acmqueue

Proving the Correctness of Nonblocking Data Structures

Nonblocking synchronization can yield astonishing results in terms of scalability and realtime response, but at the expense of verification state space.

Mathieu Desnoyers, EfficiOS

So you've decided to use a nonblocking data structure, and now you need to be certain of its correctness. How can this be achieved?

When a multithreaded program is too slow because of a frequently acquired mutex, the programmer's typical reaction is to question whether this mutual exclusion is indeed required. This doubt becomes even more pronounced if the mutex protects accesses to only a single variable performed using a single instruction at every site. Removing synchronization improves performance, but can it be done without impairing program correctness?

Whether this feat can be achieved—and whether it can be extended to algorithms involving more complex data structures—depends on the relationship of the variable to the rest of the program. It also depends on the compiler, architecture, and operating system details, as well as other interesting aspects discussed throughout this article.

Nonblocking data structures[27] can be used to communicate between threads without using mutual exclusion or other synchronization that would otherwise make a thread block awaiting another thread. This article looks at what makes nonblocking data structure design and implementation tricky, and it surveys modeling techniques and verification tools that can provide valuable feedback on the correctness of those algorithms.

WHAT MAKES NONBLOCKING DATA STRUCTURE PROGRAMMING TRICKY?

There are many aspects to consider when programming nonblocking data structures, including the language and architecture memory models, atomicity, ordering, linearizability, and performance.

MEMORY MODELS

Unless the programmer provides explicit key words or synchronization hints, programming languages such as C, C++, and Java presume that a single thread performs variable accesses, leaving the behavior of nonsynchronized concurrent data accesses on multiprocessor systems either undefined or not well understood by programmers. With multicore and multiprocessor computers becoming pervasive, however, it is important to allow concurrent execution. One of the usual ways of providing consistency in concurrent systems is through the use of critical sections and mutual exclusion to ensure serializability,[5] which ends up creating regions of sequential code by excluding other execution threads from accessing critical sections concurrently. Unfortunately, this approach does not result in the best performance in many cases, especially when scalability to many cores is considered. Relaxing sequential execution by shrinking the duration of critical sections, however, increases complexity.

People have commonly used the `volatile` keyword in C and C++[24] to indicate that a variable

can be modified outside of its current scope to disable optimizations that may interfere with the correctness of the program. This keyword, however, tells the compiler only to assume that the variable could be modified outside of the local thread and that order among volatile accesses within a single thread needs to be preserved; it does nothing to prevent reordering by the processor. The ordering guarantees of the `volatile` keyword vary greatly from language to language, and even between language versions: for example, the `volatile` keyword in Java has a much stricter memory-ordering semantic starting from JDK 1.5 than it had in JDK 1.4.[15]

ATOMICITY

Another possible problem with nonblocking data structure programming is that an instruction executing atomically on a processor is not sufficient to ensure that its effect is made visible to other processors atomically.

Unaligned word-sized memory accesses are a good example: many architectures will allow those to be performed by a single instruction, but there is no guarantee that the in-memory result will be updated atomically.

Another example is a nonatomic read-modify-write operation. Although some architectures might end up turning the C `i++` statement into a single instruction, the compiler can very well choose to perform this in three separate instructions: load from memory to register; increment register; store register to memory. The compiler may choose to do so either because it is required by the instruction set (e.g., RISC) or simply because register pressure is too high. Moreover, with the Intel x86 instruction set, for example, variables meant to be read, modified, and written atomically by many processors running concurrently must have a LOCK prefix.[23] Unless special double compare-and-swap or transactional memory instructions are being used, if supported by the architecture, memory accesses that need to touch more than one word-aligned word-sized data structure must be performed in many instructions, and thus nonatomically.

Finally, the compiler is allowed to refetch variables from memory. Therefore, what someone might think will always be performed in a single load operation might not be.

REORDERING

Reordering can be performed at many levels for performance reasons. First, many processors can reorder loads and stores. In addition, processors can reorder execution of instructions that don't have interdependencies. Finally, compilers can reorder expression evaluation, statements, and instructions as long as program order is preserved. Unfortunately, these reorderings do not take into account that threads executing concurrently may assume that operations performed by other threads will appear in program order from their own points of view. This is why processors provide memory-barrier instructions and compilers provide compiler barriers. These operations limit reordering across the barriers in the instruction and code flows.

