Semantic Reduction of Thread Interleavings in Concurrent Programs

Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta

NEC Laboratories America, 4 Independence Way, Princeton, NJ
{kahlon,srirams,agupta}@nec-labs.com

Abstract. We propose a static analysis framework for concurrent programs based on reduction of thread interleavings using sound invariants on the top of partial order techniques. Starting from a product graph that represents transactions, we iteratively refine the graph to remove statically unreachable nodes in the product graph using the results of these analyses. We use abstract interpretation to automatically derive program invariants, based on abstract domains of increasing precision. We demonstrate the benefits of this framework in an application to find data race bugs in concurrent programs, where our static analyses serve to reduce the number of false warnings captured by an initial lockset analysis. This framework also facilitates use of model checking on the remaining warnings to generate concrete error traces, where we leverage the preceding static analyses to generate small program slices and the derived invariants to improve scalability. We describe our experimental results on a suite of Linux device drivers.

1 Introduction

Concrete error traces are critical for effective debugging of software. Unfortunately, generating error traces for concurrency related bugs is notoriously hard. One of the key reasons for this is that concurrent programs are behaviorally complex due to the many possible interleavings between threads. These interleavings make concurrent programs hard to analyze.

Verification and analysis for concurrent systems is currently a very active area of research due to the multi-core revolution. Testing, static analysis, and model checking have all been explored but not without some drawbacks. Testing has clearly been the most effective debugging technique for sequential programs. However, the multitude of interleavings among threads makes it hard to provide meaningful coverage guarantees for concurrent programs. Furthermore, replayability of the bugs detected through testing is a challenge.

Static analysis techniques have been successful for detecting standard concurrency bugs such as data races and deadlocks [28,25,24,11,26,15]. However, the large number of bogus warnings generated by static analyzers remains a drawback. Most static analyses ignore conditional statements, focusing mostly on the syntactic reachability of a pair of control locations rather than semantic reachability. This places the burden of sifting the true bugs from the false warnings on the programmer, which leads to poor productivity.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 124–138, 2009.

Model checking [5,3,6] has the advantages of systematic state space exploration, and can produce concrete error traces. However, the state explosion problem severely limits its scalability on large real-life concurrent programs.

In this paper, we propose a general framework for analysis of concurrent programs, based on semantic reduction in the thread interleavings by use of sound invariants. We first utilize *partial order reduction* (POR) techniques and constraints from synchronization primitives to construct a *transaction graph*. The transaction graph effectively captures the relevant thread interleavings for performing a sound static analysis. We then derive *sound invariants* by using *abstract interpretation* over this transaction graph. These invariants are used to further refine the transaction graph by removing unreachable nodes. This can lead to larger transactions, i.e. a reduction in the thread interleavings. The removal also facilitates the discovery of stronger invariants on the reduced graph. This process can be iterated until convergence, i.e. until no more nodes can be removed and no better invariants computed. The transaction graph is central in our approach: at any stage it provides a current snapshot of thread interleavings needed for sound static analysis. It also allows us to stage various analyses to achieve scalability, such that less precise but cheaper techniques are used first to refine the transaction graph, to enable application of more precise techniques later. Our focus is on successive reduction of thread interleavings, which is a primary source of complexity in the analysis of concurrent programs.

```
a0:  void Alloc_Page() {
a1:  a := c;
a2:  pt_lock(&plk);
a3:  if (pg_cnt ≥ LIMIT) {
a4:      pt_wait(&pg_lim,&plk);
a5:      incr (pg_count);
a6:      pt_unlock(&plk);
a7:      sh1 := sh;
a8:  } else {
a9:      pt_lock(&count_lock);
a10:     pt_unlock(&plk);
a11:     page := alloc_page();
a12:     sh := 5;
a13:     if (page)
a14:         incr( pg_count );
a15:     pt_unlock(&count_lock);
a16: end-if
a17: b := a +1;
a18: end-function
```

```
b0:  void Dealloc_Page() {
b1:  pt_lock(&plk);
b2:  if (pg_count == LIMIT)
b3:      sh := 2;
b4:      decr (pg_count);
b5:      sh1 := sh;
b6:      pt_notify(& pg_lim, & plk);
b7:      pt_unlock(& plk);
b8:  } else {
b9:      pt_lock(&count_lock);
b10:     pt_unlock(&plk);
b11:     decr(pg_count);
b12:     sh := 4;
b13:     pt_unlock(&count_lock);
b14: end-if
b15: end-function
```

Fig. 1. Example concurrent program consisting of two functions

Motivating Example
The concurrent program shown in Fig. 1 comprises of *multiple threads* executing the *Alloc_Page* and *Dealloc_Page* routines. We assume that each statement shown is executed atomically. For clarity, all shared variable accesses and synchronization constructs **pt_lock**, **pt_unlock**, **pt_wait** and **pt_notify** are highlighted.

