Computer-Assisted Construction of Efficient Concurrent Algorithms

Martin Vechev, Eran Yahav, Maged Michael, Hagit Attiya, Greta Yorsh

Practical and efficient concurrent systems are notoriously hard to design, implement, and verify. As concurrent systems become a larger part of society's mission-critical infrastructure, it is imperative that we raise the level of confidence in the correctness of these systems, and that we understand the trade-offs inherent in our algorithmic choice.

Current practices for developing concurrent systems are rather limited. Directly using low-level concurrency constructs is the realm of experts, and is extremely error-prone. Generic higher-level constructs (e.g., transactional memory) are currently limited, and are not clearly easier to use. Analytic techniques (e.g., race detection) only address a fraction of the problems, and can only be applied after the code is written and is potentially broken in a fundamental manner. This motivates us to explore a methodology and a tool that assist an algorithm designer *during* the construction process. Our hope is that at least for specific domains it would be possible to *generate efficient provably correct concurrent systems from higher-level specifications*.

A Motivating Example

Fig. 1 shows a standard sequential implementation of a set data structure. This implementation uses an underlying singly linked-list. The list is sorted in ascending key order, and uses two sentinel nodes *head* (with a key value smaller than the smallest possible key in the set) and *tail* (with a key value larger than the largest possible key in the set). The set supports three operations: `add`, `remove`, and `contains`, with their standard meaning. All operations use a macro LOCATE to traverse the list and locate an item based on the value of its key.

```
bool add(int key) {
 Entry *pred,*curr,*entry;
   LOCATE(pred, curr, key)
   k=(curr->key == key)
   if(k) return false
   entry = new Entry(key)
   entry->next = curr
   pred->next = entry
   return true
}
```

```
bool remove(int key) {
 Entry *pred,*curr,*r;
   LOCATE(pred, curr, key)
   k=(curr->key ≠ key)
   if(k) return false
   r = curr->next
   pred->next = r
   return true
}
```

```
bool contains(int key) {
 Entry *pred,*curr;
   LOCATE(pred, curr, key)
   k=(curr->key==key)
   if(k) return true
   if(¬k) return false
}
```

```
LOCATE(pred, curr, key) {
  pred=head
  curr = head->next
  while(curr->key < key){
   pred=curr
   curr = curr->next
  }
}
```

Figure 1: A sequential implementation of a set algorithm based on a sorted singly-linked-list.

Fig. 2 shows a new concurrent set algorithm we derived (see [21] for details). In the figure, we use $\langle ptr, bit \rangle$ to denote a pair of a reference and a bit value stored in the same word (as done e.g., in [9, 17]). The first thing to note is that the concurrent algorithm of Fig. 2 is quite distant from the sequential implementation of Fig. 1, and in particular, uses the lower-level compare and swap (CAS) synchronization primitive. Moreover, looking at the concurrent algorithm, it is quite challenging to understand how it works and why certain choices were made. For example, how do we separate implementation details (such as using a CAS) from key algorithmic insights? How does this algorithm compare to other sophisticated algorithms such as [9, 10, 17]? What are the commonalities between these algorithms? How can the trade-offs between the algorithms be described formally? What does it mean exactly for a concurrent algorithm to be more efficient or more concurrent than another?

```
boolean add(int key) {
 Entry *pred,*curr,*entry;
restart :
 LOCATE(pred, curr, key)
 k=(curr->key == key)
 if(k) return false
 entry = new Entry(key)
 entry->next = curr
 val=CAS(&pred->⟨next, marked⟩, ⟨curr, 0⟩, ⟨entry, 0⟩)
 if(¬val) goto restart
 return true
}
```

```
boolean remove(int key) {
 Entry *pred,*curr,*r
restart :
 LOCATE(pred, curr, key)
 k=(curr->key ≠ key)
 if(k) return false
 ⟨r, m⟩ = curr->⟨next, marked⟩
 lval=CAS(&curr->⟨next, marked⟩, ⟨r, m⟩, ⟨r, 1⟩)
 if(¬lval) goto restart
 pval=CAS(&pred->⟨next, marked⟩, ⟨curr, 0⟩, ⟨r, 0⟩)
 if(¬pval) goto restart
 return true
}
```

Figure 2: A concurrent set algorithm using a marked bit to mark deleted nodes. The bit is stored with the $next$ pointer in a single word. Synchronization is implemented using CAS. The code of `contains` remains as the one of Fig. 1, does not use synchronization and does not restart. The LOCATE is the one of Fig. 1. The algorithm assumes an automatic garbage collector.