It is important to understand that atomicity of a memory access does not necessarily provide ordering. In some architectures such as x86, the LOCK prefix, used to specify atomic operations, implies a memory barrier. A great many other architectures, however, such as PowerPC,[22] ARM,[2] and MIPS,[28] perform their atomic operations with LL/SC (load-linked/store-conditional) instructions and usually require explicit memory-barrier instructions to provide ordering.

LINEARIZABILITY

Atomicity and ordering are not necessarily enough to ensure that an entire nonblocking data structure will behave in the same way as one that is always accessed sequentially. Nonblocking operations normally contain a linearization point[20] to guarantee correctness with respect to the sequential definition of that operation. Linearization points are the atomic operations that will perform the mutations necessary to provide the correct externally observable effects with respect to the sequential specification of the data structure simultaneously with validation of operation success or failure. This ensures that no globally visible inconsistent operation state lingers when an update operation aborts. This also ensures that the behavior of the data structure as a whole matches the behavior expected from using the data structure sequentially. It should be understood that reasoning in terms of linearization points has some limitations. For example, it does not consider delay between invocation of a method and execution of its linearization point.[18,25]

The following is a good example of a linearization issue: suppose you have a queue such as the concurrent queue with wait-free enqueue/blocking dequeue (http://lttng.org/urcu). This enqueues nodes at the tail and dequeues from the head of the queue. It provides nonblocking (wait-free) enqueue by requiring threads performing dequeue, splice, and iterations to busy-wait if they find a NULL next pointer in a node that is not the tail of the queue. The dequeue operation dequeues one node at a time, whereas the splice operation moves all nodes from a source queue into a destination queue. To illustrate enqueue and splice operations, figures 1, 2, 3, 4, 5, and 6 represent queue nodes as boxes. Within these boxes, circles represent pointers to the next node. A gray circle is a NULL pointer. Solid arrows represent the target of a pointer, and dashed arrows represent the previous pointer target. Circles containing a number represent the order in which updates are stored in memory.

The empty queue is shown in figure 1. An enqueue operation is performed in two steps, shown in figures 2 and 3. The first step moves the tail pointer to the next node using an atomic exchange operation. It leads to a transient state during which threads concurrently performing a dequeue,

