Theories and Tools for Safe Concurrent Data Structures

Warren A. Hunt Jr.[*] and Ian Wehrman[†]

1 Overview

With the decline of single-threaded processor performance and the proliferation of multi-core computers, the importance of parallelism is manifest. Its exploitation is a practical necessity for the performance and scalability of large-scale critical systems. But even small sequential parts of otherwise-concurrent programs significantly limit possible speedups from parallelism. Concurrency must therefore be found in even the most difficult corners of programs, typically involving interaction among threads across cores via shared data structures.

Such pervasive concurrency is provided by *concurrent data structures*: complex and often racy implementations of abstract data structures that carefully omit locks and memory fences to maximize throughput. Concurrent data structures are widely used in operating systems, embedded and real-time systems, and high-performance web, database and application servers—all of which are routinely deployed for the storage, manipulation and transmission of sensitive data in scenarios like credit card transaction processing and electronic securities trading. Correctness of the underlying data structures is manifestly necessary for the overall security of these applications.

A basic security property that such data structures must exhibit for reliable operation is memory safety. Violations of memory safety underlie a large fraction of software security vulnerabilities today. Buffer overflows alone account for half of the thirty most serious vulnerabilities in the US-CERT database as of June 2011. Early detection of such violations—whether dynamically via program or runtime instrumentation, or statically via formal verification or compile-time analyses—is therefore an important issue for secure software construction.

Concurrent data structures have other security implications as well. Because concurrent data structures are typically designed to ensure that misbehaving threads cannot interfere with the others' progress, they can help avoid *priority inversion*—in which a high-priority thread is blocked by one with low priority—which should not be allowed in a secure system. Concurrent data structures are also an important component in techniques for monitoring running programs to detect and avoid buffer overflow attacks [8]. The performance overhead from traditional coarse-grained data structures made earlier attempts at such monitoring too costly to be practically useful.

[*]University of Texas at Austin, `hunt@cs.utexas.edu`

[†]University of Texas at Austin, `iwehrman@cs.utexas.edu`

Concurrent data structures are small but subtle; hard to design, implement and debug. Ensuring their correctness, and the safety of programs that use them, is extremely challenging. Concurrent data structures are thus an excellent candidate for formal verification. But existing high-level techniques are technically insufficient—either too conservative or unsound—because they are not founded on a sufficiently accurate *memory model*: a specification of the manner in which programs interact with the memory systems of the architectures on which they execute. To reason soundly about these difficult programs, and to prove them safe, it is necessary to account for the underlying machine's memory model.

2 Concurrent Data Structures

When a data structure in memory is shared among processes, they must synchronize their access to preserve its integrity and correctness. The traditional way to accomplish this is by using locks to ensure that at most one process can manipulate and access the data structure at a time. But this is often an overly conservative policy for mediating shared access to a data structure. It can be safe, e.g., for one process to enqueue an element at once end of a list while another process concurrently dequeues from the other end. Furthermore, when locks are used to guard access to a data structure, one misbehaving process can impede the progress of the rest of the system if it delays or deadlocks after having acquiring the lock.

For these reasons, there is a strong performance incentive to use more sophisticated synchronization methods for shared data structures. These methods may involve more finely grained locking strategies, in which different locks protect different parts of a data structure, allowing distinct processes concurrent access in some situations. In some cases locks can be elided completely in favor of synchronization primitives provided by the hardware, like memory fences or interlocked instructions. If used properly in the design of a data structure, these techniques can significantly improve the granularity of concurrent access and mitigate the delays caused by misbehaving processes. But these synchronization primitives are no panacea. It is considerably more difficult to design a correct data structure using just basic synchronization primitives than with coarse-grained locks. Furthermore, use of the synchronization primitives themselves comes with significant performance overhead: their execution can take hundreds or thousands of times more clock cycles on a modern multiprocessor system compared to unsynchronized instructions.

To mitigate the performance penalty of using either locks or primitive synchronization instructions, they must be used sparingly. But, without sufficient synchronization, correctness of the data structure can easily be sacrificed. Hence, there is a need to reason about these data structures, both to ensure their correctness and to determine whether fewer synchronization points are needed. Such fine-grained data structures often have race conditions, however, and may exhibit non-sequentially consistent behavior, which complicates this reasoning.

