Efficiently Inferring Thread Correlations
Technical Report TR-09-59203

M. Segalov[1], T. Lev-Ami[1], R. Manevich[2], G. Ramalingam[3], and M. Sagiv[1]

[1] Tel Aviv University, {tla,segalovm,msagiv}@post.tau.ac.il
[2] University of California Los Angeles, rumster@gmail.com
[3] Microsoft Research India, grama@microsoft.com

Abstract. We present a new analysis for proving properties of fine-grained concurrent programs with a shared, mutable, heap in the presence of an unbounded number of objects and threads. The properties we address include memory safety, data structure invariants, partial correctness, and linearizability. Our techniques enable successful verification of programs that were not be handled by previous concurrent shape analysis algorithms. We present our techniques in an abstract framework we call *thread-correlation analysis*. Thread-correlation analysis infers invariants that capture the correlations between the local states of different threads and the global state (content of the heap).
However, inferring such invariants is non-trivial, since it requires reasoning about a quadratic number of interactions between threads. We provide two novel techniques for reducing the cost of applying the abstract transformers. In our experiments, these techniques were able to reduce the time of the analysis by a factor of 35.

1 Introduction

This paper is concerned with analysis and verification (of basic safety and other functional correctness properties) of fine-grained concurrent programs, especially those with dynamically-allocated concurrent data structures. Such data-structures are important building blocks of concurrent systems and are becoming part of standard libraries (e.g., JDK 1.6).

We would like to automatically verify properties such as memory safety, preservation of data structure invariants, and linearizability [14] for such algorithms. Automatic verification of these algorithms is challenging because they often contain fine-grained concurrency with benign data races, low-level synchronization operations such as `CAS`, and destructive pointer-updates which require accurate alias analysis. Furthermore, in the general case, the data-structure can grow in an unbounded fashion and the number of threads concurrently updating the data-structure can also grow in an unbounded fashion.

The essential contributions of our paper, however, are independent of shape analysis and can be used for other analyses of concurrent programs as well. For this reason, we describe our techniques in a simple setting, independent of shape

analysis. (Our implementation, however, realizes these ideas in a shape analysis and our empirical results concern this concurrent shape analysis.)

Our approach follows a line of work for verifying concurrent programs in a thread-modular way [2, 5, 6, 8, 12], in which concrete states are abstracted from the point of view of a given thread.

Thread Correlation Abstraction. In order to handle fine-grained concurrency, we defer from "pure" thread-modular abstractions by tracking correlations between thread local states. Our abstraction records correlations between pairs of (abstract) thread states where one thread state is the primary thread and the other, usually abstracted more coarsely, is part of the environment.

Efficiently Reasoning about Interference. Our experience with analyzing programs using the thread correlation abstraction has shown that it is rather precise in verifying the programs and properties of interest albeit very costly However, reasoning about thread interference in the computation of the abstract post operator, requires taking into account pairs of thread views, which incurs a quadratic factor. In practice, we found that this creates a major bottleneck, which prevents successful verification of even some simple programs (within the time bounds we used). One of the key contributions of this paper is a set of techniques for computing the abstract post more efficiently. The results presented in this paper enable successful verification of programs that were not be handled by previous concurrent static analysis algorithms.

Main Contributions. The main contributions of this paper are:

Abstract Domain We present a new abstract domain for reasoning about concurrent programs with fine-grained concurrency. The new domain makes the idea of process-centric abstraction (also referred to as "environment abstraction" [5, 6]) explicit in terms of thread correlations expressed *directly over the concrete domain*, as opposed to specific abstractions [5] or parametric abstract domains [2, 12]. We believe the ideas are applicable to concurrent program analysis problems other than shape analysis and are more accessible.

Sound Transformer We define a sound abstract post operator (transformer) for the new abstraction from the concrete sequential semantics. The transformer reasons rather precisely about interference between threads.

Transformer Optimizations We present two refinements to the computation of the above transformers that lead to significant speedups.

Implementation We have implemented an analysis based on the new abstract domain and used it to automatically verify properties of several concurrent data structure implementations.

Evaluation We present an empirical evaluation of our techniques and show the advantages of the optimizations to the abstract transformer computation. For example, for a lock-free implementation of a concurrent set using linked lists [21], our optimizations reduce the analysis time from 56, 347 CPU

seconds to 1,596 — a 35 fold speed-up. We have also analyzed erroneous mutations of concurrent algorithms and our tool quickly found errors in any of the incorrect variations.

The work described in this paper is a continuation of [2]. The main contributions of this paper are: we introduce a new, simple, abstract domain for capturing correlations between pairs of threads and describe techniques for computing transformers efficiently for this domain; we also present an empirical evaluation of our approach that shows that our techniques lead to a dramatic reduction in verification time while still being able to prove the same properties.
Outline of the rest of this paper. Sec. 2 presents an overview of our analysis in a semi-formal way. Sec. 3 formalizes our analysis using the theory of abstract interpretation [7]. Sec. 4 defines optimizations to the transformers. Sec. 5 evaluates the effectiveness of our optimizations on realistic benchmarks. Sec. 6 concludes with discussion of related works.