FIGURE 1 **Empty Wait-Free Concurrent Queue**

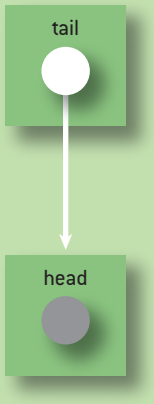


FIGURE 2 **Enqueue First Node Transient State**

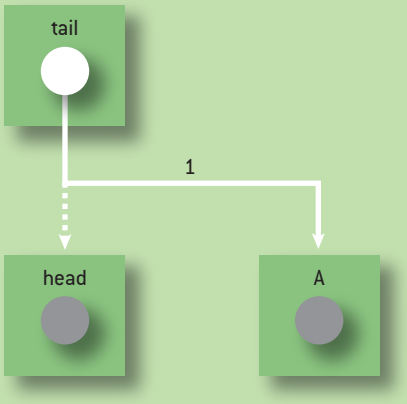


FIGURE 3

First Node Enqueue Completed

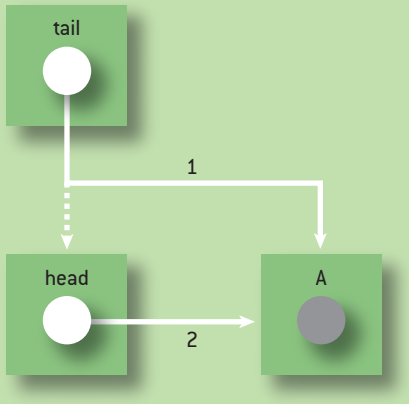
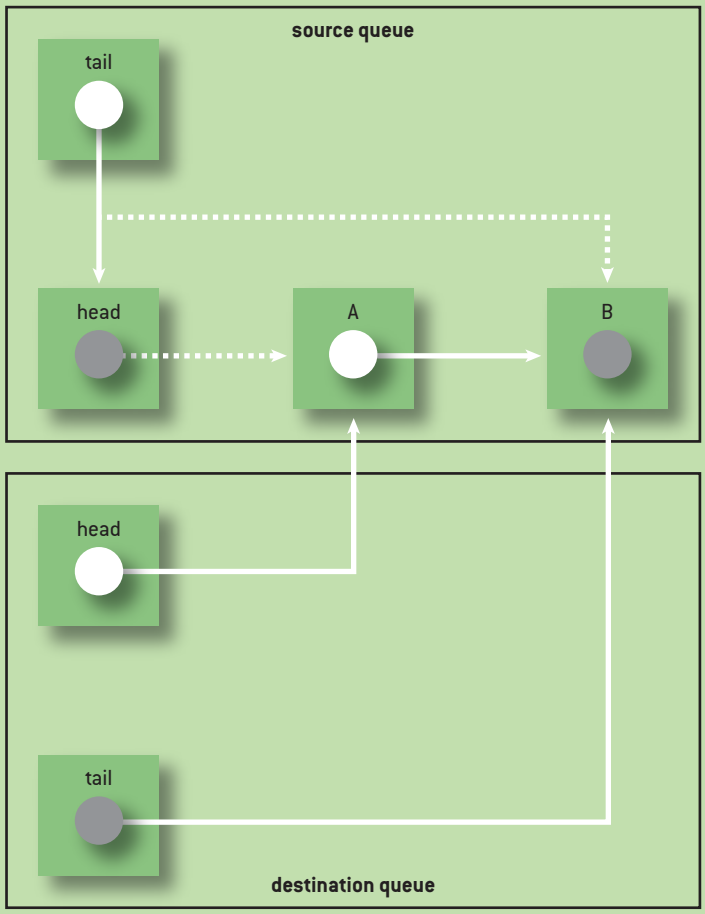


FIGURE 4

Splice Completed



splice, or iteration need to busy-wait. Enqueuing is completed by storing the new node's address into the last node's next pointer.

Performing one more enqueue operation adds a new node *B* at the tail, so you end up with a queue that has two nodes, *A* and *B*, leading to the initial state of the source queue presented in figure 4. The splice operation shown in this figure moves all nodes from the source queue into the destination queue.

Note that an arbitrary number of operations can occur while the queue is in a transient state. This would allow, for example, the enqueue operation of a second node to complete while the first node is not yet completely enqueued, as shown in figure 5. When encountering this queue structure, both

FIGURE 5

Enqueue Second Node Over First-Node-Enqueue Transient State

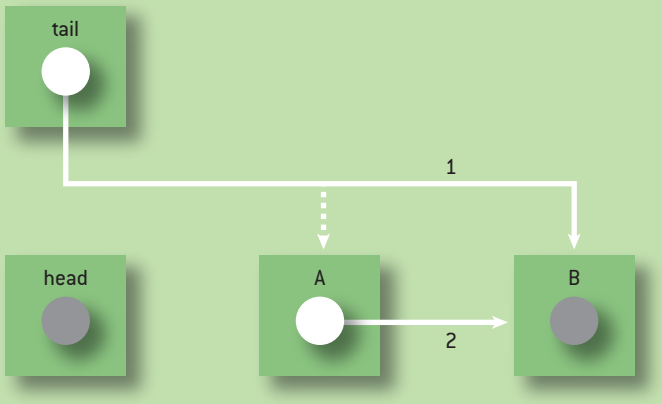
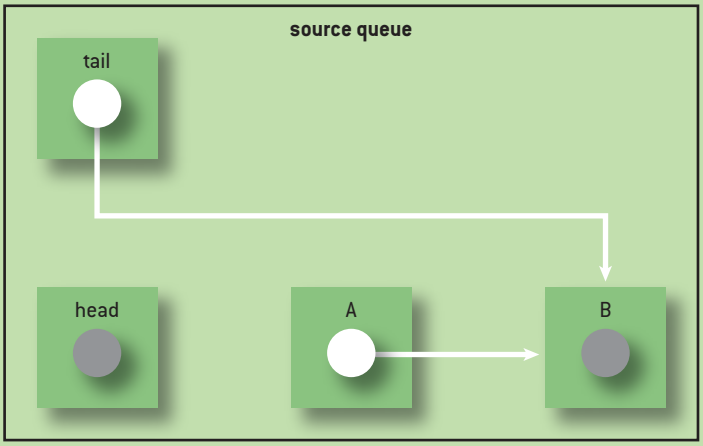


FIGURE 6

Splice Waiting For First-Node-Enqueue Completion



splice and dequeue operations would need to busy-wait until the first enqueue completes.

