Compositional Bitvector Analysis for Concurrent Programs with Nested Locks*

Azadeh Farzan and 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 [6,7]. We have implemented our algorithm on top of an existing sequential data flow analysis tool, and demonstrate 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

* See [5] for an extended version of this paper including proofs and further discussions.

[1] Concurrent program slicing has been discussed previously [11], but to our knowledge there is no method up until now that handles locks precisely.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 253–270, 2010.
© Springer-Verlag Berlin Heidelberg 2010

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.

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. 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 [9]. Under the concurrent program model in [9], 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 [9]) 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 [7] if the locks are acquired in a *nested* manner (i.e. locks are released in the reverse order that they 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.

In this paper we discuss the class of *intraprocedural forward may* bitvector analyses for a concurrent program model with nested locking as our main contribution. Nested locks are a common programming practice; for example Java synchronized methods and blocks syntactically enforce this condition. Due to lack of space, all further discussions on the generalization of this case have been included in an extended version of this paper, which includes discussions on backward and interprocedural analyses, as well as an extension to a parameterized concurrent program model.

We have implemented our algorithm on top of the C language front-end CIL [18], 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 [19,15,20], flow analysis for concurrent software has also been studied. Flow-insensitive analyses can be directly adapted into the concurrent setting. Existing flow-sensitive analyses [14,16,17,21] 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 [9] 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 [10,22].

Foundational work on nested locks appears in [6,7]. Recently, analyses based on this work have been developed, including [8] and [12]. Notably, the authors of [8] 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, Σ_T) with a unique entry node s_T and a function $stmt_T : \Sigma_T \rightarrow Stmt$ that maps transitions to program statements. We assume no two threads have a common node (transition), and refer to the set of all nodes (transitions) by N (Σ). In the following, we will often identify transitions 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 finite path π through thread T starting at s_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 $\rho = t_1 \ldots t_n \in \Sigma^\star$ of transitions 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 $(\nexists T, T', p. T \neq T' \wedge \text{Lock-Set}_T((t_1 \cdots t_p)_T) \cap \text{Lock-Set}_{T'}((t_1 \cdots t_p)_{T'}) \neq \emptyset)$.

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 $\rho = t_1 \ldots t_n \in \Sigma^\star$ and $1 \leq r \leq s \leq n$ we use $\rho[r]$ to denote t_r, $\rho[r, s]$ to denote $t_r \ldots t_s$, and $|\rho|$ to denote n.

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

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' 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

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 [6] and [7], 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 $\rho \in \Sigma_T^\star$ denotes a sequence of transitions of T (in practice, ρ will be a run or a suffix of a run of T).

- $\text{Locks-Held}_T(\rho, i) = \{l \in \mathcal{L} \mid \forall k \geq i. l \in \text{Lock-Set}_T(\rho[1,k])\}$: the set of locks held continuously by T through ρ, starting no later than at position i.
- $\text{Locks-Acq}_T(\rho) = \{l \in \mathcal{L} \mid \exists k. \rho[k] = T{:}acq(l)\}$: 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).

[3] In the special case where $l = l'$, this condition implies that locks are not re-entrant.
[4] Note that the domain of $fah_T(\rho)$ (denoted $dom(fah_T(\rho))$) is exactly $\text{Lock-Set}_T(\rho)$.

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

As observed in [6,7], a necessary and sufficient condition for pairwise reachability cannot be stated in terms of locksets (the "current" lock behaviour of each thread). One needs to additionally consider the historical lock behaviour (fah and bah) of each thread, which places ordering constraints on locking events. This notion will be made more precise in Proposition 2; see [6,7] for more details. As an example of fah and bah, consider 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 with a specific problem. The abstract semantics is specified by a local semantic functional $[\![\cdot]\!]_{\mathbb{D}} : \Sigma \to (\wp(\mathbb{D}) \to \wp(\mathbb{D}))$ where for each transition t, $[\![t]\!]_{\mathbb{D}}$ denotes the transfer function associated with t. $[\![\cdot]\!]_{\mathbb{D}}$ gives abstract meaning to every CFA transition (program statement) in terms of a transformation function from a semi-lattice $(\wp(\mathbb{D}), \sqcap)$ (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 transition[5] t of $\mathcal{CP}$, defined as

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

$CMOP[t]$ 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 $CMOP[t]$.

