Abstract Transformers for Thread Correlation Analysis

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@cs.ucla.edu
[3] Microsoft Research India, grama@microsoft.com

Abstract. We present a new technique for speeding up static analysis of (shared memory) concurrent programs. We focus on analyses that compute *thread correlations*: such analyses infer invariants that capture correlations between the local states of different threads (as well as the global state). Such invariants are required for verifying many natural properties of concurrent programs.

Tracking correlations between different thread states, however, is very expensive. A significant factor that makes such analysis expensive is the cost of applying abstract transformers. In this paper, we introduce a technique that exploits the notion of *footprints* and *memoization* to compute individual abstract transformers more efficiently.

We have implemented this technique in our concurrent shape analysis framework. We have used this implementation to prove 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 verified include memory safety, data structure invariants, partial correctness, and linearizability. Our empirical evaluation shows that our new technique reduces the analysis time significantly (e.g., by a factor of 35 in one case).

1 Introduction

This paper is concerned with analysis and verification of (shared memory) concurrent programs. We present a new technique that makes such analyses more efficient. The technique presented in this paper speeds up the verification significantly (e.g., reducing the verification time from $56,347$ seconds to $1,596$ seconds — a 35 fold speed-up — for one program).

One key abstraction technique for dealing with the state space explosion problem in analyzing concurrent programs is thread-modularity (see, e.g., [7]), which works by abstracting away correlations between the local states of different threads. Unfortunately, thread-modular analysis fails when the proof of a desired property relies on invariants connecting the local states of different threads, which is the case in several natural examples.

Thread-Correlation Analysis. Hence, we focus on analysis using abstractions that track correlations between pairs of (abstract) thread states (e.g., see [2, 5, 11]). The abstract domain elements, in our analysis, essentially represent invariants of the form $\forall t, e \,.\, \bigvee_{i=1}^{n} \varphi_i[t, e]$ 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 (describing some relation between the states of threads t and e). In our experience, we found such abstractions to be sufficiently precise for verifying the programs and properties of interest, but the corresponding analyses were quite time-consuming.

Abstract Transformers. The idea of using abstractions that correlate states of different threads is not new. In this paper we address the question of how to define precise, yet efficient, transformers for such domains, a question that has not been systematically studied before. This is, however, an important question because, as we found, the cost of applying abstract transformers is one of the main reasons why thread-correlation analyses are expensive. The abstract transformer corrresponding to a statement must determine how the execution of the statement by some thread affects the invariant computed by the analysis so far. The transformer must consider all possible (abstract) states of the executing thread, and identify the effect of the statement execution on all possible (abstract) states of any pair of threads. This introduces a non-linear factor that makes the transformer computation expensive. One of our key contributions is a set of techniques for computing the abstract transformer more efficiently.

Implementation and Evaluation. We have implemented the techniques described in this paper in our framework for concurrent shape analysis. We have used this implementation to verify properties, such as memory safety, preservation of data structure invariants, and linearizability [13], 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). 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, the data-structure can grow in an unbounded fashion and the number of threads concurrently updating it can also grow in an unbounded fashion.

Our empirical evaluation shows that our optimizations lead to significant reduction in the analysis time.

Main Contributions. Our contribution is not specific to 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. Specifically, we present our ideas using a simple abstract domain for concurrent programs. This domain formalizes our notion of thread correlation by abstracting concrete states, which capture correlations between the states of all threads, into

abstract states that capture only correlations between the states of every pair of threads. (Our implementation, however, realizes these ideas in a shape analysis and our empirical results concern this concurrent shape analysis.)

The main contributions of this paper are:

Sound Transformer We define a sound abstract post operator (transformer) for the new abstract domain 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 above ideas 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 [18], 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 all of the incorrect variations.

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 [6]. 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. Proofs and elaborations are found in [21].

2 Overview

In this section, we explain our approach informally, using an adaptation of a very simple example originally constructed to show the limitations of concurrent separation logic [19]. 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.

