of data structures is one of the most important decisions in designing a nonblocking environment. This step is often overlooked because data-structure choices may be inherited from sequential or blocking designs. In choosing data structures, designers need to consider the minimum requirements and the range of desirable characteristics.

For example, if lock-free FIFO lists are considered, one should ask whether FIFO order is indeed necessary or if a more relaxed order is acceptable. If it is the latter, then the design may be simplified and performance under certain conditions may be improved by using lock-free LIFO lists instead. The classic LIFO list algorithm has simpler memory safety requirements (as discussed later) and in general has shorter path length than FIFO algorithms.

Another example is that a search structure that may be a good choice in sequential code (e.g., balanced trees) may not be the best choice in nonblocking systems compared with hash structures. Also, static structures such as hash tables with open addressing can be simpler to manage than dynamic structures such as hash tables with chaining.

SAFE MEMORY RECLAMATION ISSUES

Nonblocking operations, by definition, do not wait for actions by other nonblocking operations and cannot expect to prevent other nonblocking operations from taking actions. This poses a problem for nonblocking operations that dereference pointers to dynamic structures that could be removed and freed by other operations. Without proper precautions a nonblocking operation may be about to access a dynamic block when the block gets removed from the structure and freed by another operation. This could lead to problematic outcomes such as an access violation if the block were freed back to the operating system, corrupting a different structure that happens to allocate and use the memory of the freed block, or returning an incorrect result.

Using the LIFO-free list (in figure 1) as an example, one can see that a Pop operation may lead to accessing free memory. For example, a thread `P` reads a pointer to address `A` from the variable `Header` in line 1 of Pop. Then, another thread `Q` pops the block at `A` from the list and frees it back to the operating system. Now `P` proceeds to line 2 in Pop and dereferences the pointer to `A`, potentially suffering an access violation.

Paul McKenney[13] offers a detailed discussion of safe memory reclamation issues and solutions. The following is a brief list of categories of memory-safety solutions and their implications:

- **Automatic GC (garbage collection).** On systems with GC, such as Java applications, memory safety is implicitly guaranteed. In the preceding example, as long as thread `P` holds a reference to block `A`, block `A` is guaranteed not to be freed. Of course, this raises the question of whether GC itself is nonblocking.
- **RCU (Read-Copy-Update) and epoch-based collectors.** Briefly, RCU-like solutions guarantee that blocks removed from certain data structures are not freed until it can be established that all threads have reached quiescence points where it is impossible that any threads can still be holding references to those blocks.[14]
- **Reference counting.** Blocks are associated with counters that are incremented and decremented as threads acquire and release references to such blocks. Typically, a block is freed only when its reference count goes to zero and it is guaranteed that no new references to it can be created.[20]
- **Hazard pointers.** Briefly, in this method each thread that may traverse blocks that may be

removed and freed owns a number of hazard pointers. Before dereferencing a pointer, a thread sets one of its hazard pointers to the pointer value. Other threads that may remove the block guarantee that they will not free the block until no hazard pointers point to it.[8,16]

Hardware transactional memory may simplify nonblocking safe memory reclamation.[4] As discussed later, however, most upcoming mainstream HTM implementations are best-effort and require an alternate non-HTM path.

The options for safe memory reclamation in order of increasing flexibility (and difficulty of memory-safety solutions) are:

- Freed blocks will be reused but never freed for different uses.
- Freed blocks can be coalesced and reused for different types and sizes but not returned to the operating system.
- Freed blocks may be coalesced, reused arbitrarily, or returned to the operating system.

Note that for some algorithms (e.g., the classic LIFO list), memory safety might not be a problem if the operating system supports nonfaulting loads. In the scenario just mentioned of thread `P` reading address `A` in line 2 of Pop after the block at `A` was freed to the operating system, a system with nonfaulting loads would allow such a read.

ABA PREVENTION

The ABA problem is common in optimistic concurrency algorithms, including nonblocking ones. The term ABA refers to the change of the value of a shared variable from `A` to `B` and back to `A` again. Using the LIFO-list Pop operation as an example and ignoring the packed integer with `Header` for now, the following is an ABA problem scenario starting with a list that includes blocks `A` and `B`:
(a.) A thread `P` reads the value `A` from `Header` in line 1 and `B` from `*A` in line 2;
(b.) Other threads pop blocks `A` and `B` and push blocks `C` and `A`, leaving `Header` holding the value `A` again;
(c.) `P` performs a CAS operation on `Header` with parameters `A` and `B` in line 3, and the CAS succeeds.

