Compositional Bitvector Analysis For Concurrent Programs With Nested Locks

Azadeh Farzan Zachary Kincaid

University of Toronto

Abstract. We propose a new technique to perform bitvector data flow analysis for concurrent programs. Our algorithm works for concurrent programs with nested locking synchronization. We show that this algorithm computes precise solutions (meet over all paths) to bitvector problems. Moreover, this algorithm is compositional: it first solves a local (sequential) data flow problem, and then efficiently combines these solutions leveraging reachability results on nested locks [5,6]. We implemented our algorithm on top of an existing sequential data flow analysis tool, and demonstrated that the technique performs and scales well.

1 Introduction

Writing concurrent software is difficult and error prone. In principle, static analysis offers an appealing way to mitigate this situation, but dealing with concurrency remains a serious obstacle. Theory and practice of automatically and statically determining dynamic behaviours of concurrent programs lag far behind those for sequential programs. Enumerating all possible interleavings to perform flow-sensitive analyses is infeasible. It is imperative to formulate compositional analysis techniques and proper behaviour abstractions to tame this so-called *interleaving explosion* problem. We believe that the work presented in this paper is a big step in this direction. We propose a compositional algorithm to compute *precise* solutions for bitvector problems for a general and useful class of concurrent programs.

Data flow analysis has proven to be a useful tool for debugging, maintaining, verifying, optimizing, and testing sequential software. Bitvector analyses (also known as the class of gen/kill problems) are a very useful subclass of data flow analyses. Bitvector analyses have been very widely used in compiler optimization. There are a number of applications for precise concurrent bitvector analyses. To mention a few, reaching definitions analysis can be used for precise slicing of concurrent programs with locks, which can be used as a debugging aid for concurrent programs[1]. Both problems of race and atomicity violation detection can be formulated as variations of the reaching definitions analysis. Lighter versions of information flow analyses may also be formulated as bitvector analyses. Precision will substantially decrease the number false positives reported by any of the above analyses.

[1]Concurrent program slicing has been discussed previously [10], but to our knowledge there is no method up until now that handles locks precisely.

There is an apparent lack of techniques to precisely and efficiently solve data flow problems, and more specifically bitvector problems for concurrent programs with dynamic synchronization primitives such as locks. The source of this difficulty lies in the lack of a precise and efficient way to represent program paths. programs. Control flow graphs (CFG) are used to represent program paths for most static analyses on *sequential* programs, but concurrent analogs to CFGs suffer major disadvantages. Concurrent adaptations of CFGs mainly fall into two categories: (1) Those obtained by taking the Cartesian product of CFGs for individual threads, and removing inconsistent nodes. These product CFGs are far too large (possibly even infinite) to be practical. (2) Those obtained by taking the union of the CFGs for individual threads, adding inter-thread edges, and performing a may-happen-in-parallel heuristic to get rid of infeasible paths. These union CFGs may still have an abundance of infeasible paths and cannot be used for precise analyses.

Bitvector problems have the interesting property that solving them precisely is possible without analyzing whole program paths. The key observation is that, in a *forward may* bitvector analysis, a fact f is true at a control location c iff there exists a path to c on which f is generated and not subsequently killed; what happens "before" f is generated, is irrelevant. Therefore, bitvector problems only require reasoning about *partial* paths starting at a generating transition. For programs with *only* static synchronization (such co-begin/co-end), bitvector problems can be solved with a combination of sequential reasoning and a light concurrent predecessor analysis [8]. Under the concurrent program model in [8], a fact f holds at a control location c if and only if the control location c' at which f is *generated* is an *immediate* concurrent predecessor of c. Therefore, it is sufficient to only consider concurrent paths of length *two* to compute the precise bitvector solution. Moreover, the concurrent predecessor analysis is very simple for co-begin/coend synchronization.

Dynamic synchronization (which was not handled in [8]) reduces the number of feasible concurrent paths in a program, but unfortunately makes their *finite* representation more complex. This complicates data flow analyses, since a precise concurrent data flow analysis must compute the *meet-over-all-feasible-paths* solution, and the analysis should only consider *feasible* paths (that are no longer limited to paths of length two). Evidence of the degree of difficulty that dynamic synchronization introduces is the fact that pairwise reachability (which can be formulated as a bitvector problem) is undecidable for recursive programs with locks. It is however decidable [6] if the locks are acquired in a *nested* manner (i.e. locks are released in the reverse order that were acquired). We use this result to introduce *sound and complete* abstractions for the set of feasible concurrent paths, which are then used to compute the meet-over-all-feasible-paths solution to the class of bitvector analyses.

We propose a *compositional* (and therefore scalable) technique to *precisely* solve bitvector analysis problems for concurrent programs with nested locks. The analysis proceeds in three phases. In the first phase, we perform the sequential bitvector analysis for each thread individually. In the second phase, we use a

sequential data flow analysis to compute an abstract semantics for each thread based on an abstract interpretation of sequential trace semantics. We then combine the abstract semantics for each pair of threads to compute a second set of data flow facts, namely those who reach concurrently. In the third phase, we simply combine the results of the sequential and concurrent phases into a *sound and complete* final solution for the problem. This procedure is quadratic in the number of threads and exponential (in the worst case) in the number of shared locks in the program; however, we do not expect to encounter even close to the worst case in practice. In fact, in our experiments the running time follows a growth pattern that almost matches that of sequential programs. Our approach avoids the limitations typically imposed by concurrent adaptations of CFGs: it is scalable and compositional, in contrast with the product CFG; and it is precise, in contrast with union CFGs.

We have implemented our algorithm on top of the C language front-end CIL [17], which performs the sequential data flow analyses required by our algorithm. We show through experimentation that this technique scales well and has running time close to that of sequential analysis in practice.

Related Work. Program flow analysis was originally developed for sequential programs to enable compiler optimizations [1]. Although the majority of flow analysis research has been focused on sequential software [19,14,20], flow analysis for concurrent software has also been studied. Flow-insensitive analyses can be directly adapted into the concurrent setting. Existing flow-sensitive analyses [13,15,16,18,21] have at least one of the following two restrictions: (a) the programs they handle have extremely simplistic concurrency/synchronization mechanisms and can be handled precisely using the union of control flow graphs of individual programs, or (b) the analysis is sound but not complete, and solves the data flow problem using heuristic approximations.

RADAR [2] attempts to address some of the problems mentioned above, and achieves scalability and more precision by using a race detection engine to kill the data flow facts generated and propagated by the sequential analysis. RADAR's degree of precision and performance depends on how well the race detection engine works. We believe that although RADAR is a good practical solution, it does not attempt to solve the real problem at hand, nor does it provide any insights for static analysis of concurrent programs.

Knoop et al [8] present a bitvector analysis framework which comes closest to ours in that it can express a variety of data flow analysis problems, and gives sound and complete algorithms for solving them. However, it cannot handle dynamic synchronization mechanisms (such as locks). This approach has been extended for the same restricted synchronization mechanism to handle procedures in [3,4,22] and generalizations of bitvector problems in [9,22].

Foundational work on nested locks appears in [5,6]. Recently, analyses based on this work have been developed, including [7] and [11]. Notably, the authors of [7] detect violations of properties that can be expressed as phase automata, which is a more general problem than bitvector analysis. However, their method is not tailored to bitvector analysis, and is not practically viable when a "full"

solution (a solution for every fact and every control location) to the problem is required, which is often the case.

2 Preliminaries

A concurrent program $\mathcal{CP}$ is a pair $(\mathcal{T}, \mathcal{L})$ consisting of a finite set of threads $\mathcal{T}$ and a finite set of locks $\mathcal{L}$. We represent each thread $T \in \mathcal{T}$ as a control flow automaton (CFA). CFAs are similar to a control flow graphs, except actions are associated with edges (which we will call transitions) rather than nodes. Formally, a CFA is a graph (N_T, E_T) with a unique entry node s_T and a function $stmt_T : E_T \to Stmt$ that maps edges to program statements. We assume no two threads have a common node (edge), and refer to the set of all nodes (edges) by N (E). In the following, we will often identify edges with their corresponding program statements. CFA statements execute atomically, so in practice we split non-atomic statements prior to CFA construction.

For each lock $l \in \mathcal{L}$, we distinguish two synchronization statements $acq(l)$ and $rel(l)$ that acquire and release the lock l, respectively. Locks are the only means of synchronization in our concurrent program model. For a path π through thread T, we let $\text{Lock-Set}_T(\pi)$ denote the set of locks held by T after executing π[2].

A local run of thread T is any finite path starting at its entry node; we refer to the set of all such runs by $\mathcal{R}_T$. A run of $\mathcal{CP}$ is a sequence $\rho = \rho_1 \dots \rho_n \in E^\star$ of edges such that:

i) ρ projected onto each thread T (denoted by ρ_T), is a local run of T
ii) There exists no point p along ρ at which two threads T, T' hold the same lock $(\nexists T, T', p.T \neq T' \wedge \text{Lock-Set}_T((\rho_1 \cdots \rho_p)_T) \cap \text{Lock-Set}_{T'}((\rho_1 \cdots \rho_p)_{T'}) \neq \emptyset)$.