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.)

```
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

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, pc[t] $c[t]$ denote the program counter and 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. For simplicity, we assume that the set of local variables is the same for all threads (and is the union of local variables of all threads).

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 implies (1) is:

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

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, e.g., (v) ensures that (i) is *stable*: deallocations by different threads cannot affect it, if it already holds. Notice that the correlation invariants cannot be inferred by pure thread-modular approaches. Our work goes beyond pure thread-modular analysis [8] 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:

$$\forall t, e \,.\, \begin{pmatrix} t \neq e \wedge \\ pc[t] = 5 \wedge \\ c[t] \neq c[e] \wedge c[t] \neq b \wedge c[e] \neq b \wedge \\ a(c[t]) \wedge a(c[e]) \wedge a(b) \wedge \\ \neg empty \end{pmatrix} \bigvee \begin{pmatrix} t \neq e \wedge \\ pc[t] = 6 \wedge \\ c[t] \neq c[e] \wedge c[t] \neq b \wedge c[e] \neq b \wedge \\ a(c[t]) \wedge a(c[e]) \wedge a(b) \wedge \\ \neg empty \end{pmatrix} \bigvee \cdots \tag{4}$$

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 section 4, we show the conditions under which this technique can be used to make the transformer more efficient without affecting precision.

The *summary abstraction* 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. If such information is omitted from the footprint, we still get a sound transformer, though there may be a loss in precision. However, we found that this heuristic can be used to significantly reduce the computation time while maintaining a precise-enough abstraction. In general, an analysis designer can choose the information to be omitted from the footprint appropriately to achieve the desired tradeoff.

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 [6].

3.1 The Concrete Semantics (C, TR)

A concurrent 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 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 $\sigma^L[t \mapsto l]$ 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

$$TR = \{((\rho, g), (\rho[t \mapsto l'], g') \mid t \in Tid \,.\, (\rho[t], g) \rightsquigarrow (l', g'), \} \ . \tag{5}$$

3.2 The Abstract Semantics (CI, TR_{CI})

We now present an abstraction to deal with an unbounded number of threads. As we saw in Sec. 2, tracking information about a single thread in the style of thread-modular analysis [7] can be imprecise. 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

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

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 $\alpha_{CI}: C \to CI$ and the concretization $\gamma_{CI}: CI \to C$ are:

$$\alpha_{CI}(X) \equiv \bigcup_{\sigma \in X} \beta_{CI}(\sigma) \ , \qquad \gamma_{CI}(R) \equiv \{\sigma \mid \beta_{CI}(\sigma) \subseteq R\} \ .$$

Composing With Other Abstractions. Note that when *Loc* and *Glob* are finite sets, CI gives us a finite abstraction. In Section 3.3, we show how to compose the above abstraction with a subsequent abstraction to create other finite, tractable, abstract domains. For the sake of exposition, we first show how to define a sound transformer for the simple domain CI before we consider such an extension.

An Abstract Transformer. We define the abstract transformer $TR_{CI} : CI \to CI$ as follows:

$$TR_{CI}(R) \equiv \bigcup_{d \in R} tr_{CI}^{direct}(d) \cup TR_{CI}^{ind}(R) \ . \tag{7}$$

The function $tr_{CI}^{direct} : CID \to 2^{CID}$ 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 $tr_{CI}^{direct}(\phi)(t_p, t_s)$.

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

The function TR_{CI}^{ind} captures what we call the *indirect effects*: i.e., the effect of a transition by some thread t on ci-disjuncts whose primary thread is not t. As a first attempt, let us consider the following candidate definition for TR_{CI}^{ind}:

$$TR_{CI}^{ind}(R) = \bigcup_{(\ell_1, g, _) \in R, (\ell_2, g, \ell_3) \in R} \{(\ell_2, g', \ell_3) \mid (\ell_1, g) \rightsquigarrow (\ell'_1, g')\} \ .$$