The shared variable *sh* is written to at location **a**12, **b**3 and **b**12. Since the set of locks held at **a12** and **b3** (viz., $\{count\_lk\}$ and $\{plk\}$, respectively) are disjoint, the pair $(\mathbf{a}12, \mathbf{b}3)$ normally constitutes a data race warning according to lockset-based race detection techniques.

When *some thread* reaches the control location **b**3, we automatically establish the interval invariant ψ_3 : **pg_count** $\in [\text{LIMIT}, +\infty)$, regardless of the control location of the other threads. This invariant is quite challenging to establish (the reader is invited to attempt). Specifically, establishing ψ_3 requires us to reason about the sequence of synchronizations between threads as well as the conditional branches involved.

Likewise, we establish the invariant φ_{12} : **pg_count** $\in (-\infty, \text{LIMIT})$ when *some thread* reaches the control location **a**12, regardless of the location of the other threads. Once again, this invariant is non-trivial to establish.It involves conditional branches as well as thread synchronization.

Using the invariants computed, we conclude that locations **a**12 and **b**3 are *not simultaneously reachable*. In other words, the lockset-based method yields a bogus warning that can be eliminated by means of the automatically computed invariants ψ_3, φ_{12}.

2 Program Model

We consider concurrent imperative programs comprising threads that communicate using shared variables and synchronize with each other using standard primitives such as locks, rendezvous, etc.

Program Representation. Each thread in a concurrent program $T_i : \langle F_i, \mathsf{e}_i, G_i, L_i \rangle$ consists of procedures F_i, entry procedure $\mathsf{e}_i \in F_i$, a set of global variables G and thread local variables v. Each procedure $p \in F$, is associated with a tuple of formal arguments $\mathsf{args}(p)$, a return type t_p, local variables $L(p)$, and a control flow graph (CFG). Each procedural CFG $\langle N(p), E(p), \text{action} \rangle$ consists of a set of nodes $N(p)$ and a set of edges $E(p)$ between nodes in $N(p)$. Each edge $m \rightarrow n \in E(p)$ is associated with an *action* that is an assignment, a call to another procedure, a return statement, a conditional guard, or a synchronization statement. The actions in the CFG for a procedure p may refer to variables in the set $G \cup \mathsf{args}(p) \cup L(p)$.

A multi-threaded program Π consists of a set of threads $T_1, \ldots, T_N$ for some fixed $N > 0$ and a set of shared variables S. Note that every shared variable $s \in S$ is a global variable in each thread T_i.

Threads synchronize with each other using standard primitives like locks, rendezvous and broadcasts. Locks are standard primitives used to enforce mutually exclusive access to shared resources. They occur very commonly in many parallel

programming paradigms and are widely used to enforce thread synchronization. Rendezvous are motivated by `wait/notify` primitives of Java and condition variables in the `POSIX` thread library. Rendezvous find limited use in applications such as browsers, device drivers and scientific programs. Broadcasts are seldom seen in practice.

2.1 Preliminaries

Static program analysis can be used to compute sound invariant assertions that characterize the values of program variables at different program points. Since our approach involves reasoning with infinite sets of integers and reals, the *abstract interpretation* framework forms an integral part. We provide a concise description of abstract interpretation in this section. A detailed presentation is available elsewhere [9,8].

Let Ψ be a CFG of a sequential (single threaded) program. Concurrent programs are treated in Section 4. For simplicity, we assume that Ψ consists of a single procedure that does not involve calls to other procedures. Procedures, including recursive procedures, can be handled by implementing a context-sensitive program analysis. All variables involved in Ψ are assumed to be integers. Pointers and arrays are lowered into integers using a process of memory modeling followed by abstraction. Such an abstraction is implemented by the F-Soft framework [14]. As mentioned earlier, each edge in the CFG is labeled with an assignment or a condition.

An *abstract domain* Γ consists of assertions drawn from a selected assertion language which form a lattice through the partial order $\sqsubseteq$ modeling logical inclusion between assertions. Each object $a \in \Gamma$ represents a set of program states $[[a]]$. For the analysis, we require the following operations to be defined over Γ:

(a) Join: Given $a_1, a_2 \in \Gamma$, the join $a = a_1 \sqcup a_2$ is the smallest abstract object a w.r.t $\sqcup$ such that $a_1 \sqsubseteq a,\ a_2 \sqsubseteq a$.
(b) *Meet:* The meet $a_1 \sqcap a_2$ corresponds to the logical conjunction.
(c) *Abstract post condition* $post_\Gamma$ models the effect of assignments.
(d) *Inclusion test* $\sqsubseteq$ to check for the termination.
(e) *Widening* operator ∇ to force convergence for the program loops.
(f) *Projection* operator $\exists$ removes out-of-scope variables.
(g) *Narrowing* operator $\triangle$ is used for solution improvement.