We use $\mathcal{R}_{\mathcal{CP}}$ to denote the set of all runs of $\mathcal{CP}$ (just $\mathcal{R}$ when there is no confusion about $\mathcal{CP}$). For a sequence $\rho = \rho_1 \dots \rho_n \in E^\star$ and $1 \leq r \leq s \leq n$ we use $\rho[r]$ to denote ρ_r, $\rho[r, s]$ to denote $\rho_r \dots \rho_s$, and $|\rho|$ to denote n.

A program $\mathcal{CP}$ *respects nested locking* if for every thread $T \in \mathcal{T}$ and for every local run π of T, π releases locks in the opposite order it acquires them. That is, there exists no l, l' $(l \neq l')$ such that π contains a contiguous subsequence $acq(l); acq(l'); rel(l)$ when projected onto the the acquire and release transitions of l and l'[3].

From this point on, whenever we refer to a concurrent program $\mathcal{CP}$, we assume that it respects nested locking. Restricting our attention to programs that respect nested locking allows us to keep reasoning about run interleavings tractable. We will make critical use of this assumption in the following.

2.1 Locking Information

[2] Formally, $\text{Lock-Set}_T(\pi) = \{l \in \mathcal{L} \mid \exists i.\pi_T[i] = acq(l) \wedge \nexists j > i \ s.t. \ \pi_T[j] = rel(l)\}$

[3] this implies that locks are not re-entrant

Consider the example in Figure 1. We would like to know whether the fact d generated at the location b reaches the location a (without being killed at location c). If the thread on the right takes the `else` branch in the first execution of the loop, it will have to go through location c and kill the fact d before the execution of the program can get to location a. However, if the program takes the `then` branch in the first iteration of the loop and takes the `else` branch in the second one, then execution can follow to a without having to kill d first. This example shows that in general, the sorts of interleavings that we must consider in a bitvector analysis can be quite complicated.

```
acq(l2);                    acq(l1);
   acq(l1);                    acq(l2);
      ...                         ...
   rel(l1);                    rel(l2);
   a: ...                      while (...) {
rel(l2);                          if (...) {
                                     rel(l1);
                                     acq(l1);
                                  } else {
                                     b: ... // gen "d"
                                  }
                               }
                               c: ... // kill "d"
                            rel(l1);
```

Fig. 1. Locking information.

In [5] and [6], compositional reasoning approaches for programs that respect nested locking were introduced, which are based on *local* locking information. We quickly give an overview of this here. In the following, $T \in \mathcal{T}$ denotes a thread, and $\rho \in E_T^\star$ denotes a sequence of edges of T (in practice, ρ will be a run or a suffix of a run of T).

- Locks-Held$_T(\rho, i) = \{l \in \mathcal{L} \mid \forall k \geq i.l \in \text{Lock-Set}_T(\rho[1,k])\}$: the set of locks held continuously by T through ρ, starting no later than at position i.
- Locks-Acq$_T(\rho) = \{l \in \mathcal{L} \mid \exists k.\rho[k] = T{:}acq(l)\}$: the set of locks that are acquired by T along ρ.
- $fah_T(\rho)$ (forward acquisition history): a partial function which maps each lock l whose last acquire in ρ has no matching release, to the set of locks that were acquired after the last acquisition of l (and is undefined otherwise)[4].
- $bah_T(\rho, i)$ (backward acquisition history): a partial function which maps each lock l that is held at $\rho[i]$ and is released in $\rho[i, |\rho|]$ to the set of locks that were released before the first release of l in $\rho[i, |\rho|]$ (and is undefined otherwise).

We omit T subscripts for all of these functions when T is clear from the context.

Of these definitions, fah and bah are the least common, so we illustrate their use with runs from Figure 1. The run of the right thread that starts at the beginning, enters the `while` loop, and takes the `else` branch to end at `b` has forward acquisition history $[\texttt{l1} \mapsto \{\texttt{l2}\}]$. If that run continues to loop, taking the `then` branch and then the `else` branch to end back at `b`, that run has forwards acquisition history $[\texttt{l1} \mapsto \{\}]$. The run of the left thread that executes the entire code block has backwards acquisition history $[\texttt{l2} \mapsto \{\}]$ at `a` and $[\texttt{l2} \mapsto \{\texttt{l1}\}; l1 \mapsto \{\}]$ between the acquire and release of `l1`.

2.2 Bitvector data flow Analysis

Let $\mathbb{D}$ be a finite set of data flow facts of interest. The goal of data flow analysis is to replace the *full* semantics by an abstract version which is tailored to deal

[4] Note that the domain of $fah_T(\rho)$ (denoted $dom(fah_T(\rho))$) is exactly Lock-Set$_T(\rho)$.

with a specific problem. The abstract semantics is specified by a local semantic functional $[\![\cdot]\!]_{\mathbb{D}} : E \to (\mathbb{D} \to \mathbb{D})$ where for each transition t, $[\![t]\!]_{\mathbb{D}}$ denotes the transfer function associated with t. $[\![\cdot]\!]_{\mathbb{D}}$ gives abstract meaning to every CFA edge (program statement) in terms of a transformation function from a semi-lattice $(\mathbb{D}, \sqcap)$ (where $\sqcap$ is $\cup$ or $\cap$) into itself. We will drop $\mathbb{D}$ and simply use $[\![t]\!]$ when $\mathbb{D}$ is clear from the context. We extend $[\![\cdot]\!]$ from transitions to transition sequences in the natural way: $[\![\epsilon]\!] = id$, and $[\![t\rho]\!] = [\![\rho]\!] \circ [\![t]\!]$.

Bitvector problems can be characterized by the simplicity of their local semantic functional $[\![\cdot]\!]$: for any transition t, there exist sets $gen(t)$ and $kill(t)$ $(\subseteq \mathbb{D})$ such that $[\![t]\!](D) = (D \cup gen(t)) \setminus kill(t)$. Equivalently, for any t, $[\![t]\!]$ can decomposed into $|\mathbb{D}|$ monotone functions $[\![t]\!]_i : \mathbb{B} \to \mathbb{B}$, where $\mathbb{B}$ is the Boolean lattice $(\{\mathrm{ff}, \mathrm{tt}\}, \Rightarrow)$.

Our goal is to compute the concurrent meet-over-paths ($CMOP$) value of edge[5] t of $\mathcal{CP}$, defined as

$$CMOP[t] = \bigsqcap_{\rho t \in \mathcal{R}_{\mathcal{CP}}} [\![\rho]\!]_{\mathbb{D}}(\top_{\mathbb{D}})$$

$CMOP[t]$ is the optimal solution to the data flow problem. Note in particular that only runs that respect the semantics of locking contribute to the solution. This definition is not effective, however, since $\mathcal{R}_{\mathcal{CP}}$ may be infinite; the contribution of this work is an efficient algorithm for computing $CMOP[t]$.

In the following, we discuss the class of *intraprocedural forward may* bitvector analyses for a concurrent program model with nested locking as our main contribution. In Section 6, we discuss how to adapt our techniques to other program models and analyses, including interprocedural analysis, backwards analysis, and an extension to a parameterized concurrent program model.

3 Concurrent Data Flow Framework

Fix a concurrent program $\mathcal{CP}$ with set of threads $\mathcal{T}$, set of locks $\mathcal{L}$, and a set of data flow facts $\mathbb{D}$ with meet $\cup$ (bitvector problems that use $\cap$ for meet can be solved using their dual problem). For a data flow fact $d \in \mathbb{D}$, and for a transition t, let $[\![t]\!]_d$ denote $[\![t]\!]_{\mathbb{D}}$ projected onto d. Call a sequence π **d-preserving** if $[\![\pi]\!]_d = id$. In particular, the empty sequence ϵ is d-preserving for any $d \in \mathbb{D}$.

The following observation from [8] is the key to the efficient computation of the *interleaving effect*. It pinpoints the specific nature of a semantic functional for bitvector analysis, whose codomain only consists of *constant functions* and the *identity*:

Lemma 1. *[8] For a data flow fact $d \in \mathbb{D}$, and a transition t of a concurrent program $\mathcal{CP}$, $d \in CMOP[t]$ iff there exists a run $t_1 \cdots t_n t \in \mathcal{R}_{\mathcal{CP}}$ and there exists k, $(1 \leq k \leq n)$ such that $[\![t_k]\!]_d = const_{tt}$ and for all m, $(k < m \leq n)$, we have $[\![t_m]\!]_d = id$.*

[5] For the CFA formulation of data flow analysis, data flow transformation functions and solutions correspond to transitions rather than nodes.

Call such a run a **d-generating run** for t, and call t_k the generating transition of that run.