Here, the transition $(\ell_1, g) \rightsquigarrow (\ell'_1, g')$ by one thread changes the global state to g'. As a result, a ci-disjunct (ℓ_2, g, ℓ_3) may be transformed to (ℓ_2, g', ℓ_3). While the above definition is a sound definition, it is not very precise. In fact, this definition defeats the purpose of tracking thread correlations because it does not check to see if the ci-disjunct (ℓ_2, g, ℓ_3) is "consistent" with the executing

ci-disjunct. We say that two ci-disjuncts x and y are consistent if there exists $\sigma \in \Sigma$ such that $\{x, y\} \subseteq \beta_{CI}(\sigma)$.

We first define some notation. Let *CIMap* denote $Loc \times Glob \rightarrow 2^{Loc}$. Define function $map : CI \rightarrow CIMap$ by $map(R) \equiv \lambda(\ell, g).\{\ell_e \mid (\ell, g, \ell_e) \in R\}$. Function *map* is bijective and *CI* and *CIMap* are isomorphic domains. 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 refer to an element of $par(R)$ as a *cluster*. A cluster (e.g., $(\ell_1, g, \{\ell_2, \ell_3\})$) represents a set of ci-disjuncts with the same first and second component (e.g., $\{(\ell_1, g, \ell_2), (\ell_1, g, \ell_3)\}$). $par(R)$ partitions a set of ci-disjuncts R into clusters.

We define TR^{ind}_{CI} as follows:

$$TR^{ind}_{CI}(R) = \bigcup_{c_e, c_t \in par(R)} tr^{indirect}_{CI}(c_e, c_t) \ . \tag{9}$$

$$\begin{aligned} & tr^{indirect}_{CI}((\ell_1, g_1, e_1), (\ell_2, g_2, e_2)) \equiv \\ & \qquad \textbf{if} \quad (g_1 = g_2 \wedge \ell_1 \in e_2 \wedge \ell_2 \in e_1) \\ & \qquad \textbf{then} \quad \{(\ell_2, g'_1, \ell_3) \mid (\ell_1, g_1) \rightsquigarrow (\ell'_1, g'_1), \ell_3 \in (e_1 \cap e_2) \cup \{\ell'_1\}\} \\ & \qquad \textbf{else} \quad \{\} \ . \end{aligned} \tag{10}$$

The first parameter of $tr^{indirect}_{CI}$ is a cluster representing the executing thread, and the second is a cluster representing thread(s) on which we compute the interference. We call this the *tracked thread*. The if-condition is a consistency check between two clusters. If the condition is false, then the clusters are inconsistent: i.e., a ci-disjunct from the first cluster and a ci-disjunct from the second cluster can not arise from the same concrete state. Hence, the transformer produces no new ci-disjunct.

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

Note that the transformer is not guaranteed to be the most-precise transformer [6]. In terms of efficiency, we can see that the expensive part of the transformer is the application of $tr^{indirect}_{CI}$, which operates over pairs of elements in $par(R)$, requiring a quadratic number of queries to tr.

3.3 Composing With Other Abstractions

In this section, we show how we can compose the abstraction *CI* defined in the previous section with other abstractions of *Loc* and/or *Glob* to create other, more tractable, abstract domains. This can be used to create finite state abstractions even if *Loc* and/or *Glob* are infinite-state.

Abstract Domain Let $(2^{Loc}, \alpha_P, \gamma_P, Loc_P)$ be a Galois Connection we want to use to abstract the primary thread's local state. Let $(2^{Glob}, \alpha_G, \gamma_G, Glob_G)$ be a Galois Connection we want to use to abstract the global state. Let $(2^{Loc}, \alpha_S, \gamma_S, Loc_S)$ be a Galois Connection for abstracting the secondary thread's local state.

Let $ACID = Loc_P \times Glob_G \times Loc_S$. Let $ACI = 2^{ACID}$, with the Hoare powerdomain ordering. (Since $ACID$ is already ordered, it is possible for several

elements of ACI to represent the same concrete set of states, which we allow as a convenience.) We use ACI as an abstraction of CI, with the abstraction function $\alpha_{ACI} : CI \rightarrow ACI$ defined by:

$$\alpha_{ACI}(S) = \{(\alpha_P(\{\ell_e\}), \alpha_G(\{g\}), \alpha_S(\{\ell_t\})) \mid (\ell_e, g, \ell_t) \in S\}.$$

We use the first local store (Loc_P) as the primary source of information about locals.The second local store (Loc_S) is used primarily to express correlation invariants (and to check for consistency between different ci-disjuncts). (This can be seen in the definition of the abstract transformers presented earlier.) Thus, in practice, the domain Loc_P is richer and captures more information than the domain Loc_S.