3 Our Approach

We have developed a theory for reasoning about the execution of such programs on x86-like machines. Our theory takes into account the memory model of the architecture, which is described using *write buffers*—FIFO queues of writes that connect each processor to the shared memory. When a processor performs a store instruction, a new write is enqueued on its write buffer. The result of a load uses the most recently enqueued write in the processor's write buffer if one exists; otherwise it uses the value in shared memory. Processors may also issue fence instructions, which have the effect of committing all pending writes in the processor's buffer to shared memory. Writes may otherwise commit nondeterministically to memory, but always in the order they were written into the write buffers.

Our theory consists first of a semantic model x86 multiprocessor machine states. Unlike simpler traditional models, our model consists of both a shared memory as well as an array of lists that represent write buffers. The model also incorporates additional structure to allow compositionality, in which partial descriptions of system states are combined to create a more complete system description. Additional structure also describes which addresses in memory are shared among the processors and which are owned (with sole write access) by a single processor.

This model exhibits strong locality properties: the behavior of a programs executed from some partial system state is essentially the same as its behavior when executed from a more complete system state. This provides the theoretical foundation for *local reasoning*, wherein program reasoning need only consider the system resources (i.e., the partial system description) that is actually accessed and manipulated by the program at runtime.

On top of this semantic model for x86 multiprocessors we have built a new logic of assertions, which denote sets of states. Traditional logics use assertions to describe the value of locations in main memory. Our assertion language additionally allows the user to describe the ordering of buffered writes that have not yet been committed to memory, as well as the effect of flushing these buffered writes to memory. For example, we write the assertion $x \rightsquigarrow 1\ ;\ y \rightsquigarrow 2$ to denote a state with two buffered writes, the first to x with value 1 and the second to y with value 2. Because writes may also non-deterministically commit to main memory, this assertion also denotes the state in which the first x write has committed to memory but not the y write, and also the state in which both writes have committed to memory. On the other hand, the assertion $\mathbf{bar}(x \rightsquigarrow 1\ ;\ y \rightsquigarrow 2)$ describes the result of flushing both these writes to memory (as with a fence or *barrier* operation), and so denotes only the state in which both writes have committed to memory.

The assertion logic is then used as part of another logic of program specifications, similar in spirit to Hoare logic or separation logic. This program logic consists of axioms for the primitive commands of the programming language, as well as inference rules for deriving specifications of larger programs. For example, the axiom for the load command,

$$\{e \rightsquigarrow e'\ ;\ P\}\ x := [e]\ \{(e \rightsquigarrow e'\ ;\ P) \wedge x = e'\},$$

indicates that whenever the address e is loaded into local variable x in a state in which its most recent buffered value is given by e' (possibly with some more recent writes to distinct addresses described by the assertion P), the resulting system state will be as before, but with x set to e'. The precondition also asserts that e is an allocated memory address, ensuring memory safety for the load command.

Using a frame inference rule, we may additionally conclude that the presence of earlier buffered writes do not affect the behavior of the load command, e.g.:

$$\{f \rightsquigarrow f' \; ; \; e \rightsquigarrow e' \; ; \; P\} \; x := [e] \; \{(f \rightsquigarrow f' \; ; \; e \rightsquigarrow e' \; ; \; P) \wedge x = e'\}.$$

Additional details about the semantic model, assertion logic and program logic are available in [7, 6].

4 Comparison with Existing Approaches

Our current research is focused on the development of foundational theories for program reasoning that account for realistic memory models in order to enable high-level, modular reasoning about concurrent data structures and similarly difficult programs, as described above. The second author's dissertation work develops such a theory specific to x86-like multiprocessors via a Hoare-style logic for partial-correctness properties [7, 6] like memory safety and other more elaborate specifications. The theory may be used to give rigorous proofs of programs whose safety has previously only been argued informally, or in some cases incorrectly, because most other formal approaches are insufficient.

The traditional approach to program verification is *operational reasoning*, in which a general-purpose logic (e.g., first-order logic or type theory) is used to construct a proof about an encoding of the operational semantics of the program in the logic (see, e.g., [4]). The operational approach is flexible and mature, but is comparatively ad hoc, providing little guidance and requiring significant creativity on the part of the prover. It thus provides a less-clear avenue toward automation of important properties like memory safety. Tool support can significantly aid the construction of an operational proof within a general-purpose logic, but it is not able to construct proofs of non-trivial programs without considerable expert guidance. Consequently, the operational approach requires significant time and expertise to apply, making it prohibitively expensive in practice.