Given a program Ψ and an abstract domain Γ, we seek a map $\pi : L \mapsto \Gamma$ that maps each CFG location $\ell \in L$ to an abstract object $\pi(\ell)$. Such a map is constructed iteratively by the *forward propagation* iteration used in data-flow analysis:

$$\pi^0(\ell) = \begin{cases} \top, \text{ if } \ell = \ell_0 \\ \bot, \text{ otherwise} \end{cases} \textbf{ and } \pi^{i+1}(\ell) = \bigsqcup_{e:\ m \to \ell} post_\Gamma(\pi^i(m), e)\,.$$

If the iteration converges, i.e, $\pi^{i+1}(\ell) \sqsubseteq \pi^i(\ell)$ for all $\ell \in L$ for some $i > 0$, π^{i+1} is the result of our analysis. However, unless the lattice Γ is of finite height or satisfies the *ascending chain condition*, convergence is not always guaranteed. On the

other hand, many of the domains commonly used in verification do not exhibit these conditions. Convergence is forced by using *widening* and *narrowing* [9].

Using abstract interpretation, we may lift dataflow analyses to semantically rich domains such as *intervals*, *polyhedra*, *shape graphs* and other domains to verify sophisticated, data-intensive properties. Intervals, Octagons and Polyhedra are instances of numerical domains that may be used to reason about the numerical operations in the program.

The interval domain consists of assertions of the form $x_i \in [\ell, u]$, associating each variable with an interval containing its possible values. The domain operations for the interval domain such as join, meet, post condition, inclusion, etc. can be performed efficiently (see [7] for details). However, the interval domain is *non-relational*. It computes an interval for each variable that is independent of the intervals for the other variables. As a result, it may fail to handle many commonly occurring situations that require more complex, relational invariants. The polyhedral domain [10] computes expressive linear invariants and is quite powerful. However, this power comes at the cost of having exponential time domain operations such as *post condition*, *join*, *projection* and so on.

The octagon domain due to Miné [22] extends the interval domain by computing intervals over program expressions such as $x - y$, $x + y$ and so on, for all possible pairs of program variables. The domain can perform operations such as post, join and projection efficiently using a graphical representation of the constraints and a canonical form based on shortest-path algorithms.

3 Transaction Graphs

In this section, we describe how we capture thread interleavings in the form of a transaction graph. We also describe our procedure for constructing an initial transaction graph by utilizing partial order reduction techniques and constraints due to synchronization primitives. The transaction graph will be used as a basis for the static analysis technique to be presented in the next section.

Let $\mathcal{P}$ be a concurrent program comprised of threads T_1,..., T_n and let N_i and E_i be the set of control locations and transitions of the control flow graph (CFG) of T_i, respectively. We write $\ell_i \rightsquigarrow m_i$ to denote a path from l_i to m_i.

Definition 1 (Transaction Graph). *A transaction graph $\Pi_{\mathcal{P}}$ of $\mathcal{P}$ is defined as $\Pi_{\mathcal{P}} = (N_{\mathcal{P}}, E_{\mathcal{P}})$, where $N_{\mathcal{P}} \subseteq N_1 \times ... \times N_n$ and $E_{\mathcal{P}} \subseteq N_{\mathcal{P}} \times N_{\mathcal{P}}$. Each edge of $\Pi_{\mathcal{P}}$ represents the execution of a sequence of statements by a thread T_i. Specifically, an edge is of the form $(l_1, \ldots, l_i, \ldots, l_n) \rightarrow (l_1, \ldots, m_i, \ldots, l_n)$,, such that there is a path in T_i from $l_i \rightsquigarrow m_i$. Such an edge represents an execution of a program segment in thread T_i.*

A transaction graph considers a subset of the possible tuples of control states concurrently reachable by n different threads. Edges of the graph consist of a sequence of moves by a single thread. The *product graph* is a transaction graph $\Pi_{\mathcal{P}} \equiv \otimes_i N_i$ consisting of the *cartesian product* of the control locations in each thread, and each edge representing the execution of a single statement by a single

thread. In the presence of rendezvous and broadcast actions on the edges, the definition may be updated to necessitate synchronous rendezvous to happen in two consecutive moves.

The cartesian product graph consists of a superset of all the concurrent control states that need to be considered for the analysis of a given program. It is reduced to a transaction graph of manageable size by using partial order reduction (POR) [13] wherein we remove *redundant* control states that produce a result consistent with some other interleaving (see below).

Shared Variable Identification: The first step towards building a transaction graph consists of automatically and conservatively identifying shared variables. In general, shared variables are either global variables of threads, aliases thereof and pointers passed as parameters to API functions. Since global variables can be accessed via local pointers, alias analysis is key for identifying shared variables.

Our shared variable detection technique uses *update sequences*(Cf. [16]) to track aliasing information as well as whether a variable is being shared. An update sequence of assignments from pointers p to q along a sequence $\{l_j\}$ of consecutive thread locations is of the form $l_0 : p_1 = p, l_1 : p_2 = q_1, ..., l_{i-1} : p_i = q_{i-1}, l_i : p_{i+1} = q_i, ..., l_k : p = q_k$, where q_i is aliased to p_i between locations l_{i-1} and l_i. Then p is aliased to a shared variable if there exists an update sequence starting at a global variable or an escaped variable. Update sequences can be tracked via a simple and efficient dataflow analysis (see [17] and [16] for details).