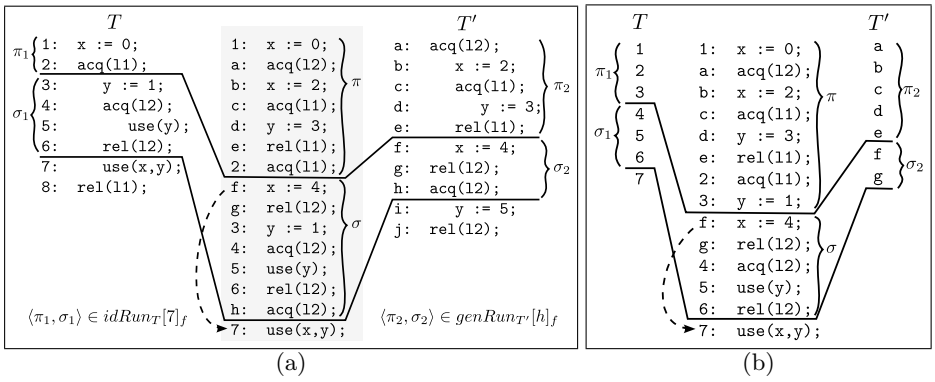


Fig. 2. A witness run (a) and a *normal* witness run (b) for definition `f` reaching `7`

This lemma restricts the possible interference within a concurrent program: *if there is any interference, then the interference is due to a single statement within a parallel component.* Consider the program in Figure 2(a). In a reaching definitions analysis, only transition `f` (of T') can generate the "definition at `f` reaches" fact. For any witness trace and any fact d, we can pinpoint a single transition that generates this fact (namely, the last occurrence of a generating transition on that trace). This is *not* true for data flow analyses which are not bitvector analyses. For example, in a null pointer dereference analysis, witnesses may contain a chain of assignments, no single one of which is "responsible" for the pointer in question being null, but *combined* they make the value of a pointer null. Our algorithm critically takes advantage of the simplicity of bitvector problems to achieve both efficiency and precision, and cannot be trivially generalized to handle all data flow problems.

Based on Lemma 1 and the observation from [6] that runs can be projected onto runs with fewer threads, we get the following:

Lemma 2. *For a data flow fact $d \in \mathbb{D}$, and for a transition t of thread T, there exists a d-generating run for t if and only if one of the following holds:*

- *There exists a local d-generating run for t (that is, a d-generating run consisting only of transitions from T). Call such a run a* single-indexed d-generating run.
- *There exists a thread T' ($T \neq T'$) such that there is a d-generating run π for t consisting only of transitions from T and T' and such that the generating transition of π belongs to T'. Call such a run a* double-indexed d-generating run.

Proof. First, we note a simple fact: for any run $\pi \in \mathcal{R}_{\mathcal{CP}}$ and for any subset of threads $\mathcal{T}' \subseteq \mathcal{T}$, $\pi_{\mathcal{T}'} \in \mathcal{R}_{\mathcal{CP}}$. That is, we may take any run of $\mathcal{CP}$, project onto an arbitrary subset of threads, and obtain another run. This is clear: since threads initially hold no locks, not executing transitions from a thread will never disable transitions that would be enabled had the thread been running.

Let π be an d-generating path to t and let t_g be the generating transition of π. If t_g is a transition of T, then π_T is a d-generating path for t. If t_g is a transition of some other thread T', then $\pi_{\{T,T'\}}$ is a d-generating path for t. □

Thus, to determine whether $d \in \mathit{CMOP}[t]$ (i.e. fact d may be true at t), it is sufficient to check whether there is a single- or double-indexed d-generating run to t. Therefore, the precise solution to the concurrent bitvector analysis problem can be computed by only reasoning about concurrent programs with one or two threads, so long as we consider each pair of threads in the system. The existence of a single-indexed d-generating run to t can be determined by a *sequential* bitvector data flow analysis, which have been studied extensively.

Here, we discuss a compositional technique for enumerating the double-indexed d-generating runs. In order to achieve compositionality, we (1) characterize double-indexed d-generating runs in terms of two local runs, and (2) provide a procedure to determine whether two local runs can be combined into a global run. First, we define for each thread T, each transition t of T, and each data flow fact $d \in \mathbb{D}$:

- $IR_T[t]_d = \{\langle \pi, \sigma \rangle \mid \pi\sigma t \in \mathcal{R}_T \ \wedge [\![\sigma]\!] = id\}$
- $GR_T[t]_d = \{\langle \pi, \sigma \rangle \mid \pi\sigma \in \mathcal{R}_T \ \wedge \ [\![\sigma[1]]\!] = \mathit{const}_{tt} \ \wedge \ [\![\sigma[2, |\sigma| - 1]]\!] = id \wedge \sigma[|\sigma|] = t\}$.

Intuitively, $IR_T[t]_d$ and $GR_{T'}[t']_d$ correspond to sets of *local* runs of threads T and T' which can be combined (interleaved in a lock-valid way) to create a *global* d-generating run to t. For example, in Figure 2(a), the definition at line `f` reaches the use at line `7` (in a reaching-definitions analysis) since the local runs $\pi_1\sigma_1$ of T and $\pi_2\sigma_2$ of T' can be combined into the run $\pi\sigma$ (demonstrated in the center) to create a double-indexed generating run. The following proposition is the key to our compositional approach:

Proposition 1. *Assume a concurrent program $\mathcal{CP}$ with two threads T_1 and T_2. There exists a double-indexed d-generating run to transition t_1 of thread T_1 if and only if there exists a transition t_2 of thread T_2 such that there exists $\langle \pi_1, \sigma_1 \rangle \in IR_{T_1}[t_1]_d$ and $\langle \pi_2, \sigma_2 \rangle \in GR_{T_2}[t_2]_d$ and a run $\pi\sigma \in \mathcal{R}_{\mathcal{CP}}$ such that $\pi_{T_1} = \pi_1$, $\pi_{T_2} = \pi_2$, $\sigma_{T_1} = \sigma_1$ and $\sigma_{T_2} = \sigma_2$.*

Since IR and GR are sets of *local* runs, they can be computed locally and independently, and checked whether they can be interleaved in a second phase. However, $IR_T[t]_d$ and $GR_T[t]_d$ are (in the general case) infinite sets, so we need to find finite means to represent them. In fact, we do not need to know about all such runs: the only thing that we need to know is whether there exists *a* d-generating run in one thread, and *a* d-preserving run in the other thread that

can be combined into a lock-valid run to carry the fact d generated in one thread to a particular control location in the other thread. Proposition 2, a simple consequence of a theorem from [5], provides a means to represent these sets with finite abstractions.

Proposition 2. *Let $\mathcal{CP}$ be a concurrent program, and let T_1, T_2 be threads of $\mathcal{CP}$. Let $\pi_1\sigma_1$ be a local run of T_1 and let $\pi_2\sigma_2$ be a local run of T_2. Then there exists a run $\pi\sigma \in \mathcal{R}_{\mathcal{CP}}$ with $\pi_{T_1} = \pi_1$, $\pi_{T_2} = \pi_2$, $\sigma_{T_1} = \sigma_1$, and $\sigma_{T_2} = \sigma_2$ if and only if:*

- *Lock-Set$(\pi_1) \cap$ Lock-Set$(\pi_2) = \emptyset$*
- *$fah(\pi_1)$ and $fah(\pi_2)$ are consistent*
- *Lock-Set$(\pi_1\sigma_1) \cap$ Lock-Set$(\pi_2\sigma_2) = \emptyset$.*
- *$fah(\pi_1\sigma_1)$ and $fah(\pi_1\sigma_2)$ are consistent*
- *$bah(\pi_1\sigma_1, |\pi_1|)$ and $bah(\pi_2\sigma_2, |\pi_2|)$ are consistent*
- *Locks-Acq$(\sigma_1) \cap$ Locks-Held$(\pi_2\sigma_2, |\pi_2|) = \emptyset$ and Locks-Acq$(\sigma_2) \cap$ Locks-Held$(\pi_1\sigma_1, |\pi_1|) = \emptyset$.*

Observe that Proposition 2 states that one can check whether two *local* runs can be interleaved into a *global* run by performing a few consistency checks on finite representations of the local lock behaviour of the two runs. In other words, one does not have to know what the runs are; one has to only know what the locking information for the runs are. Therefore, we use this information as our finite representation for the set of runs; more precisely, we use a quadruple consisting of two forwards acquisition histories, a backwards acquisition history, and a set of locks acquired to represent an *abstract* run[6]. Let $\mathcal{P}$ be the set of all such abstract runs. We say that two run abstractions are *compatible* if they may be interleaved (according to Proposition 2). We then define an abstraction function $\alpha : E^\star \times E^\star \to \mathcal{P}$ that computes the abstraction of a run:

$$\alpha(\langle \pi, \sigma \rangle) = \langle fah(\pi), \text{Locks-Acq}(\sigma), fah(\pi\sigma), bah(\pi\sigma, |\pi|) \rangle$$

For each transition $t \in E_T$ and data flow fact d, this abstraction function can be applied to the sets $IR_T[t]_d$ and $GR_T[t]_d$ to yield the sets $IR^\alpha_T[t]_d$ and $GR^\alpha_T[t]_d$, respectively:

$$IR^\alpha_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in IR_T[t]_d\}$$
$$GR^\alpha_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in GR_T[t]_d\}$$

For example, the abstraction of the preserving run $\pi_1\sigma_1$ and the generating run $\pi_2\sigma_2$ from Figure 2(a) are (in order):

$$\langle \underbrace{[l_1 \mapsto \{\}]}_{fah(\pi_1)}, \underbrace{[l_1, \mapsto \{l_2\}]}_{fah(\pi_1\sigma_1)}, \underbrace{[l_2 \mapsto \{\}]}_{bah(\pi_1\sigma_1, |\pi_1|)}, \underbrace{\{l_2\}}_{\text{Locks-Acq}(\sigma_1)} \rangle$$
$$\langle \underbrace{[l_2 \mapsto \{l_1\}]}_{fah(\pi_2)}, \underbrace{[l_2 \mapsto \{\}]}_{fah(\pi_2\sigma_2)}, \underbrace{[l_2 \mapsto \{\}]}_{bah(\pi_2\sigma_2, |\pi_2|)}, \underbrace{\{l_2\}}_{\text{Locks-Acq}(\sigma_2)} \rangle$$

[6] Note that for a run $\pi\sigma$, we can compute Lock-Set(π) as $dom(fah(\pi))$, Lock-Set$(\pi\sigma)$ as $dom(fah(\pi\sigma))$, and Locks-Held$(\pi\sigma, |\pi|)$ as $(dom(fah(\pi)) \cap dom(fah(\pi\sigma))) \setminus$ Locks-Acq(σ).