Challenges

In the following, we describe three major challenges that we extracted from our experience with construction and verification of efficient concurrent algorithms.

Challenge 1: Leveraging Commonalities between Algorithms
Construction and implementation: can we reuse construction steps of other algorithms in order to create a new one? Can the construction of new algorithms be partially automated?
Verification: can we share proof effort by using the same proof steps for multiple algorithms?

The quest to find repeatable steps or a general methodology for constructing concurrent algorithms is not a new idea. A paper by Herlihy [11] outlines a precise methodology for constructing concurrent algorithms. Another line of work revolves around building powerful primitives. The idea being that one could use these primitives as building blocks of more complex algorithms. An example of a primitive is a multiple-compare-and-swap (MCAS) operation. Among others, such implementations were provided in [7, 13]. These approaches, while providing clear primitives or steps, typically result in impractical algorithms. Another flavor of universal constructions is Software Transactional Memory (STM), for which there are various implementation (e.g. [5, 8]).

It was initially envisioned that one could convert sequential algorithms to concurrent ones by wrapping the algorithm with a transaction. In this approach, assuming the sequential implementation is correct, the resulting transactional implementation will be correct by construction.

Universal constructions, such as STM, trade-off efficiency for correctness. Unfortunately, in some cases the space and time overhead incurred by these methods is unacceptable in practice. The core source of the inefficiency of STM is that it is oblivious to the concurrent object's specification. Without a specification, it is necessary to be conservative and restart operations even when the concurrent execution does not violate the underlying (missing) specification. As a result, an efficient implementation using STM often has to break an operation into multiple fine-grained transactions, leading to the same correctness challenges as non-transactional fine-grained implementations.

Significant efforts have been made towards proving and deriving various concurrent algorithms, e.g. [3, 4, 6, 14, 19]. However, existing derivation methodologies are geared towards reconstructing an existing algorithm by a series of provably-correct derivation steps. The emphasis in these approaches is not on finding *repeatable* derivation steps that could be used across algorithms. The fact that steps are not repeatable, together with the intricate formal details of these methods, leads to limited adoption by algorithm designers. In fact, these methods are mainly used as a proof methods of existing algorithms rather than as a construction method of new algorithms.

Challenge 2: Comparing Algorithms
Asymptotic complexity provides a natural way to compare sequential algorithms. However, for concurrent algorithms, it is not clear how they should be compared. For example, what does it mean for one algorithm to be "more concurrent" than another?

Since we are interested in constructing efficient algorithms, it is mandatory that we formally define what does it mean for an algorithm to be efficient, and how algorithms could be compared in terms of efficiency. Initial studies in [1, 13] provided theoretical definitions, but it is still not clear whether these classifications are fine enough to distinguish algorithms that exhibit significant differences in practice.

Challenge 3: Representation
How do you represent practical concurrent algorithms in a way that exposes underlying dependencies?

In our experience, one of the key challenges underlying all of these questions is the lack of a meaningful algorithmic *representation*. Ideally, such a representation would make it easier to extract the maximal concurrency of an algorithm and make it natural to perform transformation steps between algorithms.

The problem with representation is again a gap between theory and practice. Concurrency theory offers a wealth of models for reasoning about concurrency: Petri nets [20], Process algebras (e.g., CCS [18], CSP [12], ACP [2]), trace theory [16], and more. Despite (or due to?) the wide variety of theoretical models, the concurrency model used in most widely adopted programming languages is based on the notion of *threads*. Threads require the algorithm designer to explicitly reason about all possible interleavings of her program and prune invalid interleavings by using synchronization constructs. Obtaining a correct and efficient concurrent algorithm using this abstraction of concurrency is extremely difficult [15]. Often, the programmer is forced to manually overconstrain the concurrency in order to make reasoning about correctness feasible. Finding a practical yet well-founded model for representing concurrent programs is the fundamental challenge underlying all other challenges described in this paper.

Past Experience

As a first modest step to address the aforementioned challenges, we engaged in an initial case study involving two fundamental domains: concurrent garbage collection [22, 23] and concurrent data structures [21].

The specific challenge we tackled in these domains was: *constructing algorithms with fine-grained atomic sections while striving to find repeatable steps*.