Now the list is corrupted. `Header` points to `B`, which is not in the list anymore. What went wrong is that (without the packed integer) `P`'s Pop algorithm cannot differentiate between the case where `Header` never changed between lines 1 and 3 and the scenario above where the list changed but `Header` returned to its old value before the CAS in line 3.

The classic LIFO algorithm packs an integer with the pointer in the `Header` variable and is designed such that the counter will change if the list changes between lines 1 and 3 of Pop (assuming that the counter never overflows).

Packing an integer with the main content of variables vulnerable to the ABA problem is the classic ABA prevention solution.[2] It remained the only solution for decades, but it has its limitations. It requires wide atomic operations to allow space for a large enough integer that cannot overflow (or at least cannot overflow in the lifetime of one operation, such as Pop in the example). Another disadvantage of this solution is that it requires the integer subfield to retain its semantics indefinitely. This can make it very difficult to free the memory of dynamic blocks that include variables packed with ABA-prevention counters, thus meaning that memory cannot be coalesced or reused for different purposes.

Although the ABA problem can occur in algorithms that do not use dynamic memory, its solutions are intertwined with safe memory reclamation solutions. First, as already mentioned, the

classic ABA solution can hinder memory reclamation. Second, any memory-safety solution (GC, RCU, reference counting, hazard pointers) can be adapted to construct an ABA solution, possibly with an added level of indirection.

An advantage of the RCU type of solution is that traversal operations have almost no overhead, while reference counting and hazard pointers have nontrivial overhead for traversal operations. On the other hand, unlike RCU, reference counting and hazard-pointer methods guarantee bounds on the number of removed blocks that are not ready for reuse.

A disadvantage of reference counting that can be significant in some cases is that it can cause a supposedly read-only operation (such as the check operation mentioned earlier) to actually write to shared data (i.e., reference counters) and prevent update operations from ever completing.

The operations LL (load-linked), SC (store-conditional), and VL (validate) are inherently immune to the ABA problem. `LL(location)` returns the value in a shared location. `VL(location)` returns a Boolean indicator of whether or not the shared location has been written by another thread since the last LL to it by the current thread. `SC(location, value)` writes a new value to the location if and only if it has not been written to by another thread since it was last LL'ed by the current thread, and it returns a Boolean indicator of the occurrence of such a write. If the read in line 1 of Pop is replaced with `LL(&Header)` and the CAS in line 3 of Pop is replaced with `SC(&Header,n)`, then the Pop operation would be immune to the ABA problem, without the need for using a packed integer.

Actual architectures that support LL/SC (e.g., IBM Power PC) do not support the full semantics of ideal LL/SC/VL. None supports the VL operation, and all impose restrictions on what can be executed between LL/SC and prohibit the nesting or interleaving of LL/SC pairs on different locations. So, while actual LL/SC support can help the lock-free LIFO algorithm avoid the ABA problem, it is limited in preventing the ABA problem in general.

While implementations of ideal LL/SC/VL present an absolute ABA prevention solution,[18] their strong semantics disallow many correct scenarios. For example, all ABA scenarios in the lock-free LIFO-list Push operation are benign. Therefore, it is advisable to consider algorithms expressed in terms of reads and CAS operations, and address only the harmful cases of ABA, such as Pop but not Push in the lock-free LIFO-list algorithm.

It is advisable to consider ABA prevention and safe memory reclamation solutions together to avoid unnecessary duplication of overhead or solutions that are contradictory or contrary to the overall system requirements.

PORTABILITY OF REQUIRED ATOMIC OPERATIONS

The range of hardware support for atomic operations needed for nonblocking algorithms and methods varies significantly. If portability is an important issue, designers need to take that into account in selecting data structures, algorithms, and supporting methods for safe memory reclamation and management, and ABA prevention.