Static Partial Order Techniques: The transaction graph is computed using partial order reduction (POR) which has been used extensively in model checking [13]. POR exploits the fact that concurrent computations are partial orders on operations of threads on shared variables. Therefore, instead of exploring all interleavings that realize this partial order, it suffices to explore just a few (ideally just one).

In this setting, we use POR over the complete product graph $\mathcal{P}$ of the thread CFGs, instead of over the state space of the concurrent program. We consider a pair of statements st_1 and st_2 of threads T_1 and T_2, respectively, to be *dependent* if (i) st_1 and st_2 access a common shared variable (including variables used for synchronization), and (ii) at least one of the accesses is a write operation. Whereas, this is a purely syntactic characterization, we may use semantic considerations to obtain a more refined dependence relation.

When constructing the transaction graph, a key goal is to maximize the lengths of the resulting transactions in the presence of scheduling constraints imposed by synchronization primitives. The general problem of delineating transactions of maximal length for threads synchronizing via locks and rendezvous is presented elsewhere [18]. For completeness, we present a simple transaction delineation algorithm for the easier case of two threads synchronizing via locks. Our algorithm is broadly similar to that of Godefroid [13] and is shown in Alg. 1. It is a worklist algorithm that traverses through the global control states of the concurrent program and computes their successors. The pair (l_1, l_2) represents

Algorithm 1. Transaction Delineation based on Static POR

1: Initialize $W = \{(in_1, in_2)\}$, where in_j is the initial state of thread T_j, and *Processed* to $\emptyset$.
2: **repeat**
3: Remove a state (l_1, l_2) from W and add it to *Processed*
4: **if** neither l_1 nor l_2 is a shared object access **then**
5: let $Succ_i = \{(m_1, m_2) |$ *where (a)* $m_{i'} = l_{i'}$ *with* $i' \in \{1, 2\}$ *and* $i' \neq i$*, and (b)* m_i *is CFL-reachable from* l_i *via a local path* x *of thread* T_i *such that* m_i *is the first shared object access encountered along* $x\}$
6: Set $Succ = Succ_1 \cup Succ_2$
7: **else if** l_1 is a shared object access of *sh*, say, **then**
8: **if** m_2 is a statement accessing *sh* that is CFL-reachable from l_2 via a path x of T_2 such that (a) l_1 conflicts with m_2, and (b) no lock held at l_1 is acquired (and possibly released) along x **then**
9: Let $Succ_1 = \{(m_1, l_2) |$ *where* m_1 *is CFL-reachable from* l_1 *via a path* y *such that* m_1 *is the first shared object access along* y *after* $l_1\}$
10: Let $Succ_2 = \{(l_1, m_2)\}$
11: Set $Succ = Succ_1 \cup Succ_2$
12: **else**
13: $Succ = \{(m_1, l_2) |$ m_1 *is CFL-reachable from* l_1 *via a path* x *such that* m_1 *is the first shared object access along* x *after* $l_1\}$
14: **end if**
15: **else if** l_2 is a shared object access of *sh*, say, **then**
16: compute *Succ* as in steps 7-14 with the roles of l_1 and l_2 reversed
17: **end if**
18: Add all states of *Succ* not in *Processed* to W.
19: **until** W is empty

a global control state in which threads T_1 and T_2 are at control locations l_1 and l_2, respectively.

At the initial state (in_1, in_2) of the given concurrent program, we let each thread execute until it encounters its first shared object access (steps 4-6). Next, in order to compute the successors of a global state (l_1, l_2) we need to decide whether a context switch is required at location l_i of thread T_i. A conflict analysis is carried out to determine whether T_1 is currently accessing a shared object *sh* at the location l_1 and whether thread T_2 starting at l_2 can reach a location m_2 which accesses *sh* and is dependent with l_1, i.e., l_1 conflicts with m_2. If so, then we explore interleavings wherein T_1 executes l_1 first leading to the set of successors $Succ_1$ (step 9) and those wherein T_2 executes the path leading to m_2 before l_1 is executed leading to the set of successors $Succ_2$ (step 10). On the other hand, if the synchronization primitives ensure that no path starting at l_2 leads to T_2 accessing *sh* then no context switch is required at l_1 and the successors of (l_1, l_2) result only from executing transitions of T_1 (step 13).

A crucial difference between Alg. 1 and the classical POR algorithm is that in deducing reachability of m_i from l_i we need to take recursion into account, i.e., we need to deduce CFL-reachability of m_i from l_i in thread T_i.