Queues in a transient state, as illustrated in figure 6, raise the possibility of an interesting optimization to the splice operation: instead of busy-waiting when encountering the first node with a NULL next pointer in a non-empty queue, one could simply assume that the queue is empty. After all, if the splice operation is called again after a short time, then it would eventually get the queue content. Unfortunately, even though this approach might seem appropriate at first glance, it would break the linearizability of the queue. Consider the following scenario: one thread *X* is enqueuing node *A*, and another thread *Y* is first enqueuing *B*, then performing a splice operation to grab the entire queue. Given a program that has only those two threads and no other dequeuer thread, thread *Y* should be allowed to expect that if it performs an enqueue operation followed by a splice, the splice will never encounter an empty queue because it *must* contain the element added by the previous enqueue. If thread *X* is preempted while adding the first element into the queue, however, thread *Y*'s splice operation could encounter a queue that would appear empty, even though its sequential specification should allow it to assume the queue contains at least one node.

PERFORMANCE CONSIDERATIONS

Other elements that make nonblocking data structure programming hard are the considerations of throughput and scalability associated with atomic accesses and memory barriers. Depending on the design and use of the data structure, it might be cheaper, performance-wise, to hold a lock and perform the operation using a sequence of regular nonatomic instructions than to perform a sequence of more expensive atomic operations with memory barriers of their own. Therefore, performance considerations are largely driving the careful choice of atomic instructions and barriers to implement higher-level operations, thus increasing complexity.

Nonblocking data structures have many interesting properties such as reentrancy and mutual-exclusion deadlock immunity, as well as, in some cases, good scalability and throughput. Implementing them efficiently, however, involves understanding interactions with interruptions and traps (at kernel level), signals (at userspace level), multithreading, scheduler preemption, and thread migration, in addition to the low-level compiler and processor aspects.

MODELS

As shown in the previous section's discussion about linearizability, providing counterexamples is a great way of illustrating problems in nonblocking data structure design and implementation. Designers of nonblocking algorithms should try to find representations that will provide a deeper understanding of the algorithm, helping them to consider as many race scenarios as possible. These representations will not only enhance the designers' understanding of detailed interactions between threads, but will also help in expressing their ideas to others.

Which brings us to code review: encouraging many people to think of different ways in which an algorithm could misbehave, each with his or her own focus and expertise, will likely shed more light on the problem than having just one person look at it from a single point of view. This is a useful trick even for an individual reviewing a nonblocking algorithm: looking at an algorithm over and over, at different moments in the day, in all sorts of contexts, with a plethora of models to represent the algorithm, will help achieve a thorough study from various viewpoints.

This calls for representing algorithms in various ways. Diagrams are a great way of showing the

relationships between various objects in a data structure, dependency between memory operations, and the various state transitions that an object can go through.

The concurrent queue with wait-free enqueue/blocking dequeue presented in the previous section illustrates how to use diagrams to represent nonblocking data structures. Previous work has provided examples of the states of a data structure represented as diagrams: in the RCU (read-copy-update) linked list and grace period explanation;[12] showing the various states in which a hash table can be found in Cliff Click's hash table explanation;[8] and showing instruction dependencies at the processor level.[10,11]

Many optimizing compilers internally use models to represent dependencies between statements. It helps them move statements around and carry optimizations without changing the behavior as seen from program order. Some of these representations, to name a few, are DFG (data flow graph), which represents the data dependencies between statements (for example, memory accesses or register accesses); CFG (control flow graph), which represents the control dependencies between statements (for example, branches or loops); and a convenient combination of those two, PDG (program dependence graph).[14] Modeling algorithms at the PDG level can be useful for verification, as will be shown in the Testing section.

Representing algorithms as sequences of statements in a programming language is another representation that allows thinking about the code in a more sequential manner. However, it should not be assumed that the code executes sequentially. This representation is merely a starting point for considering all possible reorderings that could be performed. As silly as it might sound, writing the code on a computer, with pencil and paper, with or without thorough commenting on the possible reordering at each line, as an initial draft, or recopying from memory are all different ways of interacting with the code that can lead to a better understanding of the reasons why the code is written a certain way.