Abstracting basic transitions Recall that $\rightsquigarrow \subseteq (Loc \times Glob) \times (Loc \times Glob)$ represents a single step transition by a thread. Let $\rightsquigarrow_a \subseteq (Loc_P \times Glob_G) \times (Loc_P \times Glob_G)$ be a sound abstraction of this relation: i.e., abusing notation, $\rightsquigarrow_a$ should satisfy $\gamma \circ \rightsquigarrow_a \supseteq \rightsquigarrow \circ \gamma$. More precisely, we want

$$\begin{aligned} &\{(x, y) \mid (\ell_e, g_a) \rightsquigarrow_a (\ell'_e, g'_a), x \in \gamma_P(\ell'_e), y \in \gamma_G(g'_a)\} \ \supseteq \\ &\quad \{(\ell', g') \mid \ell \in \gamma_P(\ell_e), g \in \gamma_G(g_a), (\ell, g) \rightsquigarrow (\ell', g')\} \ . \end{aligned}$$

Abstract Transformer We now present a sound abstract transformer TR_{ACI} for the domain ACI, which is very similar to the transformer for domain CI defined in the previous section. In particular, equations 7 and 9 defining the transformer remain the same as before. The function tr^{direct} is the same as before, except that $\rightsquigarrow$ is replaced by $\rightsquigarrow_a$ as follows:

$$tr^{direct}_{ACI}(\ell_p, g, \ell_s) \equiv \{(\ell'_p, g', \ell_s) \mid (\ell_p, g) \rightsquigarrow_a (\ell'_p, g')\}. \tag{11}$$

The definition of function $tr^{indirect}$, however, is a bit more complex.

We define the abstract transformer in terms of a sound consistency-check operation for comparing elements across different abstract domains as follows, where the indices i and j are either P or S. Let $\approx_{i,j} \subseteq Loc_i \times Loc_j$ be such that

$$\gamma_i(x) \cap \gamma_j(y) \neq \{\} \Rightarrow x \approx_{i,j} y.$$

We define a corresponding relation $\overline{\in}_{i,j} \ \subseteq \ Loc_i \times 2^{Loc_j}$ by

$$x \ \overline{\in}_{i,j} \ S \text{ iff } \exists y \in S. x \approx_{i,j} y.$$

We will omit the indices i, j if no confusion is likely. Informally, $x \approx y$ indicate that x and y may represent the same concrete (local) state.

$$\begin{aligned} &tr^{indirect}_{ACI}((\ell_1, g_1, e_1), (\ell_2, g_2, e_2)) \equiv \\ &\quad \text{let } g = g_1 \sqcap g_2 \text{ in} \\ &\quad \{(\ell_2, g', \ell_s) \mid \ell_1 \ \overline{\in} \ e_2, \ell_2 \ \overline{\in} \ e_1, (\ell_1, g) \rightsquigarrow_a (\ell'_1, g'), (\ell_s \in e_2 \wedge \ell_s \ \overline{\in} \ e_1) \vee (\ell_s = \ell'_1)\} \end{aligned} \tag{12}$$

Note that the above definition closely corresponds to equation 10, except that $\in$ is replaced by a corresponding sound approximation $\overline{\in}$.

Theorem 2. *The abstract transformer TR_{ACI} is sound.*

4 Efficiently Computing Indirect Effects

As mentioned earlier, the expensive part of computing transformers is the computation of indirect effects. We now present a couple of techniques for making this computation more efficient. (The techniques we present are inspired by well-known optimization techniques used in inter-procedural analysis.)

4.1 Abstracting The Executing Cluster

The computation of indirect effects, $tr^{indirect}(c_e, c_t)$, determines how a single transition by a thread e (described by a cluster c_e) transforms another cluster c_t (describing the state of another thread t). However, not all of the information in the clusters c_e and c_t is *directly* used in the computation of indirect effects. This lets us abstract away some of the information, say, from c_e to construct c'_e and compute $tr^{indirect}(c'_e, c_t)$ to determine the indirect effects. We refer to c'_e as the *footprint* of c_e. This helps speed up the computation because different clusters $c_1, \cdots, c_k$ can have the same footprint c_f: in such a case, it is sufficient to compute the indirect effect due to the single footprint c_f, instead of the k different clusters. We call this technique *summary abstraction*. The notion of a footprint can be applied to ci-disjuncts (as in the example below) or, equivalently, to clusters (as in our formalism).

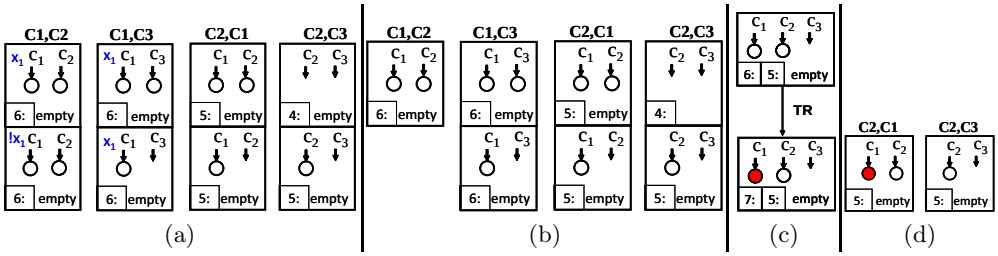

Fig. 2. Abstract states for summary abstraction