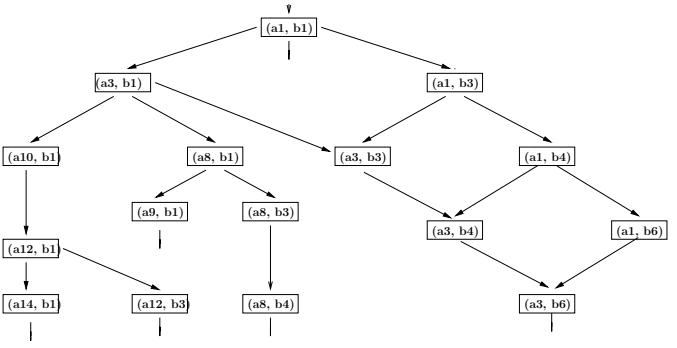

Fig. 2. Transaction Graph

Example 1. In Figure 1, the statements at locations **a**12 and **b**12 are dependent since the outcome for *sh* differs based on their relative order of execution. Statements **b**5 and **a**14 are independent, the actions performed by them commute. Statements **b**4 and **b**11 of Fig. 1 both decrease a shared variable pg_count by 1. Semantically, these statements are independent, since their order of execution does not affect the reachable states.

Synchronization Constraints. We now illustrate utilization of synchronization constraints in our running example from Figure 1. Fig. 2 shows a portion of the transaction graph obtained after pruning nodes with non-disjoint locksets. Examples of tuples pruned for non-disjoint lockset include (**a**5,**b**4), (**a**6,**b3**), (**a**3,**b**2) and so on which are mutually excluded by the lock `plk`. Note that a static lockset analysis on each thread allows us to avoid generating such pairs in the first place.

The send and wait statements **b6** and **a4**, respectively, enforce that in a 2−threaded execution, **b**7 must be executed before **a**4. Therefore all the nodes in the set $\{(\mathbf{a}5, \mathbf{b}7), (\mathbf{a}6, \mathbf{b}7), (\mathbf{a}7, \mathbf{b}7)\}$ are all unreachable. Again, computing the synchronization language at each location visited by each thread will ensure that such tuples are never considered.

The POR and synchronization-based constraints yield transactional sequence of actions for each thread so that an interleaving with another thread *need not* be considered during the execution of this sequence, while still capturing all the feasible interleavings. However, such sequences may be conditional on the location of the other thread. For instance, statements {**b**1, . . . ,**b7** } are transactional provided the `Alloc_Page` thread resides in one of the locations {**a**1,. . . ,**a**7}. This allows us to *compact* many locations in the product graph into one single location in the initial transaction graph.

To summarize, partial order techniques and synchronization constraints are used to construct a transaction graph over the global control states. Although our presentation described these steps separately, our implementation derives an initial set of thread conflicts based on both considerations, which drives the construction of the transaction graph. The initial transaction graph is used to compute sound invariants, described in the next section.

4 Generation of Sound Invariants

We now apply abstract interpretation over the initial transaction graph to compute sound invariants. The key difference between abstract interpretation for sequential programs as opposed to transaction graphs, is the treatment of invariants that relate thread-local variables to shared variables. Such invariants are quite useful, since threads typically perform many actions involving locals and globals in an atomic section. However, local variables of a thread T_1 may not be shared directly with another thread T_2. A general solution is to consider a cartesian product of the abstract domain with itself, yielding tuples of invariant facts $\langle \varphi_1, \ldots, \varphi_n \rangle$ for each node of the transaction graph, wherein $\varphi_i[S, L_i]$ relates the global variables S with the local variables L_i of the i^{th} thread.

For the sake of simplicity, we restrict our attention to two threads (i.e, $n = 2$). The invariant tuple annotating a node $\langle \ell_i, m_j \rangle$ of the transaction graph is denoted $\langle \varphi_i, \psi_j \rangle$, wherein $\varphi_i[S, L_1]$ relates the values of the shared and local variables for thread T_1 and $\psi_j[S, L_2]$ for thread T_2. Since the programs communicate through shared variables, we require that the set of shared variables described by any pair φ_i, ψ_j are the same: $(\exists L_1)\ \varphi_i \equiv (\exists L_2)\psi_j$.

This *consistency condition* will be maintained in our analysis. However, maintaining this condition through the abstract interpretation process is tricky.

Example 2. Consider a simple action by a thread T_1, $a : \ \ell_1 \xrightarrow{x\ :=\ 10} \ell_2$ which updates a shared variable x to 10, while thread T_2 remains in location m. This corresponds to a move in the transaction graph from $\langle \ell_1, m \rangle \rightarrow \langle \ell_2, m \rangle$.

Let $\langle (x = 5), (x = l, l = 5) \rangle$ be the assertion labeling the node (ℓ_1, m). After a move by the first thread, the assertion labeling the node (ℓ_2, m) should be $\langle (x = 10), (l = 5, x = 10) \rangle$. Note that to maintain consistency, we are forced to update the invariant for thread T_2 even if T_2 did not perform any action.

A similar situation arises at the "join-nodes" of the transaction graph. Due to the consistency condition, these nodes do not act as true join nodes as in the sequential case. We address these difficulties by introducing a "meld" operator in the abstract domain, in order to maintain the consistency condition.