2 Overview

In this section, we explain our approach informally, using a very simple example we adapted, originally constructed to show the limitations of concurrent separation logic [22]. We use this example to motivate the need for tracking thread correlations and show the difficulties in computing postconditions efficiently. Fig. 1 shows a concurrent program with producer threads and consumer threads communicating via a single-object buffer, `b`, and a global flag `empty`. For simplicity, instead of locks or semaphores, we use the `await` construct, which atomically executes the then-clause when the await-condition holds.

```
Boolean empty = true;
Object b = null;
```

```
produce() {
[1]   Object p = new();
[2]   await (empty) then {
        b = p; empty = false;
      }
[3] }
```

```
consume() {
      Object c;
      // Boolean x;
[4]   await (!empty) then {
        c = b; empty = true;
      }
[5]   use(c);
      // x = f(c);
[6]   dispose(c);
      // use(x);
[7] }
```

Fig. 1. A concurrent program implementing a simple protocol between a producer thread and a consumer thread transferring objects in a single-element buffer. The commented out lines are only used and explained in Sec. 4

2.1 The Need for Thread Correlations

In this example, the system consists of an unbounded number of producer and consumer threads. Each producer allocates a new object, transfers it to a single

consumer via the buffer, and the consumer uses the object and then deallocates the object. Our goal is to verify that `use(c)` and `dispose(c)` operate on objects that have not been deallocated. (This also verifies that an object is not deallocated more than once.)

One way to verify properties of concurrent systems is by establishing a global invariant on the reachable configurations and show that the invariant entails the required properties (e.g., see [1]). In our program, we need to show that the following property holds:

$$\forall t \,.\, pc[t] \in \{5,6\} \Rightarrow a(c[t]) \ , \tag{1}$$

where t ranges over threads, $c[t]$ denotes the value of the variable `c` of thread t, and $a(c[t])$ is true iff $c[t]$ points to an object that has not yet been disposed.

This verification requires the computation of an inductive invariant that implies (1). In particular, the invariant should guarantee that the dispose command executed by one consumer thread does not dispose an object used by another consumer thread and that an object that a producer places in the buffer is not a disposed object.

A natural inductive invariant that generalizes (1) is:

$$\begin{array}{rll} & pc[t] \in \{5,6\} \Rightarrow a(c[t]) & \wedge \quad (i) \\ & \neg empty \Rightarrow a(b) & \wedge \ (ii) \\ \forall t, e \,.\, & pc[t] = 2 \Rightarrow a(p[t]) & \wedge \ (iii) \\ & t \neq e \wedge pc[t] = 2 \Rightarrow p[t] \neq c[e] & \wedge \ (iv) \\ & t \neq e \wedge pc[t] \in \{5,6\} \Rightarrow c[t] \neq c[e] & \quad (v) \end{array} \tag{2}$$

This invariant ensures that dispose operations executed by threads cannot affect locations pointed-to by producer threads that are waiting to transfer their value to the buffer and also cannot affect the values of other consumer threads that have not yet disposed their values. Here e is a thread that represents the environment in which t is executed. Specifically: (i) describes the desired verification property; (ii) is the buffer invariant, which is required in order to prove that (i) holds when a consumer copies the value from the buffer into its local pointer `c`; (iii) establishes the producer properties needed to establish the buffer invariant. The most interesting parts of this invariant are the *correlation invariants* (iv) and (v), describing the potential correlations between local states of two arbitrary threads and the content of the (global) heap. These ensure that the invariant is inductive, for example (v) ensures that (i) is *stable*, i.e., deallocations by different threads cannot affect it, if it already holds. Also, notice that the correlation invariants cannot be inferred by pure thread-modular approaches. Our work goes beyond pure thread-modular analysis [9] by explicitly tracking these correlations.

2.2 Automatically Inferring Correlation Invariants

In this paper, we define an abstract interpretation algorithm that automatically infers inductive correlation invariants. The main idea is to infer normalized

invariants of the form:

$$\forall t, e \,.\, \bigvee_{i=1}^{n} \varphi_i[t, e] \tag{3}$$

where t and e are universally quantified thread variables and $\varphi_i[t, e]$ are formulas taken from a finite, but usually large, set of candidates. We will refer to each $\varphi_i[t, e]$ as a *ci-disjunct* (Correlation-Invariant Disjunct). As in predicate abstraction and other powerset abstractions, the set of ci-disjuncts is computed by successively adding more ci-disjuncts, starting from the singleton set containing a ci-disjunct describing t and e in their initial states. For efficiency, $\varphi_i[t, e]$ are usually asymmetric in the sense that they record rather precise information on the current thread t and a rather coarse view of other threads, represented by e.

For this program, we can use conjunctions of atomic formulas describing: (i) that t and e are different, (ii) the program counter of t; (iii) (in)equalities between local pointers of t and e, and between local pointers of t and global pointers; (iv) allocations of local pointers of t and global pointers; and (v) the value of the Boolean `empty`.

Thus, the invariant (2) can be written as:

$$\begin{pmatrix} t \neq e & (i)\wedge \\ pc[t] = 5 & (ii)\wedge \\ c[t] \neq c[e] \wedge c[t] \neq b \wedge c[e] \neq b & (iii)\wedge \\ a(c[t]) \wedge a(c[e]) \wedge a(b) & (iv)\wedge \\ \neg empty & (v) \end{pmatrix} \bigvee \begin{pmatrix} t \neq e & (i)\wedge \\ pc[t] = 6 & (ii)\wedge \\ c[t] \neq c[e] \wedge c[t] \neq b \wedge c[e] \neq b & (iii)\wedge \\ a(c[t]) \wedge a(c[e]) \wedge a(b) & (iv)\wedge \\ \neg empty & (v) \end{pmatrix} \bigvee \cdots \tag{4}$$

where the ci-disjuncts describe cases of a consumer thread t that copied the value from the buffer, (which has since been refilled), and has either used the value locally or not. The other disjuncts are not shown.