To help us address this specific challenge, we constructed a tool called PARAGLIDER for the automatic exploration of a space of concurrent algorithms. In our framework, the designer specifies a set of "building blocks" from which algorithms can be constructed. These blocks reflect the designer's insights about: (a) the sequential behavior of the operations, (b) the coordination metadata employed in the concurrent implementations, and (c) the synchronization operations used to coordinate between different threads. Given a set of building blocks, our framework automatically explores the space of algorithms, using model checking with abstraction to verify algorithms in the space. When a full coverage of the induced space turns out to be infeasible, we expect the designer to simplify the search.

In preliminary experiments with this approach, we found that this form of an iterative design also benefits the designer: she can start the exploration by specifying only a few building blocks and placing severe constraints; use our framework; study the resulting algorithm(s); detect a common pattern; and reuse our framework with a more sophisticated input constraining our system to respect the common pattern.

Using our approach, we were able to discover new algorithms as well as reconstruct existing ones, while separating algorithmic insights from implementation details.

Our case study sheds some light on Challenge 1. However, our current approach is limited and requires a significant amount of human insight. We believe that a satisfactory construction methodology must first address the

fundamental problem of representation (Challenge 3), and the problem of expressing relationships between algorithms (Challenge 2). With a suitable representation, we can re-visit the description of commonalities between algorithms (Challenge 1).

The Road Ahead

We are still in the early days of these experimental construction approaches, but the results we obtained so far give us hope for the future. In the immediate future, we plan on leveraging our current insights towards building a practical tool that can assist an algorithmic designer.

References

[1] ATTIYA, H., AND DAGAN, E. Improved implementations of binary universal operations. *J. ACM 48*, 5 (2001), 1013–1037.

[2] BAETEN, J. C. M., AND WEIJLAND, W. P. *Process algebra*. Cambridge University Press, 1990.

[3] COLVIN, R., AND GROVES, L. A scalable lock-free stack algorithm and its verification. In *SEFM* (2007), IEEE Computer Society, pp. 339–348.

[4] COLVIN, R., GROVES, L., LUCHANGCO, V., AND MOIR, M. Formal verification of a lazy concurrent list-based set algorithm. In *CAV* (2006).

[5] DAVE DICE, O. S., AND SHAVIT, N. Transactional locking ii. In *DISC* (2006).

[6] FEIJEN, W. H. J., AND VAN GASTEREN, A. J. M. *On a method of multiprogramming*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[7] FRASER, K. Practical lock freedom. *PhD thesis* (2003).

[8] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. *SIGPLAN Not. 38*, 11 (2003), 388–402.

[9] HARRIS, T. L. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proc. of conf. on Distributed Computing* (London, UK, 2001), Springer, pp. 300–314.

[10] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *Proc. of conf. On Principles Of Distributed Systems (OPODIS 2005)* (2005), pp. 3–16.

[11] HERLIHY, M. A methodology for implementing highly concurrent data objects. *TOPLAS*, 5 (1993).

[12] HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (1978), 666–677.

[13] ISRAELI, A., AND RAPPOPORT, L. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC* (1994), ACM, pp. 151–160.

[14] JACKSON, P. B. Verifying a garbage collection algorithm. In *Theorem Proving in Higher Order Logics, 11th International Conference* (1998), LNCS, Springer-Verlag, pp. 225–244.

[15] LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.

[16] MAZURKIEWICZ, A. Concurrent program schemes and their interpretation. Tech. rep., 1977.

[17] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *SPAA* (2002), pp. 73–82.

[18] MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[19] PAVLOVIC, D., PEPPER, P., AND SMITH, D. R. Colimits for concurrent collectors. In *Verification: Theory and Practice* (2003), vol. 2772 of *LNCS*, Springer-Verlag, pp. 568–597.

[20] REISIG, W. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[21] VECHEV, M. T., AND YAHAV, E. Deriving fine-grained concurrent linearizable objects. In *PLDI* (2008).

[22] VECHEV, M. T., YAHAV, E., AND BACON, D. F. Correctness-preserving derivation of concurrent garbage collection algorithms. In *PLDI* (2006), ACM, pp. 341–353.

[23] VECHEV, M. T., YAHAV, E., BACON, D. F., AND RINETZKY, N. CGCExplorer: a semi-automated search procedure for provably correct concurrent collectors. In *PLDI* (2007), pp. 456–467.