The definitions of IR^α and GR^α, and Proposition 2 imply the following proposition:

Proposition 3. *Assume a concurrent program $\mathcal{CP}$ with two threads T_1 and T_2. There exists a double-indexed d-generating run to transition t_1 of thread T_1 if and only if there exists a transition t_2 of thread T_2 such that there exists elements of $IR^\alpha_T[t]_d$ and $GR^\alpha_{T'}[t']_d$ which are compatible.*

For a fact d and a transition $t \in E_T$, the sets $IR^\alpha_T[t]_d$ and $GR^\alpha_T[t]_d$ are finite, and therefore one can use Proposition 3 to provide a solution to the concurrent bitvector problem, once $IR^\alpha_T[t]_d$ and $GR^\alpha_T[t]_d$ have been computed. In the next section, we propose an optimization that provides the same solution using a potentially much smaller subsets of these sets.

3.1 Normal Runs

The sets $IR^\alpha_T[t]_d$ and $GR^\alpha_T[t]_d$ may be large in practice, so we introduce the concept of *normal runs* to replace $IR^\alpha_T[t]_d$ and $GR^\alpha_T[t]_d$ with smaller subsets that are still sufficient for solving bitvector problems.

Definition 1. *Call a double-indexed d-generating run $\pi\sigma t$ (consisting of transitions of threads T and T', where t is a transition of T) with generating transition $\sigma[1]$ (of thread T')* normal *if:*

- $|\sigma| = 1$ *(that is, $\sigma[1]$ is an immediate predecessor of t), or*
- *All of the following hold:*
  - *The first T transition in σt is an* acquire *transition.*
  - *The last T' transition in σt is a* release *transition.*
  - *$\nexists i$ ($1 \leq i \leq |\sigma|$) such that after executing $\pi(\sigma[1,i])$, T frees all held locks.*
  - *$\nexists i$ ($1 < i \leq |\sigma|$) such that after executing $\pi(\sigma[1,i])$, T' frees all held locks.*

Intuitively, normal runs minimize the number of transitions between the generating transition and the end of the (double-indexed) run. Consider the run in Figure 2(a): it is a witness for definition at `f` reaching the use at `7`, but it is not *normal*, since σ_1 does not begin with an acquire and σ_2 does not end with a release. Figure 2(b) pictures a witness that uses the same transitions (except `h` has been removed from the end of σ_2), but which has a shorter σ component. Note that runs that are minimal in this sense are indeed normal; the reverse, however, does not hold.

Definition 2. *Let $\mathcal{CP}$ be a concurrent program, and let T be a thread of $\mathcal{CP}$.*
A pair $\langle \pi, \sigma \rangle$ is id-normal *if $\pi\sigma \in \mathcal{R}_T$, and either*

- *$\sigma = \epsilon$, or*
- *$\sigma[1]$ is an acquire transition and there is no proper prefix σ' of σ such that Lock-Set$(\pi\sigma') = \emptyset$.*

A pair $\langle \pi, \sigma \rangle$ *is* gen-normal *if* $\pi\sigma \in \mathcal{R}_T$*, and either*

- $|\sigma| = 1$*, or*
- $\sigma[|\sigma|]$ *is a release transition and there is no proper prefix* σ' *of* σ *such that* $\mathit{Lock\text{-}Set}(\pi\sigma') = \emptyset$*.*

Intuitively, for a concurrent program $\mathcal{CP}$ with two threads T and T', a normal double-indexed d-generating run to $t \in E_T$ is made up of a double-indexed d-generating run to t that is *id-normal* when projected onto T, and *gen-normal* when projected onto T'.

We show that it is sufficient to consider only *normal* runs for our analysis, by proving that the existence of a double-indexed d-generating run implies the existence of a normal double-index d-generating run.

Lemma 3. *Let* $t_1 \cdots t_n \in \mathcal{R}_T$ *and let* $t'_1 \cdots t'_m \in \mathcal{R}_{T'}$*. If there is a run* $\rho = \pi\sigma$ *(*$\rho \in \mathcal{R}_{\mathcal{CP}}$*) such that there exist* $1 \leq i \leq n$ *and* $1 \leq j \leq m$ *where:*
$\pi_T = t_1 \dots t_i \quad \sigma_T = t_{i+1} \dots t_n \quad \pi_{T'} = t'_1 \dots t'_j \quad \sigma_{T'} = t'_{j+1} \dots t'_m$
Then, the following hold:

1. *If* t'_{j+1} *is not an acquire transition, then* $\exists \pi', \sigma'$ *such that* $\pi'\sigma' \in \mathcal{R}_{\mathcal{CP}}$*, and*
   $\pi'_T = t_1 \dots t_i \quad \sigma'_T = t_{i+1} \dots t_n \quad \pi'_{T'} = t'_1 \dots t'_{j+1} \quad \sigma'_{T'} = t'_{j+2} \dots t'_m$
2. *If* t'_m *is not a release, then* $\exists \sigma'$ *such that* $\pi\sigma' \in \mathcal{R}_{\mathcal{CP}}$ *is a valid run, and*
   $\pi'_T = t_1 \dots t_i \quad \sigma'_T = t_{i+1} \dots t_n \quad \pi'_{T'} = t'_1 \dots t'_j \quad \sigma'_{T'} = t'_{j+1} \dots t'_{m-1}$

Proof. This lemma is a consequence of Lipton's theory of movers [12]:

1. non-acquire transitions are left movers, so if the first transition from T' in π is not an acquire, it can be moved to the beginning of π.
2. non-release transitions are right movers, so if the last transition from T' in π is not a release, it can be moved right to the end of π, and then need not be executed to form a run.

□

Lemma 3 is used to trim the beginning of a d-preserving run if it does not start with an *acquire* (part 1) and the end of a d-generating run if it does not end in a *release* (part 2). The run in Figure 2(b) is obtained from the run in Figure 2(a) by an application of Lemma 3.

Lemma 4. *If there is a run* $\pi\sigma$ *of concurrent program* $\mathcal{CP}$ *(consisting of two threads* T *and* T'*) such that* $\pi_T = t_1 \dots t_i$*,* $\sigma_T = t_{i+1} \dots t_n$*,* $\pi_{T'} = t'_1 \dots t'_j$ *and* $\sigma_{T'} = t'_{j+1} \dots t'_m$*, and if there exists* k *(*$j < k \leq m$*) such that* $\mathit{Lock\text{-}Set}_{T'}(t'_1 \dots t'_k) = \emptyset$*, then there exists a run* $\pi'\sigma'$ *of* $\mathcal{CP}$ *where* $\pi'_T = t_1 \dots t_i$*,* $\sigma'_T = t_{i+1} \dots t_n$*,* $\pi'_{T'} = t'_1 \dots t'_k$ *and* $\sigma'_{T'} = t'_{k+1} \dots t'_m$*.*

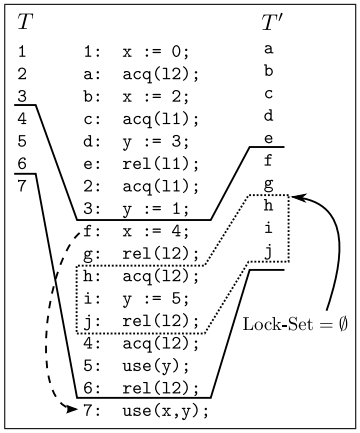
Similarly, if there exists k *(*$j \leq k \leq m-1$*) such that* $\mathit{Lock\text{-}Set}_{T'}(t'_1 \dots t'_k) = \emptyset$*, then there exists a run* $\pi\sigma'$ *where* $\sigma'_T = \sigma_T$ *and* $\sigma'_{T'} = t_i \dots t_k$*.*

Proof. Let ρ be the sequence of transitions in σ appearing after t'_k, and let t_r be the first transition from T appearing before t'_k. Then we construct π' and σ' as follows:

$$\pi' = t'_1 \cdots t'_k t_1 \cdots t_i$$
$$\sigma' = t_{i+1} \cdots t_r \rho$$

First, we see that $t'_1 \cdots t'_k$ is a run of T', and thus is a run of $\mathcal{CP}$. Since Lock-Set$(t'_1 \cdots t'_k) = \emptyset$, every transition in $t_1 \cdots t_r$ is enabled after executing $t'_1 \cdots t'_k$, and so $t'_1 \cdots t'_k t_1 \cdots t_r$ is a run. Finally, since T and T' are in the same states they were in after executing $t'_1 \cdots t'_k t_1 \cdots t_r$ as they were after executing $\pi\sigma$ up to t_k, ρ is enabled and so $\pi'\sigma'$ is a run. □

Lemma 4 is a consequence of Proposition 2. It is used to trim the beginning of d-preserving runs and the end of d-generating runs. The figure to the right illustrates the application of this lemma: thread T' holds no locks after executing `g`, so transitions `h`, `i`, and `j` need not be executed. The witness pictured for definition `f` reaching `7` corresponds to the normal witness obtained by removing the dotted box.