2.3 Computing Postconditions Efficiently

The iterative procedure successively adds ci-disjuncts describing the reachable states after applying an atomic action to the formula representing the current set of reachable states, until a fixed point is reached. We compute the abstract transformer for an atomic action by identifying its effect on every ci-disjunct $\varphi_i[t, e]$. This is non-trivial since a transition by one thread can affect the global state (and the view of the environment of another thread) and, hence, a ci-disjunct involving other threads.

To compute the effect of a transition on a ci-disjunct $\varphi_i[t, e]$, we need to account for the following three possibilities: (i) The executing thread is t; (ii) The executing thread is e; or (iii) The executing thread is some other thread *ex*. The most challenging case is (iii). In this case, the ci-disjunct does not contain information about the local state of the executing thread *ex*. Applying an abstract transformer without any information about *ex*'s local state can lead to imprecise results. Instead, we exploit the information available in the current set of ci-disjuncts. Specifically, the executing thread *ex* must itself satisfy some ci-disjunct $\varphi_j[ex, t']$. The situation with case (ii) is similar since only limited

information is available about the environment thread in the ci-disjunct and it is handled similarly.

Thus, our transformer works as follows: we consider every pair of ci-disjuncts φ_i and φ_j and apply a "mini-transformer" to this pair. The mini-transformer first checks to see if the two ci-disjuncts are consistent with each other. (E.g., if they imply conflicting values for the global variable `empty`, they cannot correspond to ci-disjuncts from the same concrete state.) If so, it uses the information available about the executing thread from φ_i to determine how the global state will change as a result of the transition, and identifies how that alters ci-disjunct φ_j.

In our experiments, the above abstraction was precise enough to verify the programs analyzed, yet quite slow. One of the key factors for the inefficiency is the quadratic explosion in the transformer, as the transformer has to consider all pairs of ci-disjuncts and the number of ci-disjuncts can become very large.

Our key contributions include effective techniques for making the transformer more efficient by reducing this quadratic factor in common cases, usually without affecting precision. These techniques are analogous to techniques used in interprocedural analysis.

In the rest of this section, let us consider the application of the mini-transformer described above to ci-disjuncts φ_j (corresponding to an executing thread *ex*) and φ_i (corresponding to two other threads t and e).

The first optimization technique, called *summarizing effects*, is based on the following observation. Typically, φ_i can be expressed in the form $\varphi_i^p \wedge \varphi_i^r$, where φ_i^r (the *frame*) cannot be affected by the execution of *ex*. We refer to φ_i^p as the *footprint* of φ_i. E.g., purely local properties of t or e will usually be in the frame. If the transition by *ex* transforms φ_i^p into $\varphi_i^{p'}$, then the transformation of the complete ci-disjunct is given by $\varphi_i^{p'} \wedge \varphi_i^r$. Next, we note that distinct disjuncts φ_i and φ_k may have the same footprint. In this case, it suffices to compute the transformation of the footprint only once.

E.g., consider the first two ci-disjuncts of (4). These ci-disjuncts have the same footprint since they differ only in the program counter value of t which cannot be altered by the execution of *ex*. Typically, the number of distinct footprints created by a set of ci-disjuncts is much smaller than the number of ci-disjuncts, which leads to significant efficiency gains. This optimization is similar to the interprocedural analysis technique where information at the calling context not modified by the call can be transmitted across the procedure call. In the paper, we show the conditions under which this technique can be used to make the transformer more efficient without affecting precision.

The second optimization applies to ci-disjunct φ_j and exploits the locality of the transformer. We abstract away information not used by the transition from φ_j (corresponding to the executing thread), constructing its footprint φ_j^f and use it for the mini-transformer. As distinct ci-disjuncts can have the same footprint, this decreases the number of ci-disjuncts passed to the mini-transformer.

One point to note here is that information not used or modified by an atomic action may still be used by the mini-transformer to check for consistency between the two ci-disjuncts. Such information must be included in the footprint to avoid

a loss of precision. This optimization is an abstraction as it does not guarantee no loss of precision. However, we found that this heuristic significantly improves the computation time while maintaining a precise-enough abstraction.

3 An Abstract Interpretation for Correlation Invariants

In this section, we formalize our analysis, which tracks correlations between pairs of threads, in the framework of abstract interpretation [7].

We start by introducing a concrete semantics (C, TR) for concurrent programs and a corresponding abstract semantics (CI, $TR^{\#}$).

3.1 The Concrete Semantics (C, TR)

A multithreaded program is a parallel composition of concurrently executing threads, where each thread is associated with an identifier from an unbounded set Tid.

The threads communicate through a global store $Glob$, which is shared by all threads. In addition, each thread has its own local store, Loc, which includes the thread's program counter.

A concrete state of the program, $\sigma = (l, g)$, consists of a global store and an assignment of a local store to each thread identifier. We denote the set of all concrete states by $\Sigma = (Tid \to Loc) \times Glob$ and the concrete domain by $C = 2^{\Sigma}$.