Example 1. We first illustrate the value of this technique using an example from the single buffer algorithm in Fig. 1 with the commented lines added. In Fig. 2 every box represents a ci-disjunct. We consider states with 3 threads $C1$, $C2$ and $C3$. The boxes under the column labelled Ci, Cj represents ci-disjuncts with primary thread Ci and secondary thread Cj. However, not all ci-disjuncts are shown. The abstraction includes the program counter of the primary thread (shown in the lower left corner) but not of the secondary thread. c_i represents the value of c in thread Ci: it points to nothing if it is null (the initial value), and a hollow circle if it points to an allocated object, and a filled circle if it points to an object that has been deallocated. c_i and c_j point to different circles

to indicate that they are not aliased. The value of x of thread $C1$ is shown as x_1 (true) or $!x_1$ (false).

We now consider the transition due to the execution of the statement `dispose(c)` by C1. The tracked thread is C2. Fig. 2(a) depicts some of the input ci-disjuncts. The ci-disjuncts in the leftmost column differ only in the value of x_1. Fig. 2(b) represents the ci-disjuncts from Fig. 2(a) after the application of summary abstraction, which abstracts away x_1 (since the executed statement does not use x). As a result, the two ci-disjuncts of the first column are now represented by a single footprint. Fig. 2(c) depicts the application of the transformer to the state obtained by combining two ci-disjuncts (the executing ci-disjunct $C1, C2$ and the tracked ci-disjunct $C2, C3$). Note that the left program counter is that of C1, and the right one is that of C2. Finally, Fig. 2(d) depicts the resulting ci-disjuncts where C2 is the primary thread. □

We now present a modified version of the transformer TR^{ind}_{CI} that incorporates this optimization. Let *Cluster* denote the set of all clusters. Our modified definition is parameterized by a function $fp_E : Cluster \rightarrow Cluster$ that abstracts away some information from the executing cluster c_e to construct its "footprint" c'_e. However, the only property required of fp_E for soundness is that $fp_E(x) \sqsupseteq x$ (for all x), where the ordering $\sqsupseteq$ is the ordering on the domain ACI (treating a cluster as a set of ci-disjuncts). Given a set S of clusters, let $\overline{fp_E}(S)$ denote $\{fp_E(c) \mid c \in S\}$. Given such a function, we define:

$$TR^{ind}_E(R) = \cup_{e \in \overline{fp_E}(par(R))} \cup_{t \in par(R)} tr^{indirect}_{ACI}(e, t).$$

Theorem 3. *If for all x, $fp_E(x) \sqsupseteq x$, then TR^{ind}_E is a sound approximation of TR^{ind}_{CI}: $TR^{ind}_E(R) \sqsupseteq TR^{ind}_{CI}(R)$.*

Note that analysis designers can define fp_E so that the above technique is an optimization (with no loss in precision, i.e., $tr^{indirect}_E(e, t) = tr^{indirect}(e, t)$), or they can define a more aggressive absraction function that leads to greater speedups with a potential loss in precision. Thus, the parameter fp_E acts as a precision-efficiency tradeoff switch.

In our implementation, we used a definition of fp_E such that $fp_E(c_e)$ is the part of c_e that is read or written by the executed statement (transition).

4.2 Exploiting Frames for The Tracked Cluster

The technique described above for the executing cluster can be used for the tracked cluster as well, but with some extensions. The modified technique involves decomposing the tracked cluster c_t into two parts: the part c^{fp}_t that is directly used to determine the indirect effect, and the part c^{fr}_t that is neither used nor modifed by the indirect effect. We refer to c^{fp}_t as the *footprint* and to c^{fr}_t as the *frame*. Unlike in the earlier case, we require the frame now because the goal of the indirect effect is to determine the updated value of the tracked cluster. We call this technique *summarizing effects*.

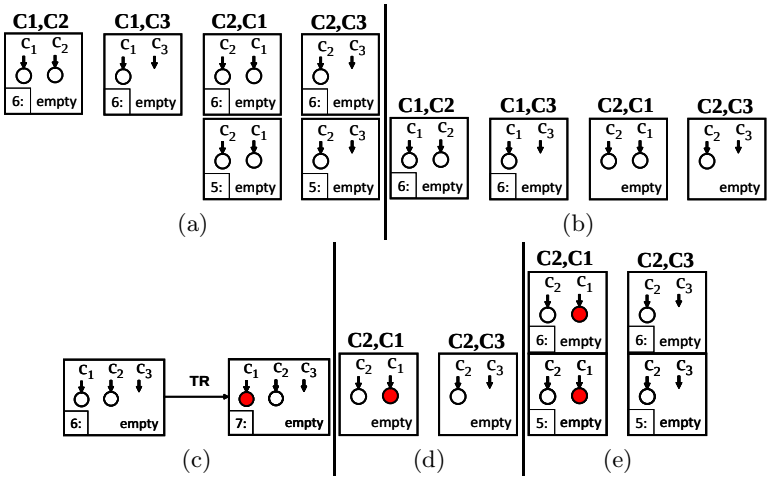

Fig. 3. Abstract states for summarizing effects

Example 2. We demonstrate the summarizing effects technique on the algorithm in Fig. 1. A set of ci-disjuncts are depicted in Fig. 3(a). The notation used is the same as that in Fig. 2. Consider the execution of `dispose(c)` by C1. C2 is the tracked thread. Note that the ci-disjuncts in the third column of Fig. 3(a) differ only by C2's program counter. This is also true for the ci-disjuncts in the fourth column.We define the frame of a ci-disjunct to consist of its program counter and the footprint to consist of everything else. Fig. 3(b) shows ci-disjuncts after we replaced the tracked ci-disjunctsby their footprints. Fig. 3(c) shows the application of the transformer on the information gathered from all the ci-disjuncts considered. Fig. 3(d) 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. 3(a) on Fig. 3(d) and get the abstract state in Fig. 3(e). □