A convenient way of presenting counterexamples is to demonstrate race scenarios by side-by-side execution sequences of two or more processors or threads. Initial variable states, valid within the specifications of the algorithm, are first detailed, and then portions of the algorithm are examined. This can be achieved by detailing, in program order, execution sequences for each processor or thread involved. Then the order of one or two statements can be altered within the constraints allowed by the compiler and processor. For each modified execution scenario, all invariants imposed by algorithm correctness should be respected. If it is possible to find a scenario that breaks those invariants, then it should be treated as a bug. It might be caused by an algorithm design issue, by missing memory barriers, or by incorrect assumptions about the atomicity of a sequence of operations.

For smaller code snippets, moving to the instruction-level scope to represent key pieces of the algorithm can be worthwhile. This is especially useful when considering memory reordering performed by the processor. It gets the compiler out of the way and lets the reviewer focus on instruction and processor semantics—at the expense of a less compact model.

The issue with nonblocking algorithms is that it is not sufficient to consider the sequentially equivalent high-level algorithm operations as happening one after another: after each instruction within the algorithm, one needs to consider what happens if other processors execute any concurrent operation of the algorithm a possibly infinite number of times. Moreover, when there are no compiler- or processor-level memory barriers in place, every reordering allowed within the

specification of the architecture needs to be taken into account. This quickly increases the number of concurrent execution flows to consider, which explains why making sure no execution flow can misbehave is hard. When faced with a large number of states to validate, model checkers, presented in the Testing section, can be very helpful in automating tedious and error-prone verification.

Every assumption made in a concurrent piece of code should be revisited with prejudice. It is important to assure that these assumptions will hold across hardware memory models, programming language memory models, and higher-level correctness constraints (such as linearizability). For each assumption made, many attempts should be made to come up with a race, as far-fetched as it may be, that can make the algorithm misbehave.

MODEL ACCURACY

To come up with an accurate model of a nonblocking algorithm, the memory models of all targeted architectures need to be taken into account. At first glance, it might appear that nonblocking algorithms should be specifically tied to a single architecture, but there are methods to model nonblocking algorithms in a way that allows reasoning about their correctness across a large set of architectures.

To model an algorithm in a way that is portable across multiple architectures, you can think of the algorithm as running on a model consisting of all the worst reordering possibilities that can be performed by the set of architectures. Memory models targeting a set of architectures can be found in the Linux kernel (http://www.kernel.org), the Userspace RCU project (http://lttng.org/urcu), and within the Concurrency Kit (http://concurrencykit.org).

The algorithm models should therefore consist of the worst-case reordering that could happen by combining the characteristics of all architectures targeted. Of course, this increases the state space to validate, but the benefit is in having a single model of an algorithm that works on a wide range of architectures. Proving that a single targeted architecture can fail is sufficient to identify an error.

When targeting many architectures, stress testing under all supported architectures is mandatory, because some specific reordering characteristics that would cause the algorithm to misbehave could be specific to only one architecture within the set.

Targeting architectures with weaker memory models will increase the size of the model's state space, because many more reorderings can occur. On the other hand, if architectures with weak memory consistency models are covered, then other architectures with stricter memory consistency models—in every way equally or more strict than previously validated models—will be a given.

This demonstrates a limitation of models: their completeness is unfortunately limited by how accurately they represent architecture behavior. Techniques such as litmus tests can help improve the accuracy of models. Those will be discussed in the Formal Methods section.

TESTING

Notwithstanding the amount of care taken while designing and implementing a nonblocking algorithm, there is always the chance that some characteristics of the processor, compiler, or operating system have gone unnoticed. Therefore, no modeling or review can replace the good old testing approach. Moreover, testing is very effective in discovering issues when porting to an unforeseen architecture.

When nonblocking algorithms are being tested, the testing coverage needs to be slightly adapted,

compared with its usual definition. Indeed, when a simple sequential algorithm is being tested, covering a large percentage of lines and branches can be a good indicator that tests were thoroughly performed. Unfortunately, this is not sufficient when testing algorithms that can execute in parallel. Testing coverage must check not only that every line of code and branch has been executed, but also that each one was executed in every context of other threads.

This means that it might take a long time for an error to trigger in production if it is caused by a small race window in which two sequences of instructions misbehave when executed concurrently. Therefore, moderate testing can be sufficient for making sure that the common cases work fine, but making sure that corner cases don't misbehave can be quite challenging.

One way of tackling this issue is stress testing. When an error condition takes time to reproduce, focusing on triggering the race over and over for a long period of time can increase the chance of reproducing it more quickly. The main aspect that can be controlled is the frequency at which the race window is executed. Therefore, as the runtime length of the algorithm to stress test increases, the time ratio spent executing each individual race window diminishes.