Hardware support requirements include:

- **No support.** For example, the reading and writing of hazard pointers can be nonatomic.[16]
- **Nonuniversal atomic operations (**such as fetch-and-add, test-and-set, and atomic swap**).** Maurice Herlihy showed that it is impossible to design wait-free (and lock-free) implementations of certain data types that can be operated on by an arbitrary number of concurrent threads using only such (nonuniversal) operations.[7]

- **Compare-and-swap.** Herlihy showed that CAS is a universal operation and can be used to design implementations of any data type with wait-free operations without limitation on the maximum number of threads that operate on it concurrently. Pointer-size CAS may suffer from the ABA problem. The classic solution to that problem requires the use of wider atomic operations (e.g., double-width load and CAS primitives).
- **The pair LL and SC** (e.g., `larx` and `stcx` on the IBM Power architecture). These were also shown by Herlihy to be universal operations. As already discussed, they are immune to the ABA problem in some cases in which CAS is susceptible. Therefore, pointer-size LL/SC may suffice or entail simpler code where double-width CAS is needed in the absence of LL/SC.
- **Hardware transaction memory.** Recently IBM (Blue Gene/Q,[21] System Z,[12] and Power[3]) and Intel[11] architectures are offering hardware support for arbitrary memory transactions. However, most of these HTM architectures (except IBM System Z) are best-effort (i.e., they require that programmers provide a nontransactional path in case the hardware transactions never succeed).

Note that if the number of threads that can concurrently execute certain operations is limited, nonuniversal atomic operations may suffice to design wait-free and lock-free implementations. For example, wait-free single-producer or single-consumer FIFO queue operations[15] (by skipping the appropriate locks in the two-lock algorithm) and single-updater sets with lock-free lookup by multiple threads[16] can be implemented with just atomic loads and stores.

CHOICE OF LANGUAGE AND MEMORY ORDERING

In addition to the variety of requirements of atomic operations, nonblocking algorithms vary in their requirements of memory-ordering primitives. For example, in the Push operation of the running LIFO-list example (in figure 1), the write in line 2 must be ordered before the write (in the case of a successful CAS) in line 3. Hardware platforms and high-level programming languages offer explicit primitives, as well as implicit guarantees of ordering among memory accesses, usually specified as the architecture or language memory consistency model.

Prior to Java 5 (2004) and C11/C++11, these languages could not be reliably used (as specified by their standards) to fully express the required memory ordering. Programmers and custom library writers relied on assembly language and machine binary codes to express such ordering.

Now with C11/C++11 and Java (5 and later), programmers can designate some variables as subject to special ordering (volatiles in Java and atomics in C11/C++11). In some cases, these designations can be heavy-handed in imposing order around such variables even when not needed by the algorithms. Standard C11/C++11 offers finer levels of ordering that allow the programmer to specify the desired order.

The implementations of such languages have varying overheads on different hardware platforms. Designers of nonblocking implementations should take into account the choice of high-level languages and their memory-ordering performance on the targeted hardware platforms.

CHOICE OF ALGORITHMS

The combined choice of data structures and algorithms is one issue where big compromises can be made to design a system that meets its overall requirements.

Nonblocking algorithms vary in their portability (e.g., requirement of special hardware support), reliability (e.g., whether or not they are widely used), complexity (e.g., reasonable ease of

implementation, maintenance, and modification), progress guarantees (e.g., wait-free, lock-free, etc.), and memory safety and ABA-prevention features (e.g., compatibility or incompatibility with certain methods). The choice of algorithms is intertwined with most of the issues discussed here.

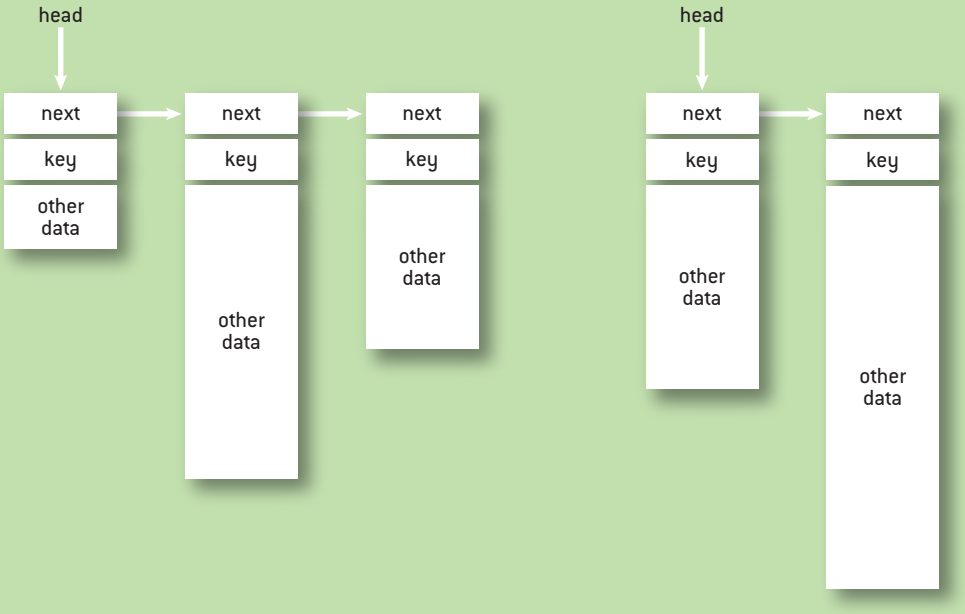
CASE STUDY

The purpose of the following example is to demonstrate the interactions among nonblocking features and issues discussed in this article.