Given a state σ, let σ^G represent the global store of σ and let $\sigma^L[t]$ represent the local store of thread t in σ.

The relation $tr \subseteq (Loc \times Glob) \times (Loc \times Glob)$ describes a step that a thread can take, given its local store and a global store. We write $x \rightsquigarrow y$ as shorthand for $(x, y) \in tr$. Let $\sigma^L[t := l]$ denote a state that is identical to σ^L, except that the local store of thread t is l. The concrete transition relation is defined as

$$\begin{aligned} TR = \{(\sigma, \sigma') \mid \exists t \in Tid\,.\, (\sigma^L[t], \sigma^G) \rightsquigarrow (l', g'), \\ {\sigma'}^L = \sigma^L[t := l'], {\sigma'}^G = g'\} \ . \end{aligned} \tag{5}$$

3.2 The Abstract Semantics (CI, $TR^{\#}$)

We now present an abstraction to deal with an unbounded number of threads. As explained in Sec. 1 and Sec. 2, the starting point for our work is the idea of a process-centric abstraction. A process-centric abstraction computes an abstraction of a program state σ from the perspective of a thread t. As we saw in Sec. 2, tracking information about a single thread in the style of thread-modular analysis [8] can be imprecise, and it is useful in many cases for the process-centric abstraction to capture some information about (other threads in) the environment. This motivates the following abstract domain.

We define an abstraction that records correlations between the local stores of two different threads and a global store. Let $CID \equiv Loc \times Glob \times Loc$ denote

the set of such correlations. We will refer to an element of *CID* as a ci-disjunct. We define the abstract domain *CI* to be the powerset 2^{CID}.

The abstraction of a single concrete state is given by

$$\beta^{stm}(\sigma) = \{(\sigma^L[t], \sigma^G, \sigma^L[e]) \mid t, e \in Tid, t \neq e\} \ . \tag{6}$$

Note that a ci-disjunct $(\sigma^L[t], \sigma^G, \sigma^L[e])$ represents the state from the perspective of two threads: t, which we call the primary thread, and e, which we call the secondary thread. We say that $(\sigma^L[t], \sigma^G, \sigma^L[e])$ is a ci-disjunct generated by threads t and e.

The abstraction of a set of states $\alpha: C \to CI$ and the concretization $\gamma: CI \to C$ are:

$$\alpha(X) \equiv \bigcup_{\sigma \in X} \beta^{stm}(\sigma) \ , \qquad \gamma(R) \equiv \{\sigma \mid \beta^{stm}(\sigma) \subseteq R\} \ .$$

Composing With Other Abstractions. Note that when *Loc* and *Glob* are finite sets, *CI* gives us a finite abstraction. In general, it will be useful to compose the above abstraction with a subsequent abstraction to create a finite, tractable, abstract domain. E.g., given any suitable abstraction $CID^\sharp$ of *CID*, we can define a corresponding abstract domain $2^{CID^\sharp}$. Typically, the subsequent abstraction will retain more information about the first local store than about the second local store. As we will see when we describe the transformers, information about the first local store is used to approximate the sequential semantics of program statements whereas the information on the second store is used to express correlation invariants.

Let *CIMap* denote $Loc \times Glob \to 2^{Loc}$. Define the function $map : CI \to CIMap$ by $map(R) \equiv \lambda(\ell, g).\{\ell_e \mid (\ell, g, \ell_e) \in R\}$. The function *map* is bijective and *CI* and *CIMap* are isomorphic domains. However, the form of *CIMap* is more suggestive of a process-centric abstraction and also suggests other subsequent abstractions. E.g., if *ATS* is an abstraction of $Loc \times Glob$ and *Env* is an abstraction of 2^{Loc}, then we can utilize the composed abstraction domain $ATS \to Env$.

In the sequel, we will restrict our attention to the domain *CI*.

An Abstract Transformer. Given $R \in CI$, let $R(\ell, g) \equiv map(R)(\ell, g)$ and $par(R) \equiv \{(\ell, g, R(\ell, g)) \mid \exists \ell_e.(\ell, g, \ell_e) \in R\}$. We define the abstract transformer $TR^\# : CI \to CI$ as follows:

$$TR^\#(R) \equiv \bigcup_{t \in R} tr^{direct}(t) \cup \bigcup_{t_e, t_t \in par(R)} tr^{indirect}(t_e, t_t) \ . \tag{7}$$

The function $tr^{direct} : CID \to 2^{CID}$ captures the effect of a transition by a thread t on a ci-disjunct whose primary thread is t. Abusing terminology, if threads t_p and t_s satisfy $\phi(t_p, t_s)$, where $\phi \in CID$, then after a transition by thread t_p, the threads will satisfy $tr^{direct}(\phi)(t_p, t_s)$.

$$tr^{direct}(\ell_p, g, \ell_s) \equiv \{(\ell'_p, g', \ell_s) \mid (\ell_p, g) \rightsquigarrow (\ell'_p, g')\}. \tag{8}$$

The function $tr^{indirect}$ serves to capture the effect of a transition by a thread t on a set of ci-disjuncts whose primary thread is not t. We use $par(R)$ to collect together the different environments for each pair of local and global stores. The first parameter of $tr^{indirect}$ is the executing thread, and the second is the thread on which we compute the interference. We call this the *tracked thread*.