Program logics, on the other hand, provide the means for concise, high-level program proofs. They consist of a language of assertions for describing system states, a language of program specifications (which makes use of the assertion language), and a logic for deriving true program specifications. Program logics are more amenable to full automation for properties like memory safety than operational approaches. But most program logics cannot be applied soundly to programs that have data races. This is because they assume a naïve sequentially consistent memory model. But modern microprocessor do not guarantee this strong condition for racy programs. Hence, program logics based on, e.g., Hoare logic or separation logic, must either restrict their scope to well locked programs, or else admit unsoundness w.r.t. realistic memory models.

Our work, by contrast, develops a program logic designed from the start to cope with racy programs. It does this by deeply incorporating details of the x86 memory model instead of naïvely assuming sequential consistency. In particular, the logic exposes the concept of a write buffer, which is central to the standard description of the x86 memory model. Our logic, makes reasoning about the interaction between programs, memory and write buffers simple when the interaction is simple, and manageable when the interaction is complicated.

Our work is additionally based on the idea of local reasoning, in which the resources referenced by a program's specification and proof are restricted to just those referenced by the program at runtime. This is in contrast to global reasoning, which relies on a description of the entire system state for sound reasoning. Local reasoning is critical both for proof malleability and modularity because a component's local specification is insensitive to the other parts of a complete system that are unrelated to its operation.

5 Proposed Work

We propose to continue this avenue of research primarily by automating a fragment of the theory useful for proving memory safety and partial correctness. There is a long history of automating program logics to yield semi-automated verifiers and fully automatic program analyzers, most recently with theories based on separation logic [1], which is particularly well suited to automation. As a first step, we propose to build a basic program checker to verify programs that have been annotated with pre- and post-specifications, as well as loop and concurrent resource invariants. For larger programs, this is too heavy an annotation burden, but for smaller programs like concurrent data structures the approach is feasible and could demonstrate the usefulness of the logic for basic reasoning about a difficult class of concurrent programs.

If this automation is successful, we propose also to develop a related theory for Power- and ARM-like multiprocessors. Like the x86, these are widespread classes of processors, which have been the focus recent formal specification efforts [5]. The memory models of Power and ARM processors are similar to one other, but quite different from the x86 memory model—neither stronger nor weaker. The development of a related theory for Power and ARM is needed to ensure memory safety and other safety properties of concurrent programs for these architectures. Such a theory would illustrate the salient differences and similarities with x86, and may lead the way to a weaker, more general theory useful for reasoning about the execution of programs on any of these architectures.

A final potential area of exploration is the development an extended theory for progress properties as well as safety properties. Of particular relevance to concurrent data structures is the hierarchy of *non-blocking* progress properties, which distinguish the conditions under which the operations of parallel threads are guaranteed to terminate. Performance is the primary motivation of concurrent data structures, so the assertion of properties that ensure timely completion of their operations is crucial to their trustworthiness. Previous work on automating progress proofs for concurrent data structures using separation logic [2], though unsound w.r.t. realistic memory models, is a natural starting point.

6 EAGER Relevance

The availability of trustworthy concurrent data structures would have a significant impact on the security and efficiency of many computer systems. Large, performance-critical concurrent commercial systems could better leverage underlying hardware parallelism for improved throughput by making use of concurrent data structures in place of simpler coarse-grained data structures. Similarly, security-critical systems can do so without sacrificing the safety of the data they manipulate.

Today, such a change to these applications' underlying data structures—from slow and obviously correct to fast but possibly faulty—is unjustifiably risky in many situations. Indeed, if errors occur only once in every million data structure operations—a plausible ratio, given the subtlety of the algorithms and the complex behavior of the underlying hardware—problems could either arise sufficiently frequently in high-throughput systems or be considered sufficiently likely to occur in high-security systems so as to make their deployment unwise. Simply put, untrustworthy concurrent data structures hampers their adoption and results in significant inefficiencies for concurrent applications. But without more sophisticated techniques for reasoning about the correctness of concurrent data structures, their verification remains economically infeasible for all but the most critical applications.

