Thread Summaries for Lock-Free Data Structures*

Sebastian Wolff

Fraunhofer ITWM, Kaiserslautern, Germany
Technische Universität Kaiserslautern, Germany
s.wolff@cs.uni-kl.de

It is an indubitable fact that lock-free data structures are unrivaled in terms of scalability on today's multi-core platforms. Unfortunately, they are also unrivaled in terms of complexity which is why automated verification is indispensable. However, static analyses suffer from poor scalability as they have to consider an unbounded number of client threads. This requirement is crucial since data structures are usually part of a library. Hence, one wants to verify the library for all possible usage scenarios once instead of verifying every usage.

In the following, we address this challenge. Additionally, we permit low-level memory interactions as known from languages like `C`. Currently, the most efficient verification technique to tackle this problem is so-called *thread-modular reasoning* [2, 1]. It abstracts the state space into a set of views. Each view captures information about an individual thread only – the relations among different threads is lost. In order to explore the state space two rules are applied exhaustively to generate new views from the existing ones. (1) Sequential rule: a view is changed by an action from the thread that it represents. (2) Interference rule: a victim view is updated according to the effect of a thread from another, interfering view taking an action. To do so the victim and interfering views are combined, the interfering thread takes an action and is finally projected away. This step is, however, problematic for two reasons: non-existent relations among threads might give false-positives and an exhaustive application requires to consider quadratically many pairs of views.

It is indeed this quadratic blow up during the state space exploration which makes thread-modular reasoning scale rather poorly [1] even though optimizations exist [4]. Intuitively, we want to avoid interference steps altogether in order to provide a truly scalable verification approach. Technically, we introduce *thread summaries* for this task and rephrase the verification problem in such a way that results for the modified problem carry over to the original one. Since this is work in progress we discuss our approach rather informally to convey the idea.

Example As an example consider Michael&Scott's queue the code of which is given in Listing 1. It maintains a singly-linked list with the pointers `Head` and `Tail` pointing to the first and last entry, respectively. The main problem introduced by lock-freedom is that adding a new entry to the list and swinging the `Tail` pointer to the newly inserted entry requires two separate actions which are not guarded from interfering threads. As a consequence, the data structure might appear malformed because of `Tail` not pointing to the last node. To resolve this issue every thread checks for this condition first and tries to *fix* the queue if needed.

In order to reason about the correctness of Micheal&Scott's queue one has to understand and exploit the underlying protocol. A protocol which is not unique to this implementation [3]. Figure 2 presents a modified version of the code which comprises the same protocol. However, the complexity of the actual implementation is avoided by *summarizing* the intended effect. That is, the original functions are rephrased as if they were executed atomically. It is worth noting that those *summaries* have properties quite beneficial for program verification. They avoid the read-copy-update cycle and the subtle interplay of threads. Moreover, they do not

*I would like to thank Lukáš Holík, Roland Meyer, and Tomáš Vojnar for the fruitful collaboration giving rise to the theory presented here. I would also thank Roland Meyer for the valuable comments on this paper.

```
/* shared */ pointer_t Head, Tail;
void enqueue(data_t v) {
  node = new Node(v);
  node.next = NULL;
  while (true) {
    tail = Tail;
    next = tail.next;
    if (tail != Tail) continue;
    if (next == NULL) {
      if (CAS(Tail.next, next, node))
        break;
    } else { CAS(Tail, tail, next); }
  }
  CAS(Tail, tail, node);
}
```

```
bool dequeue(data_t* v) {
  while (true) {
    head = Head;
    tail = Tail;
    next = head.next;
    if (head != Head) continue;
    if (head == tail) {
      if (next == NULL) return false;
      CAS(Tail, tail, next);
    } else {
      *v = head.data;
      if (CAS(Head, head, next)) {
        delete head;
        return true;
}}}}
```

Figure 1: Micheal&Scott's non-blocking queue, adapted from [5].

```
void $enqueue() {
  atomic {
    if (Tail.next != NULL) {
      Tail = Tail.next;
    } else {
      v = random_value();
      node = new Node(v);
      node.next = NULL;
      Tail.next = node;
}}}
```

```
void $dequeue() {
  atomic {
    if (Head == Tail) {
      if (Tail.next != NULL)
        Tail = Tail.next;
    } else {
      tmp = Head;
      Head = Head.next;
      delete tmp;
}}}
```

Figure 2: Thread summaries for Micheal&Scott's queue.

require any local state but determine their behavior by inspecting the shared heap. Our ambition is to use such summaries in order to enable more efficient static analyses.