$$
\begin{array}{l}
tr^{indirect}((\ell_1, g_1, e_1), (\ell_2, g_2, e_2)) \equiv \\
\quad \textbf{if} \quad (g_1 = g_2 \wedge \ell_1 \in e_2 \wedge \ell_2 \in e_1 \wedge e_1 \cap e_2 \neq \emptyset) \\
\quad \textbf{then} \quad \{(\ell_2, g_1', \ell_3) \mid (\ell_1, g_1) \rightsquigarrow (\ell_1', g_1'), \ell_3 \in (e_1 \cap e_2) \cup \{\ell_1'\}\} \\
\quad \textbf{else} \quad \{\} \ .
\end{array} \tag{9}
$$

Theorem 1 (Soundness). *The abstract transformer $TR^{\#}$ is sound, i.e., for all $R \in CI$*

$$TR(\gamma(R)) \subseteq \gamma(TR^{\#}(R)) \ . \tag{10}$$

Proof. Let $\sigma' \in TR(\gamma(R))$, i.e., there is $\sigma \in \gamma(R)$ and $t_{ex} \in Tid$ s.t.,

$$(\sigma^L[t_{ex}], \sigma^G) \rightsquigarrow (l', \sigma'^G) \text{where} \sigma'^L = \sigma^L[t_{ex} := l'] \tag{11}$$

Let $t_p \neq t_s \in Tid$, since $\sigma \in \gamma(R)$, we have

$$(\sigma^L[t_p], \sigma^G, \sigma^L[t_s]), (\sigma^L[t_s], \sigma^G, \sigma^L[t_p]) \in R \tag{12}$$

There are three cases:

- $t_p = t_{ex}$ - thus $(\sigma^L[t_p], \sigma^G) \rightsquigarrow (l', \sigma'^G)$ where $l' = \sigma'^L[t_p]$. By definition of tr^{direct} from (12) we have $(\sigma'^L[t_p], \sigma'^G, \sigma'^L[t_s]) \in TR^{\#}(R)$
- $t_s = t_{ex}$ - thus $(\sigma^L[t_s], \sigma^G) \rightsquigarrow (l', \sigma'^G)$ where $l' = \sigma'^L[t_s]$. From (12) there are e_p, e_s s.t., $(\sigma^L[t_p], \sigma^G, e_p), (\sigma^L[t_s], \sigma^G, e_s) \in par(R)$ and $\sigma^L[t_s] \in e_p, \sigma^L[t_p] \in e_s$. Let $t \in Tid \setminus \{t_p, t_s\}$, then by definition of γ and $par(R)$ we have $\sigma^L[t] \in e_p \cap e_s$. Thus, by definition of $tr^{indirect}$ we have $(\sigma'^L[t_p], \sigma'^G, \sigma'^L[t_s]) \in TR^{\#}(R)$.
- $t_p \neq t_{ex}$ and $t_s \neq t_{ex}$ - By definition of γ, we have $(\sigma^L[t_{ex}], \sigma^G, \sigma^L[t_p]), (\sigma^L[t_{ex}], \sigma^G, \sigma^L[t_s]), (\sigma^L[t_p], \sigma^G, \sigma^L[t_{ex}]) \in R$. Thus, from (12) there are e_p, e_{ex} s.t., $(\sigma^L[t_p], \sigma^G, e_p), (\sigma^L[t_{ex}], \sigma^G, e_{ex}) \in par(R)$ and $\sigma^L[t_{ex}] \in e_p, \sigma^L[t_p] \in e_{ex}$. Furthermore, $\sigma^L[t_s] \in e_p \cap e_{ex}$. Thus, from (11) by definition of $tr^{indirect}$ we have $(\sigma'^L[t_p], \sigma'^G, \sigma'^L[t_s]) \in TR^{\#}(R)$.

Thus, for any $t_p \neq t_s$ we have $(\sigma'^L[t_p], \sigma'^G, \sigma'^L[t_s]) \in TR^{\#}(R)$, i.e., $\sigma' \in \gamma(TR^{\#}(R))$.

Note that the transformer is not guaranteed to be the most-precise transformer [7]. In terms of efficiency, we can see that the expensive part of the transformer is the application of $tr^{indirect}$, which operates over pairs of elements in $par(R)$, requiring a quadratic number of queries to tr.

In the next section, we describe two techniques for reducing this quadratic factor.

4 Efficiently Computing Interference

The techniques we use to accelerate the computation of interference are inspired by well-known techniques for inter-procedural analysis. The main observation behind these techniques is that the role of computing interference is to update the global store for the tracked thread. Any information that is not pertinent to this task can be disregarded. For the tracked thread, this means that any information in the local store that is not observable by the executing thread can be removed before the call to $tr^{indirect}$ and restored after the operation is complete. We call this *Summarizing Effects*. For the executing thread, we abstract away any information that does not affect the way that the global store is changed. We call this *Summary Abstraction*. We implement summarizing effects as an optimization (i.e., the result is an equivalent abstract transformer). We implement summary abstraction as an abstraction (i.e., some precision is lost, but given the presented heuristics we are still able to prove the property at hand). Both techniques retain tr^{direct} as is, and improve the bottleneck of the computation, i.e., $tr^{indirect}$. Summarizing effects operates on the tracked thread while summary abstraction operates on the executing thread.