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

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 (defined by $[\![t]\!]_d(p) = (p \vee d \in gen(t)) \wedge d \notin kill(t))$. 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 [9] 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. *[9] 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 $[\![t_m]\!]_d = id$.*

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

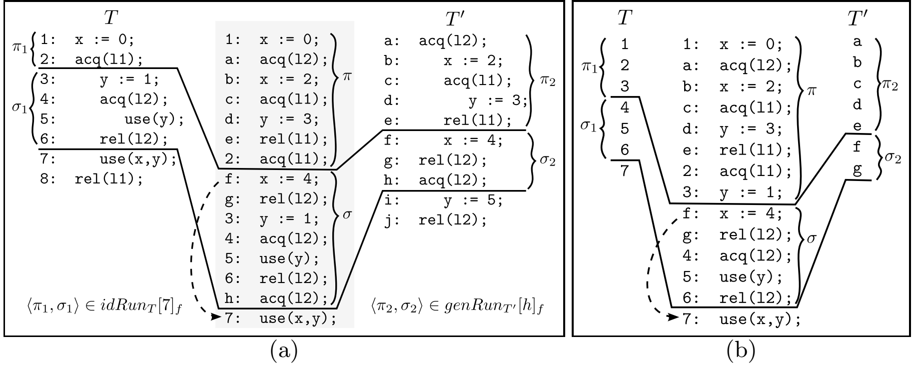


Fig. 2. A witness run (a) and a *normal* witness run (b) for definition `f` reaching `7`

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.* By *interference*, we mean any possible behaviour by other threads that may change the set of facts that hold in a program location; in the realm of the gen/kill problems, this may be in the form of a fact that sequentially holds getting killed, or a fact that does not sequentially hold getting generated. 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 [7] 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.*

Thus, to determine whether $d \in CMOP[t]$ (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 $d \in \mathbb{D}$:

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

Intuitively, $PR_T[t]_d$ 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 $\langle \pi_1, \sigma_1 \rangle \in PR_{T_1}[t_1]_d$ 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 PR 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, $PR_T[t]_d$ 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 [6], 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*[6]
- *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[7]. 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 $\alpha : \Sigma^\star \times \Sigma^\star \to \mathcal{P}$ that computes the abstraction of a run:

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

[6] Histories h and h' are consistent iff $\nexists \ell \in dom(h), \ell' \in dom(h'). \ell \in h'(\ell') \wedge \ell' \in h(\ell)$.

[7] 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).

For each transition $t \in \Sigma_T$ and data flow fact d, this abstraction function can be applied to the sets $PR_T[t]_d$ and $GR_T[t]_d$ to yield the sets $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$, respectively:

$$\widehat{PR}_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in PR_T[t]_d\}$$
$$\widehat{GR}_T[t]_d = \{\alpha(\langle \pi, \sigma \rangle) \mid \langle \pi, \sigma \rangle \in GR_T[t]_d\}$$

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):

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

The definitions of $\widehat{PR}$ and $\widehat{GR}$, 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 $\widehat{PR}_{T_1}[t_1]_d$ and $\widehat{GR}_{T_2}[t_2]_d$ which are compatible.*

For a fact d and a transition $t \in \Sigma_T$, the sets $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ are finite, and therefore one can use Proposition 3 to provide a solution to the concurrent bitvector problem, once $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ have been computed (we refer the reader to an extended version of this paper [5] for the details of these computations). 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 $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ may be large in practice, so we introduce the concept of *normal runs* to replace $\widehat{PR}_T[t]_d$ and $\widehat{GR}_T[t]_d$ with smaller subsets that are still sufficient for solving bitvector problems. 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*: Figure 2(b) pictures a witness consisting of the same transitions (except `h` has been removed from the end of σ_2), which has a shorter σ component. Note that runs that are minimal in this sense are indeed normal; the reverse, however, does not hold. We define normal runs formally as follows:

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* σ *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.*