To accelerate this process, you can use what could be called "oriented stress testing." This entails changing the configuration of the test bench while doing the stress test, based on an understanding of the design, so corner cases are more likely to be hit frequently. For example, to stress test a hash table, keeping a number of buckets purposefully low and testing with thousands of nodes with the same key are both likely to generate long hash chains. This corner case might be hard to trigger with a large hash table, even after weeks of testing, without oriented stress testing. Besides configuration changes, some tools can help automate oriented testing: CONCURRIT can help specify a deterministic execution order that should be enforced by testing runs;[6] the Relacy tool (http://www.1024cores.net/home/relacy-race-detector) allows controlling the interleaving of atomic operations between threads.

Another trick for stress testing is to add random delays within the algorithm, so different execution timings are tested. This can make race windows slightly larger, which can help hit issues more quickly.

There has been a fair amount of research on deterministic multithreading,[4,29] which consists of deterministically fixing the order of critical sections with a scheduler. Regarding correctness, the objective is to obtain repeatable results across runs, thus making bugs easier to reproduce in testing. Even though this leads to acceptable results in some cases, it remains to be seen whether this approach will scale to fine-grained locking and larger multiprocessor systems, while having a low overhead. Limitations of this approach include its requirement for concurrent accesses to use locks, which are more expensive than RCU and nonblocking synchronization; the increase of state-space size needed to track the patterns with the frequency of lock acquisition; the extra communication overhead required at lock acquisition; and the fact that small changes to the source code can significantly change lock access patterns.

When scalability, performance, and realtime response matter, stress testing can yield better results than deterministic multithreading, by using a statistical approach to testing coverage rather than fixing the layout of lock acquisition at runtime.

MODEL CHECKING

Wouldn't it be nice if the time-consuming exercise of validating a nonblocking algorithm by manually coming up with counterexamples based on all states that can be reached by concurrent

threads for any given state of a thread, and statistically through testing, could be automated? Fortunately, there are approaches that allow this, to some extent.

FORMAL METHODS

Model checking undertakes a full state-space search to verify specific assertions of a given model.[3,7] While very powerful, this approach has important limitations. For one, the amount of memory and time required to perform the full state-space search grows very quickly with the size of the state space. Therefore, this approach applies only to relatively small models. This is why keeping nonblocking algorithms very compact in terms of state space is important: it allows easier thorough state-space search. In order to ensure this, nonblocking algorithms should be designed in ways that will keep state space as small as possible.

One way to leverage model checking for nonblocking algorithms is to design algorithms as small building blocks, none of which exceeds the state-space search capacity of current computers. Each algorithm can then provide memory-ordering guarantees through an API, and can then be assumed correct by another nonblocking algorithm that would use it. Therefore, it is possible to compose nonblocking algorithms into quite complex algorithms by cutting the state-space search at their interfaces.

Another limitation of model checking is that the verification is only as accurate as the model and assertions. If the model is too simple compared with the reordering that can be performed by the processor, some errors won't be caught by the checker. Moreover, if assertions don't represent what is intended to be verified, the checker may never find a bug that would be assumed to be caught. This is because model checkers work a little bit like oracles: when everything is fine, they just report that everything went fine. It's only when an assertion fails that they provide a detailed sequence of events that led to the issue.

One way of limiting the risk of modeling errors is error injection into the model. If it is assumed that a given model and assertion should catch one type of error, a slightly altered model that triggers the assertion should be created, just to make sure that it catches the error if it is added purposefully.

Model checkers that can be used for nonblocking algorithms include Spin,[21] which allows describing a model in Promela and verifying LTL (linear temporal logic) assertions on that model. The properties to validate can be as simple as an assertion in the code, but the real power of LTL is that it provides temporal operators. It is then possible to validate that a certain state is never reached until another state is reached, for all possible executions.

Other model checkers focus on reproducing architecture behavior at the instruction level. Previous articles have reported on the results of litmus tests on a wide range of architectures to characterize their respective behavior and to formalize their memory models.[1] The authors then created a tool that allows running ARM and Power litmus tests in a verifier (http://www.cl.cam.ac.uk/~pes20/ppcmem/).[30]

The important question regarding model checking of nonblocking algorithms is: What should be modeled? In previous research, I proposed a model for compiler, memory, and processor reordering.[10] Its purpose was to allow expressing algorithms in a model that takes into account PDG dependencies at the code and instruction levels. Depending on the accuracy level needed, the complexity of the algorithm, and the resources available to run the model checker, a different degree of detail can be either included or omitted from the models.