We now present a modified form of our earlier transformer, parameterized by a couple of functions. Given functions $\mathit{frame}_T : \mathit{Cluster} \rightarrow \mathit{Cluster}$ and $\mathit{fp}_T : \mathit{Cluster} \rightarrow \mathit{Cluster}$ we define:

$$\begin{aligned} TR_T^{ind}(R) = {} & \text{let } C = \mathit{par}(R) \text{ in} \\ & \text{let } TC = \{(\mathit{fp}_T(c), \mathit{frame}_T(c)) \mid c \in C\} \text{ in} \\ & \bigcup_{e \in C} \bigcup_{(p,r) \in TC} (r \sqcap tr_{ACI}^{indirect}(e,p)). \end{aligned}$$

Note that the above technique is similar in spirit to the inter-procedural analysis technique of abstracting away information about the caller that is not visible to the callee when the callee is analyzed and restoring this information at the return site (see e.g., [14]). Furthermore, to achieve an efficiency gain with this definition, we need to *save and reuse* the value of $tr_{ACI}^{indirect}(e,p)$ when different clusters have the same footprint p. This is analogous to the technique of *memoization* in interprocedural analysis. In our context, we can capture this by rewriting the

last line of the above definition as follows:

$$\bigcup_{p \in dom(TC)} (\cup_{e \in C} tr^{indirect}_{ACI}(e,p)) \otimes \{r \mid (p,r) \in TC\}$$

where $dom(TC) = \{p \mid (p,r) \in TC\}$ and $S_1 \otimes S_2 = \{x \sqcap y \mid x \in S_1, y \in S_2\}$..

Theorem 4. *Let $frame_T$ and fp_T satisfy (for all x, y) (a) $fp_T(x) \sqsupseteq x$, (b) $frame_T(x) \sqsupseteq x$, and (c) $frame_T(x) \sqsupseteq tr^{indirect}(y,x)$. Then, TR^{ind}_T is a sound approximation of TR^{ind}_{CI}: $TR^{ind}_T(R) \sqsupseteq TR^{ind}_{CI}(R)$.*

In our implementation, the local store of the tracked thread is abstracted into the frame and omitted from the footprint. (For heap-manipulating programs, any regions of the heap that are private to the tracked thread can be handled similarly.)

5 Evaluation

We have implemented our ideas in a framework for concurrent shape analysis. Our implementation may be seen as an instantiation of the framework in Sec. 3.3, obtained by composing the thread correlation abstraction (in Sec. 3.2) with TVLA [16], an abstraction for shape analysis, and its extension HeDec [17]. We use a flow-sensitive and context-sensitive shape analysis. Context sensitivity is maintained by using call-strings.

We have used our implementation to verify properties such as memory safety, data structure invariants, and linearizability for several highly concurrent state-of-the-art practical algorithms.

Our evaluation indeed confirms the need for extra precision in tracking thread correlations, without which the analysis fails to verify the specified properties. Our evaluation also confirms the value of the optimizations described in this paper. Tab. 1 summarizes the verified data structures and the speedups gained from the use of summarizing effects and summarizing abstraction techniques. Our benchmarks are all concurrent sets implemented by sorted linked lists. More details can be found in an accompanying technical report [21].

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).

Note that the speedup is particularly high for the first two programs (namely [18] and its optimized variant). These are also the examples where the analysis took most time. We believe this confirms the non-linear speedups achieved by the techniques: we optimize a quadratic algorithm, thus we expect to gain more as the examples become bigger. These algorithms were expensive to analyze since they were interprocedural, and used a large number of pointer variables and Boolean fields. Summarizing effects significantly reduced the blow-up due to context sensitivity and summarizing abstraction was able to reduce blow-ups due to local Boolean fields.