The following Proposition, which is a consequence of Lemmas 3 and 4, implies that it is sufficient to only consider *normal* runs for the analysis. Therefore, we can ignore runs that are not normal without sacrificing soundness.

Proposition 4. *If there exists a double-indexed d-generating run of concurrent program $\mathcal{CP}$ leading to a state q, then there exists a normal double-indexed d-generating run of $\mathcal{CP}$ leading to q.*

Proof. Let $\pi\sigma t$ be a d-generating path to t (of thread T) with generating transition $\sigma[1]$ (of thread T') and such that $|\sigma|$ is minimal. We show that $\pi\sigma t$ is normal.

First, if $|\sigma| = 1$, then $\pi\sigma t$ is obviously normal. So assume $|\sigma| > 1$. If $\pi\sigma t$ fails any of the normality conditions, we may use Lemma 3 or Lemma 4 to construct a d-generating path to t with a shorter σ component, contradicting the minimality of $|\sigma|$.

Thus, d-generating paths minimizing $|\sigma|$ are normal (although the converse does not hold). Since any transition t with a double-indexed d-generating path has a double-indexed d-generating path that minimizes $|\sigma|$ (by well-ordering), it follows that t has a normal d-generating path. □

Therefore, for any transition t and data flow fact d, we define normal versions (subsets of these sets which contain only normal runs) of $IR^{\alpha}_{T}[t]_d$ and $GR^{\alpha}_{T}[t]_d$ as follows:

$$NIR_T[t]_d = \{\alpha(\langle\pi,\sigma\rangle) \mid \langle\pi,\sigma\rangle \in IR_T[t]_d \wedge (|\sigma| = 0 \\ \vee\, |\sigma| = 0(\nexists k.\text{Lock-Set}(\pi\sigma[1,k]) = \emptyset \wedge \sigma[1]\text{is an acquire}))\}$$

$$NGR_T[t]_d = \{\alpha(\langle\pi,\sigma\rangle) \mid \langle\pi,\sigma\rangle \in GR_T[t]_d \wedge (|\sigma| = 1 \\ \vee\, \nexists k.\text{Lock-Set}(\pi\sigma[1,k]) = \emptyset)\}$$

$NIR_T[t]_d$ ($NGR_T[t]_d$) is a finite representation of sets of normal d-preserving (generating) runs. In order to compute solutions for every data flow fact in $\mathbb{D}$ simultaneously, we extend $NIR_T[t]_d$ and $NGR_T[t]_d$ to *sets* of data flow facts. We define $NIR_T[t]$ to be a partial function that maps each abstract run p to the set of facts d for which there is a d-preserving run whose abstraction is p, and is undefined if there is no concrete run to t whose abstraction is p. $NGR_T[t]$ is defined analogously.

4 The Analysis

Here, we summarize the results presented in previous sections into a procedure for the bitvector analysis of concurrent programs. The procedure is outlined in Algorithm 1. Data flow facts reaching a transition t are computed in two different groups as per Lemma 2: facts with single-indexed generating runs and facts with double-indexed generating runs.

Algorithm 1 Concurrent Bitvector Analysis

```
1: Compute Summaries and Helper Sets // see Algorithm 2
2: for each T ∈ 𝒯 do
3:    Compute MFP_T // Single-indexed facts
4:    for each t ∈ E_T do
5:       Compute NDIF_T[t] // (Normal) double-indexed facts
6:       CMOP_T[t] := MFP_T[t] ∪ NDIF_T[t]
7:    end for
8: end for
```

On line 6, the facts from single- and double-indexed generating runs are combined into the solution of the concurrent bitvector analysis. Facts from single-indexed generating runs can be computed efficiently using well-known (maximum fixed point) sequential data flow analysis techniques. It remains to show how to to efficiently compute facts from double-indexed generating runs using the summaries and helper sets which are computed at the beginning of the analysis.

A naive way to compute $NDIF$ would involve iterating over all pairs of transitions from different threads to find compatible elements of NIR and NGR. This would create an $|E|^2$ factor in our algorithm, which can be quite large. We avoid this by computing thread summaries for each thread T. The summary,

$NGen_T$, combines information about each transition in T that is relevant for $NDIF$ computation for other threads. More precisely, $NGen_T$ is a function that maps each run abstraction p to the set of facts for which there is a generating run whose abstraction is p. Intuitively, $NGen_T$ groups together transitions that have similar locking information so that they can be processed at once. This speeds up our analysis significantly in practice.

Algorithm 2 Computing Summaries.

1: **for** each $T \in \mathcal{T}$ **do**
2: Compute NGR_T // *Normal generating runs*
3: $NGen_T := \lambda p. \bigcup\{NGR_T(t)(p) \mid t \in E_T \wedge p \in dom(NGR_T(t)(p))\}$
4: Compute NIR_T // *Normal preserving runs*
5: **end for**

The essential procedure for finding facts in $NDIF_T[t]$ is to: 1) find compatible (abstract) runs $p \in dom(NIR_T[t])$ and $p' \in dom(NGen'_T)$, and 2) add facts that are *both* preserved by p and generated by p'. This procedure is elaborated in Algorithm 3.

Algorithm 3 Compute NDIF (concurrent facts from non-predecessors)

Input: Thread T, transition $t \in E_T$
Output: $NDIF_T[t]$.
1: $NDIF_T[t] := \emptyset$
2: **for** each $T' \neq T$ in $\mathcal{T}$ **do**
3: **for** each $p \in dom(NIR_T[t](p))$ **do**
4: $NDIF_T[t] := NDIF_T[t] \cup \{NGen_{T'}[p'] \cap NIR_T[t](p) \mid compatible(p, p')\}$
5: **end for**
6: **end for**
7: return $NDIF_T[t]$

Complexity Analysis. The best known upper bound on the time complexity of our algorithm is quadratic in the number of threads, quadratic in the size of the domain, linear in the number of transitions in each thread, and double exponential in the number of locks. We stress that this is a *worst-case* bound, and we expect our algorithm to perform considerably better in practice. Programmers tend to follow certain disciplines when using locks, which decreases the double exponential factor in our algorithm. For example, allowing only constant-depth nesting of locks reduces the factor to single exponential. Our experimental results in Section 5 confirm that our algorithm does indeed perform very well on real programs.

4.1 Computing NIR and NGR

It remains to show how to compute NIR and NGR. Both of these sets can be computed by sequential data flow analyses. Since the analyses are local, we fix a thread T and refer to NIR_T and NGR_T as NIR and NGR. Recall for a transition t, $NIR[t]$ and $NGR[t]$ are defined to be partial functions mapping abstract runs to sets of dataflow facts; that is $NIR, NGR \in \mathcal{P} \rightsquigarrow 2^{\mathbb{D}}$. In the following, we will represent a partial function $f : \mathcal{P} \rightsquigarrow 2^{\mathbb{D}}$ by a subset $F \subseteq \mathcal{P} \times 2^{\mathbb{D}}$ defined by $F = \{\langle p, f(p)\rangle : p \in dom(f)\}$. We adapt the partial function space order and meet operation to our choice of representation as follows:

$$Y \sqsubseteq X \iff \forall p \in \mathcal{P}.\ (\exists D \subseteq \mathbb{D}.\langle p, D\rangle \in X \Rightarrow \exists D' \subseteq \mathbb{D}.D \subseteq D' \wedge \langle p, D'\rangle \in Y)$$

$$\begin{aligned} X \sqcap Y = \quad & \{\langle a, v\rangle \in X \mid \nexists v'.\langle a, v'\rangle \in Y\} \\ & \cup \{\langle a, v\rangle \in Y \mid \nexists v'.\langle a, v'\rangle \in X\} \\ & \cup \{a, \langle v \cup v'\rangle \mid \ \langle a, v\rangle \in X \wedge \langle a, v'\rangle \in Y\} \end{aligned}$$