Melding: As mentioned earlier, the shared variables S may be modified by executing some transaction edge $n_i \rightarrow n_k$ by thread T_i, updating the component φ_i of the invariant tuple. However, this may violate the consistency requirements w.r.t the invariant tuples corresponding to the other threads. To enforce this consistency, we introduce an operator $\mathsf{meld}(\varphi_k, \psi_j)$ that forces the global state values represented by assertion ψ_j to coincide with those in φ_k.

Definition 2 (Meld). *Let $\alpha[S, L_1]$ and $\beta[S, L_2]$ be assertions over shared and local variables for each thread. The assertion $\gamma : \mathsf{meld}(\alpha, \beta)$ is such that (a) $(\exists L_1)\alpha \equiv (\exists L_2)\gamma$ and $(\exists G)\beta \models (\exists G)\gamma$, and (b) the operator must be entry-wise monotonic. I.e., if $\alpha_1 \models \alpha_2$ and $\beta_1 \models \beta_2$ then*

$$\mathsf{meld}(\alpha_1, \beta_1) \models \mathsf{meld}(\alpha_2, \beta_1), \ \ \mathsf{meld}(\alpha_1, \beta_1) \models \mathsf{meld}(\alpha_1, \beta_2)\,.$$

The result γ of $\mathsf{meld}(\alpha, \beta)$ over-approximates the global variable values described by α and the local variable values described by β.

The design of a melding operator is specific to the underlying abstract domain. A simple melding operator can be constructed for most abstract domains using projection and works for domains wherein the conjunction $\sqcap$ coincides with the logical conjunction $\wedge$ (i.e, Moore closed domains). Formally, we have $\mathsf{meld}(\varphi, \psi) : (\exists L_1)\varphi \wedge (\exists G)\psi$.

Post Condition: An elementary step in the fixed point computation consists of propagating an assertion pair $\langle \varphi_i, \psi_j \rangle$ across an edge $n_i \rightarrow n_k$ of one of the threads. Let φ_k denote the result of the post-condition $post(\varphi_i, n_i \rightarrow n_k)$. In practice, however, a move by a thread $n_i \rightarrow n_k$ in the transaction graph may represent the execution of a (possibly atomic) program segment in the corresponding thread consisting of numerous basic blocks. Therefore, the "post" needs to be computed using a thread-local abstract interpretation of the segment corresponding to the edge.

The effect of executing a thread edge $n_i \rightarrow n_k$ starting from the node $\langle n_i, m_j \rangle$, labelled by the assertion $\langle \varphi_i, \psi_j \rangle$, yields the assertion pair $\langle \varphi_k, \psi'_j \rangle$ wherein: $\varphi_k : post(\varphi_i, n_i \rightarrow n_k)$, $\psi'_j : \mathsf{meld}(\varphi_k, \psi_j)$. Formally, we use a propagation operator $\mathsf{propagate}$ to model the effect of executing a transaction n_i across an edge $n_i \rightarrow n_k$: $\mathsf{propagate}(\langle \varphi_i, \psi_j \rangle, n_i \rightarrow n_k) = \langle \varphi_k, \psi'_j \rangle$.

Our goal is to produce a map η labeling each node of the transaction graph $\langle n_i, m_j \rangle$ with a pair of assertions $\eta(n_i, m_j) : \varphi_i, \psi_j$ such that $\varphi_i[S, L_1]$ relates the shared variables S with the local variables L_1, and similarly ψ_j. Secondly, each of the assertions $\langle \varphi_i, \psi_j \rangle$ holds, whenever the individual thread controls simultaneously reside in the nodes $\langle n_i, m_j \rangle$.

Formally, for any edge $\langle n_i, m_j \rangle \rightarrow \langle n_k, m_j \rangle$, we require that $\mathsf{propagate}(\eta(n_i, m_j), n_i \rightarrow n_k) \models \eta(n_k, m_j)$. A symmetric condition needs to hold for moves of the thread T_2. The map η can be constructed using forward propagation on a transaction graph G using $\mathsf{propagate}$ as the post-condition.

Loops and Recurrences: A cycle in the transaction graph corresponds directly to a loop or a recursive procedure in one or more of the threads. Such cycles are handled naturally in our abstract interpretation scheme using widening. Specifically, widening is performed conservatively at each node of the form $(l_1, \ldots, l_k)$ such that for some component l_i there exists a back-edge of the form $m_i \rightarrow l_i$ in the CFG of T_i. For example, in the case of cycles arising due to loops l_i would be a loop head. Standard iteration schemes known for sequential programs can be used for analyzing transaction graphs.

5 Refinement of Transaction Graphs

We use the abstract interpretation framework described in the previous section to automatically derive sound invariants for the concurrent program. In practice, we use abstract domains of increasing precision ranges, octagons, and polyheda to derive more accurate invariants.