Abstract System To overcome the poor scalability of static analyses for systems with an unbounded number of client threads we introduce a novel abstraction. Our *abstract* system is easier to analyse as it comes with just two concurrently operating threads: a *low-level* thread which executes the program code under scrutiny and a *high-level* thread which executes *thread summaries* like the ones presented above. Note that unlike before this system features two different types of threads.

The main contribution of our work is a theorem which guarantees that safety properties of the abstract system carry over to the concrete one. To that end, we instrument programs such that *bad states* can be identified inspecting the shared heap, e.g. by using a shared auxiliary variable. Thus, in order to verify safety properties it suffices that the reachable shared heap configurations of the proposed abstract system over-approximate those of the concrete system.

Thread Summaries In order to derive the desired theorem we have to establish a tight correspondence between the executions of the concrete and the abstract system. Intuitively, given some execution of the concrete system we pick an arbitrary thread as the low-level thread of the abstract system and replace every action of the other threads with summaries. We require those summaries to only mimic on the shared heap the actions that they replace. That is, they shall not modify the non-shared, local state of other threads, i.e. the low-level thread we just picked. It is worth noting that choosing summaries like that simplifies their task as they can ignore the local state of other threads. This results in fewer possibilities for summaries to react and thus gives a smaller search space during an analysis – a major step towards a scalable

approach. However, for a sound analysis we have to prove that no thread may ever influence the local state of another thread. We do so by enforcing a strict partitioning of the addresses in use into shared and owned ones. Moreover, thread summaries must not leak memory. Hence, cells owned by threads in the concrete system are freed in the abstract system if the owning thread is not present. Together with checking for malicious memory accesses, which we showed to be effective in previous work [4], we get adherence of both the high-level and the low-level thread to that partitioning. In particular, this means that no low-level thread will ever rely on cells that are owned by other threads in the concrete system as they appear freed in the abstract one.

Altogether, we come up with the following theorem which states the desired correspondence between abstract and concrete executions and allows a sound analysis of safety properties.

Theorem. *Let S be some concrete system with low-level threads $t_1, \ldots, t_n$ and let S' be the corresponding abstract system with some low-level thread t_i and high-level thread t_s. If the summaries executed by t_s can (i) mimic every t_i action on the shared heap, and (ii) are stateless, and if (iii) S' adheres to the address partitioning, then*

$$reach_{shared}(S) \subseteq reach_{shared}(S') \quad \textit{and} \quad reach_{local}(t_i, S) \simeq reach_{local}(t_i, S')$$

where $reach_{shared}$ yields the reachable configurations of the shared heap, $reach_{local}$ gives the reachable configurations of the local heap and $\simeq$ denotes isomorphism up to dangling pointers. □

Lastly, let us briefly discuss why we include a low-level thread in the abstract system at all. Why should we not just use a single high-level thread? In fact, keeping a low-level thread live in the abstract system allows one to verify whether or not the used thread-summary is correct, i.e. adheres to the requirements. This can be crucial. On the one hand, this enables refinement loops to be used. On the other hand, it allows for user-defined summaries while guaranteeing a sound analysis. Since we did not investigate how to effectively compute thread summaries yet the latter argument makes our approach practical even at this early stage.

Experiments We implemented a prototype[1] to check linearizability of lock-free data structures. The prototype is based on our previous tool which uses thread-modular reasoning [4]. We replaced interference steps with thread summaries and implemented all necessary checks to ensure their correctness. Our experiments prove our new verification technique to be truly superior to classical thread-modular reasoning in terms of scalability: while the analysis of Treiber's stack is only faster by factor 25, a more evident performance boost of factor 196 can be observed for Michael&Scott's queue. This substantiates the usefulness of our novel approach.

References

[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2013.

[2] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

[4] F. Haziza, L. Holík, R. Meyer, and S. Wolff. Pointer race freedom. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2016.

[5] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275. ACM, 1996.

[1] Available at: https://github.com/Wolff09/TMRexp/tree/NWPT16