Then for any transition t, we define NIR and NGR formally as:

$$NIR[t] = \bigsqcap \{\langle \alpha(\pi, \sigma), [\![\sigma]\!]_{id}(\mathbb{D})\rangle \mid \pi\sigma t \in \mathcal{R}_T \wedge normal_{id}(\alpha(\pi, \sigma))$$

$$NGR[t] = \bigsqcap \{\langle \alpha(\pi, \sigma), [\![\sigma]\!](\emptyset)\rangle \mid \pi\sigma t \in \mathcal{R}_T \wedge normal_{gen}(\alpha(\pi, \sigma))$$

where

$$[\![t]\!]_{id}(D) = D \setminus (gen(t) \cup kill(t))$$

$$\begin{aligned} normal_{id}(\pi, \sigma) = (|\sigma| > 0 \Rightarrow \exists l.\sigma[1] = acq(l)) \\ \wedge \nexists k \geq 1.\text{Lock-Set}(\pi(\sigma[1, k])) = \emptyset \end{aligned}$$

$$normal_{gen}(\pi, \sigma) = |\sigma| > 1 \Rightarrow \nexists k < |\sigma| - 1.\ \text{Lock-Set}(\pi(\sigma[1, k])) = \emptyset$$

We show how to compute $NIR[\cdot]$ using a local data flow analysis; the analysis for $NGR[\cdot]$ is similar. Since for any transition t, $NIR[t]$ is represented by a subset of $\mathcal{P} \times 2^{\mathbb{D}}$, the domain of the analysis is $\mathfrak{D} = 2^{\mathcal{P} \times 2^{\mathbb{D}}}$, with the meet operation as defined above. The semantic functional for the analysis is defined as:

$$[\![t]\!]_{NIR}(X) = filter(extend(t, X \sqcap begin(t)))$$

where

$$filter(X) = \{\langle\langle s, a, f, b\rangle, D\rangle \in X \mid dom(f) \neq \emptyset \wedge a \neq \emptyset\}$$

$$begin(t) = \{\langle\langle h, \emptyset, h, \lambda x.h(x) \cap \emptyset\rangle, \mathbb{D}\rangle \mid h \in AH[t]\}^7$$

$AH[t]$ denotes the set of forwards acquisition histories that reach t (formally, $AH[t] = \{fah(\pi) \mid \pi t \in \mathcal{R}_T\}$). In practice, AH can be computed as the maximum fixedpoint solution to the dataflow problem with domain $(2^{\mathcal{P}}, \cup)$ and

[7] equivalently, begin(t) = { $\langle \alpha(\pi, \epsilon), \mathbb{D}\rangle \mid \pi t \in \mathcal{R}_T$}

transfer functional $[\![t]\!]_{AH}(P) = \{extend_{fah}(t,p) \mid p \in P\}$. This is essentially a corollary of Lemma 5 and the fact that $[\![t]\!]_{AH}$ is distributive.

$$extend(t,X) = \{\langle extend_{\mathcal{P}}(\langle s,a,f,b\rangle,t), [\![t]\!]_{id}(D)\rangle \mid \langle\langle s,a,f,b\rangle, D\rangle \in X\}$$

$$extend_{\mathcal{P}}(t,\langle s,a,f,b\rangle) = \langle s, extend_{acq}(t,a), extend_{fah}(t,f), extend_{bah}(t,b,a)\rangle$$

$$extend_{acq}(t,a) = \text{if } t = acq(\ell) \text{ then } a \cup \{\ell\} \text{ else } a$$

$$extend_{fah}(t,f) = \begin{cases} \lambda x.\text{if } x = \ell \text{ then } \emptyset \text{ else } f(x) \cup \{\ell\} & \text{if } t = acq(\ell) \\ \lambda x.\text{if } x = \ell \text{ then } \bot \text{ else } f(x) & \text{if } t = rel(\ell) \\ f & \text{otherwise} \end{cases}$$

$$extend_{bah}(t,b,a) = \begin{cases} \lambda x.\text{if } x \in a \text{ then } b(x) \text{ else } b(x) \cup \{\ell\} & \text{if } t = rel(\ell) \\ b & \text{otherwise} \end{cases}$$

It is easy to check that $[\![t]\!]_{NIR}$ is distributive for all t, so we can compute the meet-over-all-paths solution to the NIR dataflow analysis problem (denoted $MOP_{NIR}[t]$) using standard maximum fixed point techniques. In the remainder of the section, we prove that MOP_{NIR} coincides with NIR.

Lemma 5. *For all $\pi\sigma t \in \mathcal{R}_T$,*

$$\alpha(\pi,\sigma t) = extend_{\mathcal{P}}(t, \alpha(\pi,\sigma))$$

Proof. This is a trivial consequence of the definitions of α and $extend_{\mathcal{P}}$, with one caveat: the backwards acquisition histories we compute with $extend_{\mathcal{P}}$ are slightly different from the definition of backward acquisition histories as defined in Section 2.1. The domain of a backwards acquisition history is the set of locks that are held when σ starts rather than the set of locks that are held when σ starts and released along σ. This is a minor technical detail that makes the analysis slightly easier to describe. But but the difference is immaterial, because the two formulations represent the same information for our purposes. As such, we abuse notation and claim equality. □

For convenience, we introduce a new functional $ef : E_T \to 2^{\mathcal{P}\times 2^{\mathbb{D}}} \to 2^{\mathcal{P}\times 2^{\mathbb{D}}}$ defined by

$$ef[t](X) = filter(extend(t,X))$$

We may extend ef to operate on sequences of transitions in the natural way: $ef[\epsilon](X) = X$ and $ef[t\sigma](X) = ef[\sigma](ef[t](X))$.

Lemma 6. *Let $\pi\sigma t \in \mathcal{R}_T$ and $D \subseteq \mathbb{D}$. $\langle \pi,\sigma\rangle$ is id-normal iff*

$$ef[\sigma](\{\langle \alpha(\pi,\epsilon), D\rangle\}) = \{\langle \alpha(\pi,\sigma), [\![\sigma]\!]_{id}(D)\rangle\}$$

Proof. First note that, given Lemma 5, it is clear that if $ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\})$ is not $\emptyset$, then $ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\}) = \{\langle\alpha(\pi,\sigma),[\![\sigma]\!]_{id}(D)\rangle\}$.

If $\sigma = \epsilon$, the result is trivial. So let $|\sigma| \geq 1$. Note that $ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\}) = \emptyset$ iff there exists a prefix $\sigma' t$ of σ such that $ef[\sigma' t](\{\langle\alpha(\pi,\epsilon),D\rangle\}) = \emptyset$. If σ' is the shortest prefix with this property, we have

$$\begin{aligned} ef[\sigma' t](\{\langle\alpha(\pi,\epsilon),D\rangle\}) &= filter(extend(t, ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\}))) \\ &= filter(\{\langle\alpha(\pi,\sigma' t),[\![\sigma' t]\!]_{id}(D))\end{aligned}$$

So $ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\}) = \emptyset$ iff there exists a prefix $\sigma' t$ of σ such that $dom(fah(\pi\sigma)) = \text{Lock-Set}(\pi\sigma) = \emptyset$ or $\text{Locks-Acq}(\sigma' t) = \emptyset$. Noting that $\text{Locks-Acq}(\sigma' t) = \emptyset$ iff $\sigma[1]$ is not an acquire, we have that $ef[\sigma](\{\langle\alpha(\pi,\epsilon),D\rangle\}) = \emptyset$ iff $\langle\pi,\sigma\rangle$ is not id-normal, so the lemma holds for all σ of length at least 1. □

Lemma 7. *For all $t\sigma \in E_T^\star$,*

$$[\![t\sigma]\!]_{NIR}(\emptyset) = [\![\sigma]\!]_{NIR}(\emptyset) \sqcap ef[t\sigma](begin(t)).$$

Proof. By induction on σ.

– Base case: $\sigma = \epsilon$

$$\begin{aligned} [\![t\sigma]\!]_{NIR}(\emptyset) &= [\![t]\!]_{NIR}(\emptyset) \\ &= ef[t](\emptyset \sqcap begin(t)) \\ &= ef[t](begin(t)) \\ &= [\![\sigma]\!]_{NIR}(\emptyset) \sqcap ef[t\sigma](begin(t))\end{aligned}$$

– Induction: let $\sigma = \sigma' t'$, and assume $[\![t\sigma']\!]_{NIR}(\emptyset) = [\![\sigma']\!]_{NIR}(\emptyset) \sqcap ef[\sigma'](begin(t))$.

$$\begin{aligned} [\![t\sigma' t']\!]_{NIR}(\emptyset) &= [\![t']\!]_{NIR}([\![t\sigma']\!]_{NIR}(\emptyset)) \\ &= [\![t']\!]_{NIR}([\![\sigma']\!]_{NIR}(\emptyset) \sqcap ef[\sigma'](begin(t))) \\ &= [\![t']\!]_{NIR}([\![\sigma']\!]_{NIR}(\emptyset)) \sqcap ef[t'](ef[\sigma'](begin(t))) \\ &= [\![\sigma' t']\!]_{NIR}(\emptyset) \sqcap ef[\sigma' t'](begin(t))\end{aligned}$$

We may now state the main result of this section:

Proposition 5. *For all $t \in E_T$,*

$$NIR[t] = MOP_{NIR}[t] \sqcap begin_{NIR}(t)$$

Proof. First, we note that $NIR[t] = MOP_{NIR}[t] \sqcap begin_{NIR}(t)$ iff for all $p \in \mathcal{P}$ and $d \in \mathbb{D}$, $\langle p,\{d\}\rangle \sqsupseteq NIR[t] \iff \langle p,\{d\}\rangle \sqsupseteq MOP_{NIR}[t] \vee \langle p,\{d\}\rangle \sqsupseteq begin_{NIR}(t)$.