Consider a simplified example of a kill-safe system that requires processes to share large potentially variable-size records (each with a unique key value) that can be deleted or moved arbitrarily among various data structures). Operations on the records and data structures are search, add, delete, and update of certain fields. Figure 2 shows dynamic variable-size records that can be freed and moved among structures.

Considering that the records can be moved back and forth among data structures, any nonblocking algorithm will have to deal with the ABA problem. Since the records can be large and variably sized, limiting the reuse of their memory is not an acceptable option. The classic ABA solution (used in the LIFO-list Pop example) can be excluded, as it will limit arbitrary reuse of memory. Epoch-based solutions should also be excluded because they might get extremely complex under arbitrary termination. But then, the remaining solutions—reference counting and hazard pointers—are not acceptable options either. They depend on preventing the records from being reinserted in the same structures (which is a requirement) as long as there are active references to them. There is no acceptable solution.

FIGURE 2 **Data Structure Requirements**

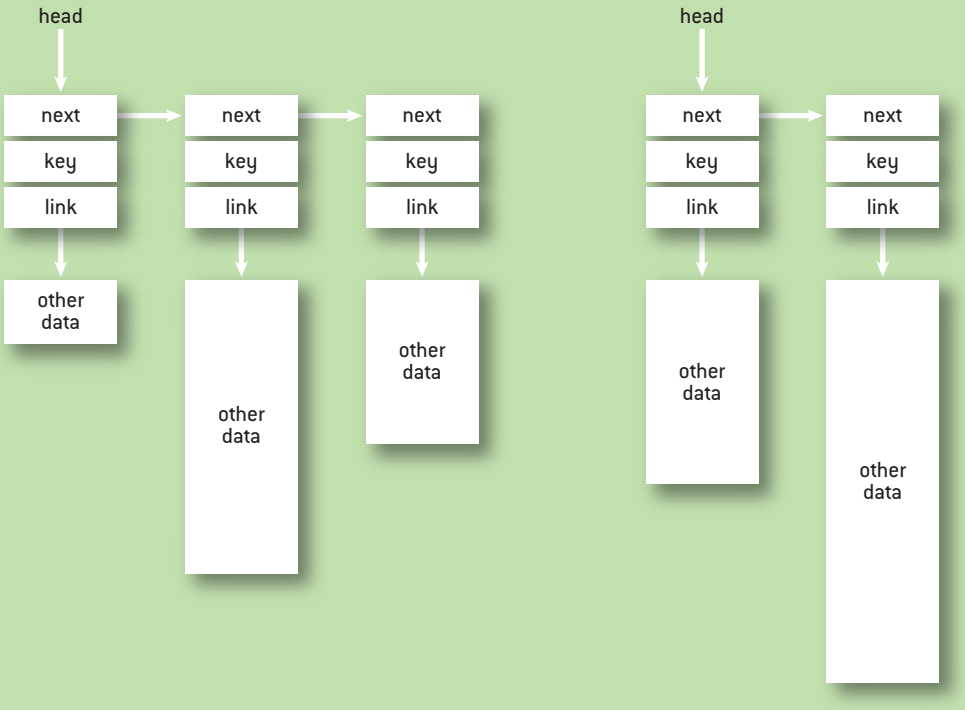


This may be a good time to consider a compromise. How about adding a separate descriptor to each record? The descriptor holds the key value and any fields needed for traversing the data structures; the fields then may be changed in place without removing and adding the records to the structure. With this compromise the memory of the full records can still be freed, but it might be acceptable to keep descriptors from being freed for arbitrary reuse.