Algorithm 2. Refinement of Transaction Graph

1: Construct an initial transaction graph G_π^0 by using partial order techniques and synchronization constraints.
2: **repeat**
3: Compute range, octagonal, and polyhedral invariants over G_π^i And prune paths from G_π^i resulting in H_π^i.
4: Compute a new product transaction graph G_π^{i+1} based on the thread conflicts and synchronization constraints in H_π^i.
5: **until** $G_\pi^{i+1} = G_\pi^i$
6: **return** G_π^i

Invariant-based slicing of thread conflicts. At each stage, we use the derived invariants to show the unreachability of code segments, e.g. in conditional branches. If these unreachable code segments contain shared variable accesses, this can lead to a reduction in the conflicts between threads, thereby allowing larger transactions in individual threads. We call this a *refinement* of the transaction graph, since it provides a more accurate view of thread interleavings required for analysis. Such refinement helps to improve accuracy of subsequent analysis by discounting spurious thread interference from unreachable code segments, while also improving scalability due to smaller transaction graphs that result from a smaller number of interleavings and larger individual transactions.

Iterative refinement. In general, we can iteratively refine the transaction graph by alternately leveraging conflict analysis (using partial order techniques and synchronization constraints) and sound invariants until we reach a fix-point, where the transaction graph cannot be refined any more.

This iterative procedure for refining a transaction graph is shown in Figure 2. The initial transaction graph construction utilizes POR and synchronization constraints (described in Section 3). This *bootstraps* the iterative process. This initial step is critical for making the computation of sound invariants scale (described in Section 4). This is because the initial transaction graph over global control states is much smaller than a naive product graph over individual statements in threads. Furthermore, the capturing of POR and synchronization constraints drastically reduces the number of interleavings considered by our invariant computation. This effectively, makes the invariants stronger.

6 Applications

The transaction graph constructed by exploiting synchronization constraints and sound invariants can be used for various analyses and verification applications on concurrent programs. These include concurrent pointer alias analysis, model checking, etc. Once the product transaction graph has been computed, any dataflow analysis of concurrent programs can be carried out sequentially over the nodes of this graph. From a model checking perspective, the product

transaction graph encodes all the context switches that need be explored. When carrying out partial order reduction over the state space during model checking (described later in this section), we allow context switching only at transaction boundaries defined by the transaction graph.

In the remainder of this section, we describe a specific application of our approach for detection of data race bugs in concurrent programs. There have been many successful efforts based on static analysis [28,25,24,11,26,15]. These approaches, however, may generate a large number of bogus warnings. Model checking [5,3,6] has the advantage that it can produce concrete error traces and does not rely on the programmer to inspect the warnings and decide whether they are true bugs or not. However, the state explosion problem severely limits its scalability, especially on large real-life concurrent programs.

Classic static data race warning generation has three main steps. First, control locations with shared variable accesses are determined in each thread. Next, the set of locks held at each of these locations of interest are computed, using lockset analysis. Pairs of control locations in different threads where (i) the same shared variable is accessed, (ii) at least one of the accesses is a write operation, and (iii) disjoint locksets are held, constitute a potential data race site and a warning is issued.

Since dataflow analysis for concurrent programs is undecidable in general, typical static data race detection methods ignore conditional statements in the threads and perform thread-local analysis only. Indeed, a pair of control locations (c_1, c_2) marked as a potential data race site may simply be unreachable in any run of the given concurrent program.

We use the static analysis framework proposed in this paper to check the reachability of the pair of control locations (c_1, c_2) appearing in such warnings. If the pair is statically unreachable, then the warning is bogus, and can be eliminated. The combined use of synchronization constraints and sound invariants provide cheaper methods than model checking to check the pairwise (un)reachability of c_1 and c_2, while providing more accuracy than existing static analysis methods for data race detection. In fact, one can use any of the existing fast methods to generate the initial set of data race warnings, and use our techniques to automatically reduce the number of warnings.

We also leverage the final transaction graph generated in our framework to perform model checking, for producing concrete error traces for the remaining warnings. Details of our symbolic (SAT-based) model checking techniques for concurrent programs are described in our previous work [19]. The additional benefit is that our transaction graph already captures reductions in thread interleavings that would have otherwise been explored during model checking. We also use slicing on the transaction graph to generate smaller models for specific warnings, by inlining the functions in the specific contexts and slicing away the rest. We can also use the derived invariants to prune the search space during model checking. The combined effect is to improve the viability of model checking on concurrent programs.

7 Experimental Results

We applied the proposed static analysis framework for reducing data race warnings generated by an initial lockset-based analysis on a suite of Linux device drivers with known data races. The results are shown in Table 1, where columns 4 and 5 report the number of warnings (#Warn) and time taken (seconds), respectively, by the lockset-based static analysis. Column 6 reports the number of warnings after reduction by using our invariant-based static analysis, with the time taken (in seconds) reported in Column 7. This analysis is successful in reducing the number of warnings to a more manageable level within a few minutes. As an additional benefit, we may now apply techniques such as model checking on the few remaining warnings. Column 8 reports the number of warnings for which our model checking procedure [19] was successful in generating a concrete error trace, with the final unresolved number of warnings reported in Column 9.