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 [18]	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 [10]	963	679	460	390	1.42	2.09	2.47
CAS-based set [23]	13,182	8,710	4,223	2,975	1.51	3.12	4.43
DCAS-based set [23]	861	477	446	287	1.80	1.93	3.00
Hand over Hand Set [12]	686	577	444	398	1.19	1.54	1.73

6 Related Work

In this paper we have presented techniques for speeding up analysis (abstract interpretation) of concurrent programs. One of the recent works in this area is [7], which presents the idea of thread-modular verification for model checking systems with finitely-many threads. However, in many natural examples tracking correlations between different thread states is necessary, and our work focuses on abstractions that track such correlations. The work on Environment Abstraction [5] presents a *process-centric abstraction* framework that permits capturing thread-correlations. Our abstract domain is similar in spirit. The novelty of our work, however, is in the definition of the transformers and its optimizations, and its application to concurrent shape analysis.

Resource invariants [19] enable the use of thread-modular analysis to verify concurrent programs without tracking thread correlations. One of the challenges in automating such verification is inferring resource invariants. [8] and [3] present techniques for inferring resource invariants. These techniques apply to programs with coarse-grained concurrency. Our implementation, however, handles *fine-grained* concurrency, including non-blocking or lock-free algorithms. [4], and more recently [22], present semi-automated algorithms for verifying programs with fine-grained concurrency, using a combination of separation-logic, shape abstractions, and rely-guarantee reasoning. While powerful, this approach requires programmers to provide annotations describing the abstract effects of the atomic statements of a thread.

The abstract states in our analysis represent quantified invariants. Quantified invariants have been previously used in Indexed Predicate Abstraction [15] and in Environment Abstraction [5]. A similar quantified invariants approach has also been used in the analysis of heap properties [20] and properties of collections [9] in sequential programs.

The work described in this paper is a continuation of [2]. The new contributions of this paper are: we introduce a new, simple, abstract domain for capturing correlations between pairs of threads and systematically study the question of defining a precise, yet efficient, transformer for this domain, and present new 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 (compared to our earlier work) while still being able to prove the same properties.

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, D. Distefano, and V. V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, 2009.
4. C. Calcagno, M. J. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, pages 233–248, 2007.
5. 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.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, New York, NY, 1979. ACM Press.
7. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, pages 213–224, 2003.
8. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, pages 266–277, 2007.
9. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
10. 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.
11. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, pages 1–13, 2004.
12. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. M. Kaufmann, 2008.
13. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.
14. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, pages 125–140, 1992.
15. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *TOCL*, 9(1):1–29, 2007.
16. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
17. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. In *SAS*, pages 363–377, 2008.
18. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
19. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
20. A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.
21. M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Efficiently inferring thread correlations, Jan. 2009. TR-09-59203.
22. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, pages 335–348. Springer-Verlag, 2009.
23. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.