The approach to reasoning about racy, concurrent programs proposed here differs significantly from the current approaches of reasoning about a program's operational meaning, and from most program logics that require race-freedom for sound operation. Consequently, the operational approach requires significant time and expertise to apply, making it prohibitively expensive in practice, while existing program logics are simply unsound for racy concurrent data structures. Sound tools based on program logics, however, could fundamentally change the way concurrent data structures are constructed and verified; e.g., with compile-time checks made throughout the development process, instead of with brittle proofs painstakingly constructed by experts.

The success of such a tool depends heavily on the foundational theory of x86 program reasoning currently under development as part of the second author's dissertation work. This is still rapidly evolving, and remains largely untested for programs of moderate size. But a successful application of the theory that culminates in an automated tool for analyzing concurrent data structures could significantly reduce the cost of their verification and deployment, resulting in improved performance and safety for a variety of applications. It is for these reasons that we consider the proposed work well suited for the EAGER program.

7 Prior NSF Support

Award: CNS-0910913; $799,943.00; 09/01/2009—08/31/2011

Title: TC: LARGE: A Formal Platform for Analyzing Internet Routing

Summary: The goal of this project is to develop a formal analysis framework for effectively analyzing reactive, concurrent, networking systems such as routers. These systems are characterized by a collection of interacting components, each of which performs non-terminating, ongoing computation while interacting with its environmental stimuli. The environmental stimulus for a component includes communications received from other components, in addition to the external world (such as a user request). Realizing this framework entailed the development of formal models of networks and concurrency, the proving of key properties of the model, and the development of tools for automating these proofs. This project has entailed close collaboration with several semiconductor companies, which facilitate smooth transfer of research results into their tool flow. The results are disseminated through publications as well as by making publicly available our analysis tool suite, reference flow, and formal definitions. This facilitates a better understanding both of the workings of networks and of the design invariants that must be preserved in the implementation of network protocols.

Publications:

1. Michael J. C. Gordon, Matt Kaufmann, and Sandip Ray, "The Right Tools for the Job: Correctness of Cone of Influence Reduction Proved Using ACL2 and HOL4", Journal of Automated Reasoning, p. , vol. , (2010). Accepted, 10.1007/s10817-010-9169-y
2. Warren A. Hunt, Jr. and Sol O. Swords, "A Mechanically Verified AIG-to-BDD Conversion Algorithm", International Conference on Interactive Theorem Proving, p. , vol. , (2010).
3. Sandip Ray and Rob Sumners, "A Theorem Proving Approach to Verification of Reactive Concurrent Programs", 4th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC2 2011), p. , vol. , (2011).
4. Ian Wehrman and Josh Berdine, "A Proposal for Weak-Memory Local Reasoning", Syntax and Semantics of Low-Level Languages, p. , vol. , (2011).
5. Matt Kaufmann and J Strother Moore, "How Can I Do That with ACL2? Recent Enhancements to ACL2", 10th International Workshop on the ACL2 Theorem Prover and Its Applications, p. , vol. , (2011).
6. Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios, "Integrating Testing and Interactive Theorem Proving", 10th International Workshop on the ACL2 Theorem Prover and Its Applications, p. , vol. , (2011).

Data management: No data is gathered, only the concepts and formal models generated as part of the research process, which are disseminated in conference and journal publications.

References

[1] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-level Code. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[2] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 16–28. ACM, 2009.

[3] M. Hall and D. Padua, editors. *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, California, June 4-8, 2011*. ACM, 2011. To appear.

[4] S. Ray, W. A. H. Jr., J. Matthews, and J. S. Moore. A mechanical analysis of program verification strategies. *J. Autom. Reasoning*, 40(4):245–269, 2008.

[5] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In Hall and Padua [3]. To appear.

[6] I. Wehrman. Semantics and syntax of a weak-memory concurrent separation logic: Sequential fragment. `http://cs.utexas.edu/~iwehrman/x86-logic/sequential.pdf`, 2010.

[7] I. Wehrman and J. Berdine. A proposal for weak-memory local reasoning. In *Syntax and Semantics of Low-Level Languages (LOLA)*, 2011.

[8] Q. Zeng, D. Wu, and P. Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In Hall and Padua [3]. To appear.