Note that if there are no locking operations in a run, then the generating transition is *always* an immediate predecessor of the t, since there are no synchronization restriction to prevent this from happening. 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:*

$$\begin{aligned} \pi_T &= t_1 \dots t_i & \sigma_T &= t_{i+1} \dots t_n \\ \pi_{T'} &= t'_1 \dots t'_j & \sigma_{T'} &= t'_{j+1} \dots t'_m \end{aligned}$$

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*

$$\begin{aligned} \pi'_T &= t_1 \dots t_i & \sigma'_T &= t_{i+1} \dots t_n \\ \pi'_{T'} &= t'_1 \dots t'_{j+1} & \sigma'_{T'} &= t'_{j+2} \dots t'_m \end{aligned}$$

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*

$$\begin{aligned} \pi'_T &= t_1 \dots t_i & \sigma'_T &= t_{i+1} \dots t_n \\ \pi'_{T'} &= t'_1 \dots t'_j & \sigma'_{T'} &= t'_{j+1} \dots t'_{m-1} \end{aligned}$$

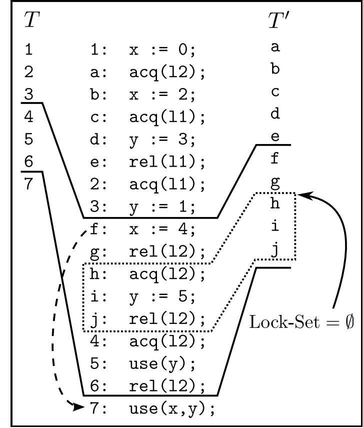
Lemma 3 is a consequence of Lipton's theory of reduction [13]. It is used to trim the beginning of a d-preserving run if it does not start with an *acquire* (part 1) and the end of a d-generating run if it does not end in a *release* (part 2). The run in Figure 2(b) is obtained from the run in Figure 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* $\textit{Lock-Set}_{T'}(t'_1 \dots t'_k) = \emptyset$*, then there exists a run* $\pi'\sigma'$ *of* $\mathcal{CP}$ *where*

$$\begin{aligned} \pi'_T &= t_1 \dots t_i & \sigma'_T &= t_{i+1} \dots t_n \\ \pi'_{T'} &= t'_1 \dots t'_k & \sigma'_{T'} &= t'_{k+1} \dots t'_m \end{aligned}$$

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

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 transition t, then there exists a normal double-indexed d-generating run of $\mathcal{CP}$ leading to t.*

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

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

$\widehat{NPR}_T[t]_d$ ($\widehat{NGR}_T[t]_d$) 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 $\widehat{NPR}_T[t]_d$ and $\widehat{NGR}_T[t]_d$ to *sets* of data flow facts. We define $\widehat{NPR}_T[t]$ to be a partial function that maps each abstract run $\widehat{\rho}$ to the set of facts d for which there is a d-preserving run whose abstraction is $\widehat{\rho}$, and is undefined if there is no concrete run to t whose abstraction is $\widehat{\rho}$. $\widehat{NGR}_T[t]$ is defined analogously. We refer the reader to an extended version of this paper [5] for more detailed information on $\widehat{NPR}$ and $\widehat{NGR}$.

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 $T \in \mathcal{T}$ **do**
3: Compute MFP_T // *Single-indexed facts*
4: **for** each $t \in \Sigma_T$ **do**
5: Compute $NDIF_T[t]$ // *(Normal) double-indexed facts*
6: $CMOP_T[t] := MFP_T[t] \cup NDIF_T[t]$
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 $\widehat{NPR}$ and $\widehat{NGR}$. This would create an $|\Sigma|^2$ 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 $\widehat{NGR}_T$ // *Normal generating runs*
3: $NGen_T := \lambda\widehat{\rho}. \bigcup\{\widehat{NGR}_T[t](\widehat{\rho}) \mid t \in \Sigma_T \wedge \widehat{\rho} \in dom(\widehat{NGR}_T[t])\}$
4: Compute $\widehat{NPR}_T$ // *Normal preserving runs*
5: **end for**

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

It remains to show how to compute $\widehat{NPR}$ and $\widehat{NGR}$. Both of these sets can be computed by a sequential data flow analysis. This analysis is based on an abstract interpretation of sequential trace semantics suggested by the abstraction function α. Best abstract transformers can derived from the definition α; more detailed information can be found in the extended version of this paper.