Table 1. Results for Static Reduction of Data Race Warnings

Driver	KLOC	# *ShVars*	#Warn.	Time (secs)	#After Invar	Time (secs)	#Wit.	#Unknown
pci_gart	0.6	1	1	1	1	4	0	1
jfs_dmap	0.9	6	13	2	1	52	1	0
hugetlb	1.2	5	1	3.2	1	0.9	1	0
ctrace	1.4	19	58	6.7	3	143	3	0
autofs_expire	8.3	7	3	6	2	12	2	0
ptrace	15.4	3	1	15	1	2	1	0
raid	17.2	6	13	1.5	6	75	6	0
tty_io	17.8	1	3	4	3	11	3	0
ipoib_multicast	26.1	10	6	7	6	16	4	2
TOTAL			99		24		21	3

Table 2. Results for Model Checking Data Race Warnings. All timings are in seconds and memory in MBs.

Witness #	Symbolic POR+BMC			Witness #	Symbolic POR+BMC		
	Depth	Time	Mem		Depth	Time	Mem
jfs_dmap : 1	10	0.02	59	raid : 4	34	4.15	61
ctrace : 1	8	2	62	raid : 5	40	9.30	59
ctrace : 2	56	10 hr	1.2G	raid : 6	70	70	116
ctrace : 3	92	2303	733	tty_io : 1	34	0.82	5.7
autofs$_e$xpire : 1	9	1.14	60	tty_io : 2	32	9.69	14
autofs$_e$xpire : 2	29	128	144	tty_io : 3	26	31	26
ptrace : 1	111	844	249	ipoib_multicast : 1	6	0.1	58
raid : 1	42	26.13	75	ipoib_multicast : 2	8	0.1	59
raid : 2	84	179	156	ipoib_multicast : 3	4	0.1	58
raid : 3	44	32.19	87	ipoib_multicast : 4	14	0.3	59

The detailed results for model checking are shown in Table 2, where we report the depth at which the bug is found, and the time and memory used by our model checking procedure that uses symbolic POR with SAT-based BMC. We were able to generate a concrete error trace for the known data race in all but one example. This is mainly due to the small sliced models we generated by using warning-specific static information, even for large drivers (such as *ipoib_multicast*). Thus, our static analysis framework enables scalable model checking for larger concurrent programs.

8 Related Work and Conclusions

We have presented a general framework for static analysis of concurrent programs, where we use partial order reduction and synchronization constraints to capture a reduced set of thread interleavings, on which we derive sound invariants by using abstract interpretation to perform further reduction. We described an application of this framework to reduce the number of data race warnings, and to enable the application of model checking to find concrete error traces.

Our work is related to prior work on verification of concurrent programs that attempts to get around the undecidability barrier by considering restricted models of synchronization and communication [1,12] or by bounding the number of context switches [27,2,23,21]. There are also other recent efforts to leverage sequential analysis in concurrent settings [4,20]. Our approach also exploits specific patterns of synchronization, but our main focus is on deriving *sound* invariants for reduction in thread interleavings, by lifting abstract interpretation techniques to the concurrency setting. Since thread interleavings are a primary source of complexity in concurrent programs, this provides us further opportunities to apply more precise analyses, including model checking.

References

1. Atig, M.F., Bouajjani, A., Touili, T.: On the reachability analysis of acyclic networks of pushdown systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 356–371. Springer, Heidelberg (2008)
2. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
3. Brat, G., Havelund, K., Park, S., Visser, W.: Model checking programs. In: ASE (2000)
4. Chugh, R., Voung, J.W., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: PLDI, pp. 316–326 (2008)
5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logics of Programs, pp. 52–71 (1981)
6. Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H.: Bandera: Extracting finite-state models from java source code. In: ICSE (2000)
7. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the Second International Symposium on Programming (1976)
8. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to Abstract interpretation, invited paper. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)

9. Cousot, P., Cousot, R.: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: ACM POPL, January 1978, pp. 84–97 (1978)
11. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: SOSP (2003)
12. Farzan, A., Madhusudan, P.: Causal dataflow analysis for concurrent programs. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 102–116. Springer, Heidelberg (2007)
13. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
14. Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: F-SOFT: Software verification platform. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 301–306. Springer, Heidelberg (2005)
15. Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and precise datarace detection for multithreaded object-oriented programs. In: PLDI (2002)
16. Kahlon, V.: Bootstrapping: A Technique for Scalable Flow and Context- Sensitive Pointer Alias Analysis. In: PLDI (2008)
17. Kahlon, V., Yang, Y., Sankaranarayanan, S., Gupta, A.: Fast and Accurate Static Data-Race Detection for Concurrent Programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 226–239. Springer, Heidelberg (2007)
18. Kahlon, V.: Exploiting Program Structure for Tractable Dataflow Analysis of Concurrent Programs (2008), `kahlonnec-labs.com`
19. Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006)
20. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
21. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
22. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
23. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: PLDI (2008)
24. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: POPL (2007)
25. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI (2006)
26. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: PLDI (2006)
27. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
28. Sterling, N.: Warlock: A static data race analysis tool. In: USENIX Winter Technical Conference (1993)