We begin by proving the "⇒" direction. Let $\langle p,\{d\}\rangle \sqsupseteq NIR[t]$. Then there exists an id-normal d-preserving run $\langle\pi,\sigma\rangle$ to t such that $\alpha(\pi,\sigma) = p$. Distinguish two cases:

1. Case: $\sigma = \epsilon$. Then $fah(\pi) \in AH[t]$ and $\langle p, \mathbb{D}\rangle \in begin(t)$, whence $\{\langle p, \{d\}\rangle\} \sqsupseteq begin(t)$.
2. Case: $|\sigma| \geq 1$. Then π is a run that ends at $\sigma[1]$, so $fah(\pi) \in AH[\sigma[1]]$ and $\langle \alpha(\pi, \epsilon), \mathbb{D}\rangle \in begin(\sigma[1])$.
   Since $\langle \pi, \sigma\rangle$ is id-normal, it follows from Lemma 6 that $ef[\sigma](\{\langle \alpha(\pi, \epsilon), \{d\}\rangle\}) = \{\langle \alpha(\pi, \sigma), [\![\sigma]\!]_{id}(\{d\})\rangle\} = \{\langle p, \{d\}\rangle\}$, and from Lemma 7 that

$$\begin{aligned}
[\![\pi\sigma]\!]_{NIR}(\emptyset) &\sqsubseteq [\![\sigma]\!]_{NIR}(\emptyset) \\
&= [\![\sigma]\!](\emptyset) \sqcap ef[\sigma](begin(\sigma[1])) \\
&\sqsubseteq ef[\sigma](begin(\sigma[1])) \\
&\sqsubseteq ef[\sigma](\{\langle \alpha(\pi, \epsilon), \{d\}\rangle\}) \\
&= \{\langle p, \{d\}\rangle\}.
\end{aligned}$$

   Since $[\![\pi\sigma]\!]_{NIR} \sqsupseteq MOP[t]_{NIR}$, we have $\{\langle p, \{d\}\rangle\} \sqsupseteq MOP[t]_{NIR}$.

Now we prove the "$\Leftarrow$" direction.

1. Let $\langle p, \{d\}\rangle \sqsupseteq begin(t)$. Then there exists some run π to t with $\alpha(\pi, \epsilon) = p$. Since for any π, $\langle \pi, \epsilon\rangle$ is id-normal and d-preserving, $\langle p, \{d\}\rangle \sqsupseteq NIR[t]$.
2. Let $\langle p, \{d\}\rangle \sqsupseteq MOP_{NIR}[t]$. Then there exists some $\rho \in \mathcal{R}_T$ such that $\langle p, \{d\}\rangle \sqsupseteq [\![\rho]\!]_{NIR}(\emptyset)$.
   Let σ by the shortest suffix of ρ such that $\langle p, \{d\}\rangle \sqsupseteq [\![\sigma]\!]_{NIR}(\emptyset)$.
   By Lemma 7, $[\![\sigma]\!]_{NIR}(\emptyset) = [\![\sigma[2, |\sigma|]]\!]_{NIR}(\emptyset) \sqcap ef[\sigma](begin(\sigma[1]))$. By the minimality of σ, we must have $\langle p, \{d\}\rangle \sqsupseteq ef[\sigma](begin(\sigma[1]))$. Since ef acts pointwise, there exists some p' such that $\langle p', \mathbb{D}\rangle \in begin(\sigma[1])$ and $\langle p, \{d\}\rangle \sqsupseteq ef[\sigma](\{\langle p', \mathbb{D}\rangle)$.
   Since $\langle p', \mathbb{D}\rangle \in begin(\sigma[1])$, there exists some π such that $\alpha(\pi, \epsilon) = p'$. Since $\langle p, \{d\}\rangle \sqsupseteq ef[\sigma](\{\langle p', \mathbb{D}\rangle\})$, $\langle \pi, \sigma\rangle$ is id-normal (since $ef[\sigma](\{\langle p', \mathbb{D}\rangle\})$ is nonempty) and is d-preserving (since $d \in [\![\sigma]\!]_{id}(\mathbb{D})$). It follows that $\langle p, \{d\}\rangle = \langle \alpha(\pi, \sigma), \{d\}\rangle \sqsupseteq NIR[t]$.

Combining both directions, we get the desired proposition. □

Having finished the development for NIR analysis, we will now define the transfer function for the NGR analysis.

$$[\![t]\!]_{NGR}(X) = filter_{NGR}(extend_{NGR}(t, X \sqcap begin(t)))$$

where

$$extend_{NGR}(t, X) = \{\langle extend_{\mathcal{P}}(t, a), [\![t]\!](D)\rangle | \langle a, D\rangle \in X\}$$
$$filter_{NGR}(X) = \{\langle \langle s, a, f, b\rangle, D\rangle \in X \mid dom(f) \neq \emptyset\}$$

We can now state a proposition analogous to Proposition 5 for NGR analysis. The proof follows a similar development to the one for Proposition 5.

Proposition 6. *For all $t \in E_T$,*

$$NGR[t] = [\![t]\!]_{NGR}(MOP_{NGR}[t])$$

5 Case Study

We implemented the intraprocedural version of our algorithm and evaluated its performance on a nontrivial concurrent program. Our experiments indicate that our algorithm scales well in practice; in particular, its performance appears to be only weakly dependent on the number of threads. This is remarkable, considering the program analysis community's historical difficulties with multithreaded code.

The algorithm is implemented in OCaml, and is applicable to C programs using the *pthreads* library for thread operations. We use the CIL program analysis infrastructure for parsing, CFG construction, and sequential data flow analysis. The algorithm is parameterized by a module that specifies the gen/kill sets for each instruction, so lifting sequential bitvector analyses to handle threads and locking is completely automatic. We implemented a reaching definitions analysis module and instantiated our concurrent bitvector analysis with it; this concurrent reaching definitions analysis was the subject of our evaluation.

We evaluated the performance our algorithm on FUSE, a Unix kernel module and library that allows filesystems to be implemented in userspace programs. FUSE exposes parts of the kernel that are relevant to filesystems to userspace programs, essentially acting as a bridge between userspace and kernelspace. We analyzed the userspace portion.

Name	$\lvert\mathcal{T}\rvert$	$\lvert N\rvert$	$\lvert E\rvert$	$\lvert\mathbb{D}\rvert$	$\lvert\mathcal{L}\rvert$	Time
5 avg	5	2568.1	3037.9	208.9	1.0	1.0
5 med	5	405.0	453.0	62.0	1.0	0.1
10 avg	10	4921.9	5820.1	401.6	1.3	1.8
10 med	10	988.5	1105.0	155.5	1.0	0.1
50 avg	50	24986.3	29546.0	2047.0	3.1	10.7
50 med	50	22628.5	26607.0	2022.5	3.0	4.6
200a	200	79985	94120	6861	6	36.4
200b	200	119905	142248	9515	4	116.5
full	425	218284	258219	17760	6	347.8

Since our implementation currently supports only intraprocedural analyses, we inlined all of the procedures defined within FUSE and ignored calls to library procedures that did not acquire or release locks. We did a type-based must-alias analysis to finitize the set of locks and shared variables. Some procedures in the program had the (implicit) precondition that callers must hold a particular lock or set of locks at each call site; these 35 procedures could not be considered to be threads because they did not respect nested locking when considered independently. Each of the remaining 425 procedures was considered to be a distinct thread in our analysis. We divided these procedures into groups of 5 procedures, and analyzed each of those separately (that is, we analyzed the program consisting of procedures 1-5, 6-10, 11-15, etc). We repeated this process with groups of 10, 50, 100, 200, and also analyzed the entire program. We present mean and median statistics for the groups of 5, 10, 50, and 100 procedures. The experiments were conducted on a 3.16 GHz Linux machine with 4GB of memory.

The $|\mathcal{T}|$, $|N|$, $|E|$, $|\mathbb{D}|$, $|\mathcal{L}|$, and Time columns indicate number of threads, number of CFA nodes, number of CFA edges, number of data flow facts, number of locks, and running time (in seconds), respectively. As a result of the inlining step, there was a very large size gap between the smallest and the largest proce-

dures that we analyzed, which we believe accounts for the discrepancy between the mean and median statistics.