4.1 Summarizing Effects

Our notion of summarizing effects is similar to a well-known technique in inter-procedural analysis of maintaining summary edges that represent the effect of procedures. Any information about the caller that is not visible to the callee can be abstracted at the call site and restored at the return site (see e.g., [17]). In this paper, we only abstract the local store of the tracked thread in this way. However, for heap-manipulating programs, any regions of the heap that are private to the tracked thread can be handled in a similar way. This is similar to the notion of procedure local heaps [25]. Another way to understand this optimization is by realizing that the abstraction of the secondary thread is coarser than the abstraction of the local store of the primary thread. The optimization takes advantage of this asymmetry by abstracting away any information on the tracked thread that is not observable in the environment of the executing thread. This information is restored after the quadratic part of the transformer has completed. In the examples, every box is a ci-disjunct where the thread on the left is the primary thread and the thread on the right is the secondary thread. Red nodes are deallocated.

Example 1. We demonstrate the summarizing effects technique on the buffer algorithm in Fig. 1. C1 is executing `dispose(c)`; C2 is the tracked thread. The abstraction used is similar to that of Sec. 2, except it is depicted graphically and for simplicity considers a bounded number of threads. The pre- and post-conditions are depicted in Fig. 2(a) and Fig. 2(b). Note that the ci-disjuncts in the third column differ only by C2's program counter. This is also true for the ci-disjuncts in the fourth column. Fig. 2(c) shows the ci-disjuncts that constitute the footprint of the the tracked thread's state. The program counter is left only in

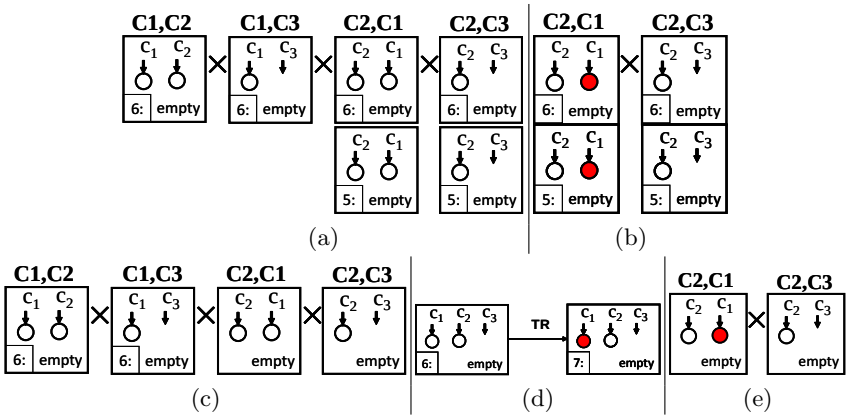

Fig. 2. Abstract states for summarizing effects

the frame. Fig. 2(d) shows the application of the transformer on the information gathered from all the ci-disjuncts considered. Fig. 2(e) depicts the states after they are projected back to the *CID* domain and before the frame is restored. Finally, we use the frame from Fig. 2(a) on Fig. 2(e) and get the abstract state in Fig. 2(b). □

The optimization amounts to determining for each the ci-disjunct of the tracked thread the *footprint*, which is needed for the computation of the transformer and the *frame* which is not. Intuitively, we split all the ci-disjuncts to equivalence classes according to their footprints. Representatives of these equivalence classes (computed by *fp*) are used for the second argument of $tr^{indirect}$. Later, we reconstruct the result by combining the original frame with the updated representatives. Formally, the right argument to the union in (7) is replaced with:

$$\bigcup_{t_f}\left(\{frame(t_t) \mid t_t \in par(R) \wedge fp(t_t) = t_f\} \otimes \bigcup_{t_e \in par(R)} tr^{indirect}(t_e, t_f)\right) \quad (13)$$

For simplicity of presentation, we formulate *fp* as choosing the representative ci-disjunct for all the ci-disjuncts that have the same footprint and *frame* as the identity function. Thus, the combination operation $\otimes$ is simply defined as:

$$R_{frm} \otimes R_{fp} = \{(\ell, g, \ell_e) \mid (\ell, \_, \_) \in R_{frm}, (\_, g, \ell_e) \in R_{fp}\} \quad . \quad (14)$$

Equations (15-17) give sufficient requirements for the transformer in (13) to be equivalent to the $tr^{indirect}$ part of (7).

$$\forall \ell_1, \ell_2, o_1, o_2, g \;.\; fp((\ell_1, g, o_1)) = fp((\ell_2, g, o_2)) \wedge \quad (15)$$
$$R(\ell_1, g) \neq \emptyset \wedge R(\ell_2, g) \neq \emptyset \Rightarrow R(\ell_1, g) = R(\ell_2, g)$$
$$\forall \ell_1, \ell_2, \ell, o, g \;.\; fp((\ell_1, g, o)) = fp((\ell_2, g, o)) \Rightarrow \ell_1 \in R(\ell, g) \Leftrightarrow \ell_2 \in R(\ell, g) \quad (16)$$
$$\forall t \;.\; fp(fp(t)) = fp(t) \quad (17)$$

The first two requirements specify that the abstraction cannot distinguish between local stores that have the same footprint. The third requirement states that *footprint* is idempotent. These requirements ensure that t_e is consistent with $fp(t_t)$ if and only if t_e and t_t are consistent (as checked in the second line of (9)).

Theorem 2. *If equations (15-17) hold, then (13) is equivalent to* $\bigcup_{t_e,t_t \in par(R)} tr^{indirect}(t_e, t_t)$.