EXECUTION-BASED MODEL CHECKING

Driving the model checker by executing the program to verify is another model-checking approach, known as EMC (execution-based model checking).[13]

Rather than traversing all states required to prove formally that a condition is never false, driving the verification with the states reached by execution of the verified system can significantly limit the number of states to explore, at the expense of completeness of the proof. Therefore, if the execution of the program causes the verified condition to fail, it will be reported. Absence of error does not guarantee validity of the program, however, since it may contain errors in states not reached by its execution.

A good EMC example is found in the lockdep verifier[9] implemented within the Linux kernel. Perhaps its most important verification is performed by constructing a graph of encountered lock dependency chains, and reporting errors if lock usage could trigger a deadlock. It is worth mentioning that lockdep can be adapted to validate locking within arbitrary applications and is therefore within the grasp of any programmer.

EMC can be performed either directly alongside execution of the program or based on a trace of the program execution. This execution trace can act as a container storing the events driving an off-line model checker.

FORMAL VERIFICATION THROUGH MATHEMATICAL PROOF

Formal mathematical proof is another way of verifying that an algorithm does not contain unwanted characteristics. The idea is to start with a theorem to prove. This typically assumes that some invariant is always true. Then the proof is obtained through induction based on ordering constraints enforced by the compiler and processor memory models, as well as the algorithm. Examples of formal mathematical proofs can be found in the literature,[12,17] and other authors have also presented interesting proofs on algorithm progress and correctness.[16,19,26]

CONCLUSIONS

When it comes to nonblocking algorithms and data structures, we are balancing different goals. On one side, compiler writers and chip designers want to exploit all the freedom allowed by any unspecified behavior. On the other, designers and developers of parallel algorithms want precisely defined behavior when it comes to dependency ordering.

Nonblocking synchronization can yield astonishing results in terms of scalability and realtime response, but it comes at the expense of verification state space. Verifying algorithms on multiple architectures, each with its own memory model, also increases state-space size. Once the weakest ordering constraints are modeled, however, new architectures that fit within those constraints can be added with no extra verification cost, other than testing to ensure that the memory model is well defined.

There are various ways to increase confidence in a nonblocking algorithm, including oriented stress testing, formal and execution-based model checking, mathematical proofs, and code review. Given that the state space to explore increases quickly with the complexity of a nonblocking algorithm, designing small algorithms with full state-space verification in mind helps full state-space verification. Those can then be composed into more complex algorithms.

As programming languages improve their awareness of concurrency, it will be interesting to see the continuing advance in modeling and verification of concurrency at the compiler level and by static-code analyzers.

ACKNOWLEDGMENTS

Thanks to Paul E. McKenney for his work on RCU and nonblocking synchronization, David Miller for his work on Linux kernel atomic operations, David Howells for his work on Linux kernel atomic operations and memory barriers, and Hugh Dickins for his work on type-safe memory. Thanks also to the Linux kernel community for their work and feedback on nonblocking synchronization, memory models, and lockdep, in particular Lai Jiangshan, Ingo Molnar, Peter Zijlstra, and Arjan van de Ven. I am indebted to Paul E. McKenney, Samy Bahra, Christian Babeux, Jeremie Galarneau, and Michel R. Dagenais for reviewing this article.