In Figure 3(a), we observe that the running time of our algorithm appears to grow quadratically in the number of threads in the program. However, the dispersion is quite high, which suggests that the running time has a weak relationship with the number of threads in the program. Indeed, the apparent quadratic relationship can be explained by the fact that the points that contain more threads also contain more total CFA edges. Figure 3(b) shows the running time of our algorithm as a function of total number of CFA edges in the program, which is a much tighter fit.

Figure 3(c) shows the running time of our algorithm as a function of the product of the number of CFA edges and the domain size of the program. This relationship is interesting because the time complexity of sequential bitvector analysis is $O(|E| \cdot |\mathbb{D}|)$. Our results indicate that there is a linear relationship between the running time of our algorithm and the product of the number of CFA edges and domain size of the program, which suggests that our algorithm's running time is proportional to $|E| \cdot |\mathbb{D}|$ in practice.

Our empirical analysis is not completely rigorous. In particular, our data points are not independent and our treatment of memory locations is not conservative. However, we believe that the results obtained are promising and suggest that the algorithm can be used as the basis for further work on data flow analysis for concurrent programs.

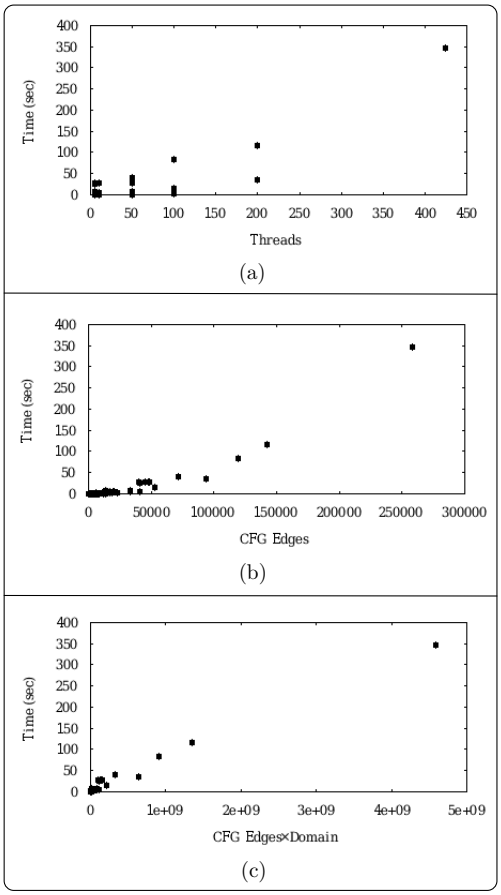


Fig. 3. Running time.

6 Generalizations

The results presented in this paper focus on forward intra-procedural bitvector analyses of concurrent programs, with a fixed set of threads and a fixed set of locks. We leave the generalization of the bitvector analyses to more expressive

classes of analyses as future work. The other constraints, however, were introduced for expository purposes rather than some limitation of our approach. In this Section, we discuss generalizations of our methods that remove these constraints.

6.1 Dynamic Lock and thread creation

Our system model does not account for lock aliasing or dynamic lock creation. It is straightforward to extend our model to handle both features by leveraging a conservative pointer analysis for lock variables. Essentially, we may ignore acquisitions and releases of lock variables that cannot be resolved to a single "actual" lock. Since ignoring locks leads to more program behaviors, the resulting analysis sound. More precise handling of lock aliases is left as future work.

In our system model, each thread is created simultaneously at the beginning of the program. For some programs, this is the correct behavior; for example, libraries such as FUSE (see Section 5), which are not equipped with program entry points, must assume that every thread may start at the beginning of the program, because this assumption conservatively approximates any program that creates threads dynamically. So by default, our analysis produces safe solutions for programs that dynamically create a finite set of threads. If a program does not have a fixed number of threads (for example, the program creates threads in a loop), since the number of thread creation points is still fixed, and we can use our results on parameterization to model the program behavior soundly. We leave more precise handling of thread creation and deletion as future work.

6.2 Function Calls and Recursion

Throughout the paper, we have presented our system model such that each thread is a finite state automaton. This means no function calls and, naturally, no recursion. All the ideas presented in this paper extend to the case that threads are pushdown systems, as long as we keep the principle of nested locking intact. The reachability and decomposition results of [5,6] hold for the case of pushdown systems, and we need to only extend the notion of control locations to configurations of pushdown systems (a control location plus the content of the stack). The analysis stays essentially the same – the only major difference is that the sequential data flow analyses employed by the algorithm must now be inter-procedural sequential data flow analyses.

6.3 Parameterization

One advantage of data flow analysis is that it can be applied to open programs; that is, programs that do not specify an entry point. This is useful for two reasons: one, there are many interesting programs that *are* open (e.g., libraries); two, it enables analyses to take advantage of programs that are composed of modules. Open programs are often problematic for program analyses because the

number of threads that are active in the program is not known a priori – analyses must be correct with respect to *any number* of threads in the program. Thus, for open programs, the ideal solution to a data flow problem the *parameterized* concurrent meet-over-paths solution (PCMOP). For a concurrent program $\mathcal{CP}$ with threads $T_1, \ldots, T_n$, for any transition $t \in \Sigma$, $PCMOP$ can be defined as follows:

$$PCMOP[t] = \bigsqcap_{\boldsymbol{k} \in \mathbb{N}^n} CMOP_{\boldsymbol{k}}[t]$$

where $CMOP_{\langle k_1,\ldots,k_n\rangle}[t]$ is the $CMOP$ solution at t for the concurrent program consisting of $k_1, k_2, \ldots, k_n$ distinct copies of $T_1, T_2, \ldots T_n$, respectively. That is, the $PCMOP$ solution is the meet over all paths of all possible concurrent programs that can be constructed using any number of copies the threads in $\mathcal{CP}$.

Computing the $PCMOP$ solution to a bitvector problem poses no additional difficulty for our algorithm. Indeed, we expect that computing the $PCMOP$ solution is sometimes even *easier* than computing the $CMOP$ solution. This is largely a consequence of Lemma 2: as far as the $CMOP$ solution is concerned, 2 copies of a thread is indistinguishable from k threads, for any $k \geq 2$. Next, note that in the presentation of our algorithm, special care was taken to assure that we did not consider concurrent executions of a single thread. If this restriction is removed, then the algorithm with compute the $PCMOP$ solution. Moreover, we may collapse the $NGen$ thread summaries even further into *program* summaries, which can speed up the algorithm.

6.4 Backwards analysis

It is straightforward to adapt our techniques to handle backwards bitvector analyses such as liveness and very busyness analysis. Traditionally, backwards data flow analyses can be solved using forward data flow analyses on reverse flow graphs. This method is not directly applicable in the concurrent case because of complications from synchronization and deadlocks. However, backwards analogues of all our results do hold, and optimal solutions to backwards analyses can be obtained by an algorithm similar to the one presented in Section 4.

7 Application and Future work

We discussed a number of very important applications of bitvector analysis in Section 1. One of the most exciting applications of our *precise* bitvector framework (in our opinion) is our ongoing work on studying more suitable abstractions for concurrent programs. Intuitively, by computing the solution to reaching-definitions analysis for a concurrent program, we can collect information about how program threads interact. We are currently working on using this information to construct abstractions to be used for more powerful concurrent program analyses, such as computing state invariants for concurrent libraries. The precision offered by our concurrent bitvector analysis approach is quite important in this domain, because it affects both the precision of the invariants that can be computed, and the efficiency of their computation.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
2. R. Chugh, J. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, pages 316–326, 2008.
3. Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *FoSSaCS '99*, pages 14–30, London, UK, 1999. Springer-Verlag.
4. Javier Esparza and Andreas Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL '00*, pages 1–11, New York, NY, USA, 2000. ACM.
5. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, pages 303–314, 2007.
6. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.
7. Nicholas Kidd, Peter Lammich, Tayssir Touili, and Thomas Reps. A decision procedure for detecting atomicity violations for communicating processes with locks. In *SPIN '09*, pages 125–142, Berlin, Heidelberg, 2009. Springer-Verlag.
8. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.
9. Jens Knoop. Parallel constant propagation. In *Euro-Par '98*, pages 445–455, London, UK, 1998. Springer-Verlag.
10. Jens Krinke. Static slicing of threaded programs. *SIGPLAN Not.*, 33(7):35–42, 1998.
11. Peter Lammich and Markus Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS '08*, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag.
12. Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
13. S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *PPOPP*, pages 129–138, New York, NY, USA, 1993.
14. S. S. Muchnick. *Advanced Compiler Design and Imlementation*. Morgan Kaufmann, 1997.
15. G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *SIGSOFT/FSE-6*, pages 24–34, New York, NY, USA, 1998.
16. G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing mhp information for concurrent java programs. In *ESEC/FSE-7*, pages 338–354, London, UK, 1999.
17. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pages 213–228, 2002.
18. R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*, pages 109–129, Irvine, Calif., 1990.
19. F. Nielson and H. Nielson. Type and effect systems. In *Correct System Design*, pages 114–136, 1999.

20. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
21. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP*, 2001.
22. Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. In *ESOP '00*, pages 351–365, London, UK, 2000. Springer-Verlag.