Proof. Soundness

Let $(\ell_p, g', \ell'_s) \in \bigcup_{t_e,t_t \in par(R)} tr^{indirect}(t_e, t_t)$, i.e., there are $(\ell_e, g_e, R(\ell_e, g_e)), (\ell_t, g_t, R(\ell_t, g_t)) \in par(R)$ s.t.,
$(\ell_p, g', \ell'_s) \in tr^{indirect}(\ell_e, g_e, R(\ell_e, g_e)), (\ell_t, g_t, R(\ell_t, g_t))$.
Thus, by (9) we have $g_e = g_t$, $\ell_e \in R(\ell_t, g_t)$, $\ell_t \in R(\ell_e, g_e)$, $R(\ell_t, g_t) \cap R(\ell_e, g_t) \neq \emptyset$, $(\ell_e, g_e) \rightsquigarrow (\ell'_e, g')$, $\ell_p = \ell_t$ and $\ell'_s \in R(\ell_e, g_e) \cap R(\ell_t, g_t) \cup \{\ell'_e\}$.
Let $(\ell_f, g_t, e_f) = fp(t_t)$
By (17) we have $fp(\ell_f, g_t, e_f) = fp(t_t)$
and by (15) we have $e_f = R(\ell_t, g_t)$.
Thus, to show that $(\ell_f, g', \ell'_s) \in tr^{indirect}(t_e, (\ell_f, g_t, e_f))$ all we need to show is that $\ell_f \in R(\ell_e, g_e)$.
However, $fp(\ell_f, g_t, R(\ell_t, g_t)) = fp(\ell_t, g_t, R(\ell_t, g_t))$, thus from $\ell_t \in R(\ell_e, g_e)$ by (16) we have $\ell_f \in R(\ell_e, g_e)$.
Finally, from definition of $\otimes$ and *frame*, from $(\ell_f, g', \ell'_s) \in tr^{indirect}(t_e, (\ell_f, g_t, e_f))$ and $t_t \in par(R)$ we have $(\ell_t, g', \ell'_s) \in (13)$. □

Completeness

Let $(\ell_t, g', \ell'_s) \in (13)$.
From definition of $\otimes$ and *frame*, there are $(\ell_t, g_t, e_t), (\ell_e, g_e, e_e) \in par(R)$ and (ℓ_f, g_t, e_f) s.t., $(\ell_f, g_t, e_f) = fp(\ell_t, g_t, e_t)$ and $(\ell_t, g', \ell'_s) \in tr^{indirect}((\ell_e, g_e, e_e), (\ell_f, g_t, e_f))$.
Thus, by (9) we have $g_e = g_t$, $\ell_e \in e_f$, $\ell_f \in e_e$, $e_f \cap e_e \neq \emptyset$, $(\ell_e, g_e) \rightsquigarrow (\ell'_e, g')$, and $\ell'_s \in e_e \cap e_f \cup \{\ell'_e\}$.
By (17) we have $fp((\ell_f, g_t, e_f)) = fp((\ell_t, g_t, e_t))$, thus by (15) we have $e_f = e_t$.
Thus, to show that $(\ell_t, g', \ell'_s) \in tr^{indirect}((\ell_e, g_e, e_e), (\ell_t, g_t, e_t))$, all we need to show is that $\ell_t \in e_e$. However, $fp(\ell_f, g_t, e_f) = fp(\ell_t, g_t, e_t)$,
thus from $\ell_f \in e_e$ by (16) we have $\ell_t \in R(\ell_e, g_e)$.

4.2 Summary Abstraction

The *summary abstraction* technique abstracts t_1 into t'_1 in (13) or (7) such that

$$tr^{indirect}(t_1, t_2) \subseteq tr^{indirect}(t'_1, t_2) \ .$$

Specifically, we let t'_1 be the part of t_1 that is read or written by the executed statement (transition). However, this may still lead to a loss of precision, since some correlations may be lost in the consistency check (the second line of (9)). This is in contrast to summarizing effects where the specified requirements ensure that the consistency check would yield the same results.

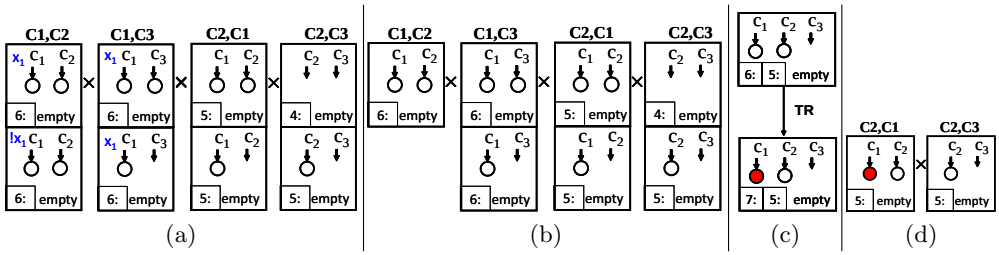

Fig. 3. Abstract states for summary abstraction

Example 2. We demonstrate the summary abstraction for single buffer algorithm in Fig. 1 with the commented lines added. Fig. 3 depicts a state where C1 is executing `dispose(c)`. The tracked thread is C2. Fig. 3(a) depicts the input ci-disjuncts. The ci-disjuncts in the leftmost column differ by the values of x_1 (which is not visible by C2). Fig. 3(b) represents the ci-disjuncts from Fig. 3(a) after the application of summary abstraction, which abstracts away x_1. Fig. 3(c) depicts the application of the transformer when considering all the relevant ci-disjuncts. Note that the left program counter is that of C1, and the right one is that of C2. Finally, Fig. 3(d) depicts the resulting ci-disjuncts.