LEGAL STATEMENT

REFERENCES

1. Alglave, J., Maranget, L., Sarkar, S., Sewell, P. 2011. Litmus: running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*: 41-44; http://dl.acm.org/citation.cfm?id=1987389.1987395.
2. ARM. 2010. *ARM Architecture Reference Manual*; http://infocenter.arm.com/help/ index.jsp?topic=/com.arm.doc.set.architecture/index.html.
3. Baier, C., Katoen, J. 2008. *Principles of Model Checking*. Cambridge: MIT Press; http://books.google.ca/books?id=nDQiAQAAIAAJ.
4. Bergan, T., Anderson, O., Devietti, J., Ceze, L., Grossman, D. 2010. Coredet: a compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News* 38(1): 53-64; http://doi.acm.org/10.1145/ 1735970.1736029.
5. Bernstein, P., Shipman, D., Wong, W. 1979. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering* 5(3): 203-216.
6. Burnim, J., Elmas, T., Necula, G., Sen, K. 2012. CONCURRIT: testing concurrent programs with programmable state-space exploration. In *Proceedings of the 4th Usenix Conference on Hot Topics in Parallelism*: 16-16; http://dl.acm.org/citation.cfm?id=2342788.2342804.
7. Clarke, E., Grumberg, O., Peled, D. 1999. *Model checking*. Cambridge: MIT Press; http://books.google.ca/books?id=Nmc4wEaLXFEC.
8. Click, C. 2007. A lock-free hash table. JavaOne Conference.
9. Corbet, J. 2006. The kernel lock validator. LWN; http://lwn.net/Articles/185666/.
10. Desnoyers, M. 2009. Low-impact operating system tracing. Ph.D. dissertation. Ecole Polytechnique de Montreal; http://www.lttng. org/pub/thesis/desnoyers-dissertation-2009-12.pdf.
11. Desnoyers, M., McKenney, P. E., Dagenais, M. R. Forthcoming. Multicore systems modeling for formal verification of parallel algorithms. *Operating Systems Review*.

12. Desnoyers, M., McKenney, P. E., Stern, A. S., Dagenais, M. R., Walpole, J. 2012. User-level implementations of read-copy-update. *IEEE Transactions on Parallel and Distributed Systems* 23(2): 375-382.
13. Drusinsky, D. 2011. *Modeling and Verification Using UML Statecharts*. Elsevier Science; http://books.google.ca/books?id=JMz-SWTfgiAC.
14. Ferrante, J., Ottenstein, K. J., Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9(3): 319-349; http://doi.acm.org/10.1145/24039.24041.
15. Gosling, J., Joy, B., Steele, G. L., Jr., Bracha, G., Buckley, A. 2013. *The Java Language Specification, Java SE 7 Edition*. Pearson Education; http://books.google.ca/books?id=2RYN9exiTnYC.
16. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V. 2009. Proving that nonblocking algorithms don't block. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*: 16-28; http: //doi.acm.org/10.1145/1480881.1480886.
17. Gotsman, A., Rinetzky, N., Yang, H. 2013. Verifying concurrent memory reclamation algorithms with grace. In *European Symposium on Programming*. Rome, Italy: Springer.
18. Haas, A., Kirsch, C. M., Lippautz, M., Payer, H. 2012. How FIFO is your concurrent FIFO queue? In *Proceedings of the Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*.
19. Herlihy, M. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 13(1): 124-149; http://doi.acm.org/10.1145/114005.102808.
20. Herlihy, M. P., Wing, J. M. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12(3): 463-492; http://doi.acm.org/10.1145/78969.78972.
21. Holzmann, G. J. 1997. The model checker Spin. *IEEE Transactions on Software Engineering* 23(5): 279-295.
22. IBM. 2010. Power ISA Version 2.06 Revision B; http://www.power.org/ resources/reading/.
23. Intel Corporation. 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z*; http://download.intel.com/products/processor/manual/325383.pdf.
24. International Organization for Standardization. 2011. Programming languages - C++, ISO/IEC 14882:2011.
25. Kirsch, C. M., Lippautz, M., Payer, H. 2012. Fast and scalable k-fifo queues. University of Salzburg, Salzburg, Austria. Technical Report 2012-04.
26. Michael, M. M. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15(6): 491-504; http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1291819&queryText%3Dhazard+pointers.
27. Michael, M. M., Scott, M. L. 1996. Simple, fast, and practical nonblocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*: 267-275; http: //doi.acm.org/10.1145/248052.248106.
28. MIPS Technologies Inc. 2012. *MIPS Architecture for Programmers, Volume II: The MIPS64 Instruction Set*.
29. Olszewski, M., Ansel, J., Amarasinghe, S. 2009. Kendo: efficient deterministic multithreading in software. *ACM SIGPLAN Notices* 44(3): 97-108; http://dl.acm.org/citation.cfm?id=1508256.
30. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D. 2011. Understanding Power multiprocessors. ACM *SIGPLAN Notices* 46(6): 175-186; http://doi.acm.org/10.1145/1993316.1993520.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MATHIEU DESNOYERS is president and founder of EfficiOS. He maintains the LTTng project and the Userspace RCU library. His research interests are in performance analysis tools, operating systems, scalability, and realtime concerns. He holds a Ph.D. degree in computer engineering from Ecole Polytechnique de Montreal (2010).