Now let's reconsider the ABA problem. Maybe the classic packed-integer solution can work. This has the advantage of allowing the descriptors (and the associated records) to be moved freely among data structures, but it adds a dependence on hardware support for wide atomic operations (e.g., 128-bit CAS). The other options (hazard pointers and reference counting) do not require wide atomic operations but will require the use of a fresh descriptor (i.e., one that does not have any old references lingering from prior operations) on every pair of moves of the records between data structures. Furthermore, both hazard pointers and reference counting add overhead (memory ordering and extra atomic operations, respectively). Finally, to deal with the actual allocation and deallocation of the variable-size records, the designer can use a lock-free allocation algorithm with coalescing and the ability to return free memory to the operating system.[17]

Now the designer is left with the task of balancing the pros and cons of sequential performance (i.e., when using packed integers with double-width primitives) and portability (when using single-width primitives but with extra ordering for hazard pointers or extra atomic operations for reference counting).

FIGURE 3 **Design Compromise**



As the reader can see, this case study goes through a few iterations over the various choices, starting with what is absolutely necessary, making a compromise (using descriptors and adding a level of indirection), and finally reaching a reasonable set of options for further consideration of their pros and cons.

SUMMARY

This article has presented the key issues that can help designers of nonblocking systems make oft-needed compromises and balance tradeoffs to reach a feasible design that satisfies all the requirements. In considering the totality of these issues, designers should go through a few passes on this list of features and issues. First, they should identify and separate the features that are absolutely necessary from those that are desirable but open to compromise. After that, they can start to consider the implications of using these various features.

ACKNOWLEDGMENTS

The author thanks Samy Al Bahra and Paul McKenney for their insightful comments on earlier versions of this article.

REFERENCES

1. Attiya, H., Hendler, D. 2005. Time and space lower bounds for implementations using k-CAS. In *Proceedings of the 19th International Conference on Distributed Computing*: 169-183.
2. Brown, P. J., Smith, R. M. 1973. U.S. Patent 3,886,525. Shared data controlled by a plurality of users (filed June 1973).
3. Cain, H. W., Frey, B., Williams, D., Michael, M. M., May, C., Le, H. 2013. Robust architectural support for transactional memory in the Power architecture. ACM/IEEE 40th International Symposium on Computer Architecture (ISCA).
4. Dragojevic, A., Herlihy, M., Lev, Y., Moir, M. 2011. On the power of hardware transactional memory to simplify memory management. 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC): 99-108.
5. Fatourou, P., Kallimanis, N. D. 2011. A highly efficient wait-free universal construction. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA): 325-334.
6. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W. N., Shavit, N. 2005. A lazy concurrent list-based set algorithm. Ninth International Conference on Principles of Distributed Systems (OPODIS): 3-16.
7. Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124-149.
8. Herlihy, M., Luchangco, V., Martin, P. A., Moir, M. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems* 23(2): 146-196.
9. Herlihy, M., Luchangco, V., Moir, M. 2003. Obstruction-free synchronization: double-ended queues as an example. 23rd International Conference on Distributed Computing Systems (ICDCS): 522-529.
10. IBM System/370 Principles of Operation. 1975. GA22-7000-4.
11. Intel Architecture Instruction Set Extensions Programming Reference. 2012.
12. Jacobi, C., Slegel, T. J., Greiner, D. F. 2012. Transactional memory architecture and

implementation for IBM System Z. 45th Annual IEEE/ACM International Symposium on Microarchitecture: 25-36.
13. McKenney, P. 2013. Structured deferral: synchronization via procrastination. *ACM Queue*. http://queue.acm.org/detail.cfm?id=2488549
14. McKenney, P. E., Slingwine, J. D. 1998. Read-copy update: using execution history to solve concurrency problems. Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems.
15. Michael, M. M., Scott, M. L. 1996. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing: 267-275.
16. Michael, M. M. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel Distributed Systems* 15(6): 491-504.
17. Michael, M. M. 2004. Scalable lock-free dynamic memory allocation. In *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*.
18. Michael, M. M. 2004. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing*: 144-158.
19. Timnat, S., Braginsky, A., Kogan, A., Petrank, E. 2012. Wait-free linked-lists. 16th International Conference on Principles of Distributed Systems: 330-344.
20. Valois, J. D. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*: 214-222.
21. Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., Michael, M. M. 2012. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*: 127-136.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MAGED M. MICHAEL is a research staff member at the IBM Thomas J. Watson Research Center. He received a Ph.D. degree in computer science from the University of Rochester. His research areas include concurrent algorithms, nonblocking synchronization, transactional memory, multicore systems, and concurrent memory management. He is an ACM Distinguished Scientist and a member of the Connecticut Academy of Science and Engineering.