This example also demonstrates the possible loss of precision due to this technique. Fig. 3(c) cannot be constructed from the ci-disjuncts in Fig. 3(a) due to conflicting values of x_1. However, after the application of summary abstraction, the values of x_1 are not considered and Fig. 3(c) can be constructed. □

5 Evaluation

We have evaluated our analysis on a number of highly concurrent state-of-the-art practical algorithms. Most of these algorithms have not been automatically verified before.

Our evaluation indeed confirms the need for extra precision in tracking thread correlations, without which the analysis loses crucial precision and fails to verify the specified properties.

We have implemented a tool based on TVLA [19] and its extension HeDec [20]. Our tool is generic in the sense that it supports diverse shape analyses. The input is a description of the algorithm and the shape analysis to be performed. We use a flow-sensitive and context-sensitive shape analysis. Context sensitivity is maintained by using call-strings. Our summarization consists of program locations, call-strings, and non-pointer local variables.

Tab. 1 summarizes the verified data structures and the speedups gained from the summarizing effects and summarizing abstraction techniques. Our benchmarks are all concurrent sets implemented by sorted linked lists.

We analyzed variants of these programs with intentionally added bugs (e.g., missing synchronization, changing the synchronization order). Our analysis found all these bugs and reported a problem (as expected, since our analysis is sound).

Table 1. Experiments performed on a machine with a 2.4Ghz Intel Q6600 32 bit processor and 4Gb memory running Linux with JDK 1.6

	Time (seconds)				Speedup		
Algorithm	Standard	Summar.	Abs.	Both	Summar.	Abs.	Both
Concurrent Set [21]	56,347	19,233	2,402	1,596	2.93	23.46	35.30
Optimized Concurrent Set	46,499	18,981	2,061	1,478	2.45	22.57	31.45
Lazy List Set [11]	963	679	460	390	1.42	2.09	2.47
CAS-based set [26]	13,182	8,710	4,223	2,975	1.51	3.12	4.43
DCAS-based set [26]	861	477	446	287	1.80	1.93	3.00
Hand over Hand Set [13]	686	577	444	398	1.19	1.54	1.73

The algorithm described in [21] uses two `CAS` operations in the `delete` method. The first operation logically marks the node as deleted and the second operation does the actual removal (moving the pointers). An interesting example of a mutation we tested is swapping the physical and logical removals. i.e., first remove the deleted node from the list by having its predecessor point to its successor and only then mark it as deleted. The analysis discovered that in such a case multiple nodes can be deleted by the `CAS` and not just the intended node.

A very significant speedup was gained in analyzing [21] and its optimized variant. We believe this is due to the following: (i) We optimize a quadratic algorithm, thus we expect to gain more as the examples become bigger. (ii) The complexity of the state that comes from the larger number of pointer variables and from the existence of Boolean fields makes the savings from ignoring the unchanged parts of the heaps as done in the summary abstraction much more significant. (iii) this algorithms is more complicated than the other examples and uses internal method calls. Summarizing effects significantly reduces the blow-up due to context sensitivity and summarizing abstraction is able to reduce blow-ups due to local Boolean fields.

6 Related Work

Process-Centric Abstraction. The thread correlation analysis presented in this paper falls within the general approach of reasoning about concurrent programs in terms of an *abstraction of the program state relative to a thread*, which is classic in work on program logic: assertions within the code of a thread refer to the state from that thread's perspective, and the thread's concurrent environment is over-approximated by, for instance, invariants [15, 23] or relations [16] on the shared state. This idea has also been used early on for automatic compositional verification [4]. More recently, this approach has led to the notion of thread-modular verification for model checking systems with finitely-many threads [8], and has also been applied more closely to our present domain of heap-manipulating programs with coarse-grained concurrency [9], and less automatically to fine-grained concurrency [3].

Abstract Interpretation with Quantified Invariants. The abstract states in our analysis are special case of quantified invariants. This approach has been

previously formalized in the work on Indexed Predicate Abstraction [18] and also appears in the work on Environment Abstraction [5, 6]. Indices, or free variables, in the indexed predicate abstraction work can range over anything, depending on the application. A similar quantified invariants approach has also been used in the analysis of heap properties [24] and properties of collections [10] in sequential programs.

References

1. E. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci*, 10(1):110–135, 1975.
2. J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.
3. C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, pages 233–248, 2007.
4. E. M. Clarke. Synthesis of resource invariants for concurrent programs. *ACM Transactions on Progrmming Languages and Systems*, 2(3):338–358, 1980.
5. E. M. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, 2006.
6. E. M. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, pages 33–47, 2008.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, New York, NY, 1979. ACM Press.
8. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.
9. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.
10. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
11. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.
12. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
13. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. M. Kaufmann, 2008.
14. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
15. C. A. R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques*, 1972.
16. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, 1983.
17. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, pages 125–140, 1992.
18. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *TOCL*, 9(1):1–29, 2007.
19. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

20. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS*, pages 363–377, 2008.
21. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
22. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
23. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, 1976.
24. A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.
25. N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309, 2005.
26. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.