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

Input: Thread T, transition $t \in \Sigma_T$
Output: $NDIF_T[t]$.
1: $NDIF_T[t] := \emptyset$
2: **for** each $T' \neq T$ in $\mathcal{T}$ **do**
3:     **for** each $\widehat{\rho} \in dom(\widehat{NPR}_T[t])$ **do**
4:         $NDIF_T[t] := NDIF_T[t] \cup \{NGen_{T'}[\widehat{\rho}'] \cap \widehat{NPR}_T[t](\widehat{\rho}) \mid compatible(\widehat{\rho}, \widehat{\rho}')\}$
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. In practice, locksets, nesting depths, and consequently acquisition history sizes are very small (even if the number of locks in the program is not very small); and the complexity of our algorithm depends on the average size of acquisition histories, and not directly on the number of locks. Our experimental results in Section 5 confirm that our algorithm does indeed perform very well on real programs.

5 A 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 of 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.

Table 1. Experimental Results for FUSE

Test	$\|\mathcal{T}\|$	$\|N\|$	$\|\Sigma\|$	$\|\mathbb{D}\|$	$\|\mathcal{L}\|$	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 create a finite version of 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.

Table 1 presents the results of our experiments. The $|\mathcal{T}|$, $|N|$, $|\Sigma|$, $|\mathbb{D}|$, $|\mathcal{L}|$, and Time columns indicate number of threads, number of CFA nodes, number of CFA transitions, 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 procedures that we analyzed, which we believe accounts for the discrepancy between the mean and median statistics.

Our thread summarization technique is a very effective optimization in practice. As an example, for a scenario with 123 threads, and a CFA size of approximately 200K (sum of the number of nodes and transitions), the analysis time is 50 seconds with summarization, while it is 930 seconds without summarization – *about 20 times slower*.

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 transitions. Figure 3(b) shows the

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

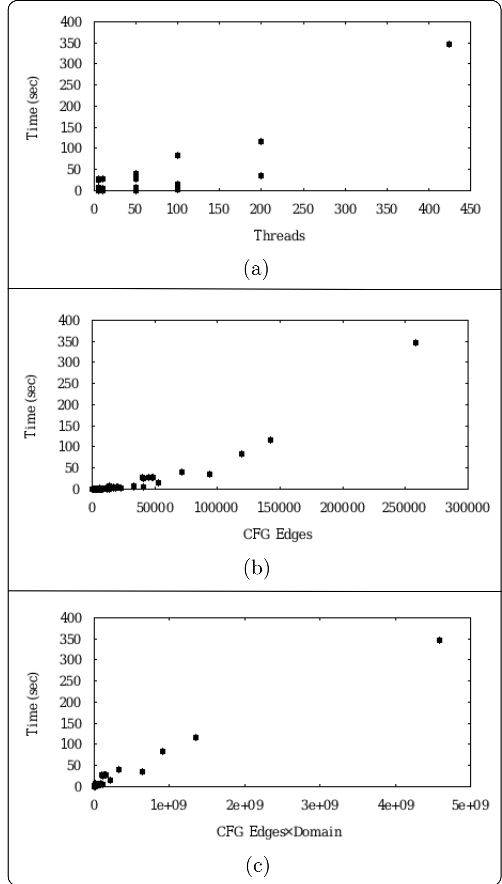


Fig. 3. Running time

6 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. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1986)
2. Chugh, R., Voung, J., Jhala, R., Lerner, S.: Dataflow analysis for concurrent programs using datarace detection. In: PLDI, pp. 316–326 (2008)
3. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural dataflow analysis. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 14–30. Springer, Heidelberg (1999)
4. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: POPL, pp. 1–11 (2000)
5. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks.Technical report, University of Toronto (2010), `http://www.cs.toronto.edu/~zkincaid/pub/cbva.pdf`
6. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL, pp. 303–314 (2007)
7. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
8. Kidd, N., Lammich, P., Touili, T., Reps, T.: A decision procedure for detecting atomicity violations for communicating processes with locks. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 125–142. Springer, Heidelberg (2009)
9. Knoop, J., Steffen, B., Vollmer, J.: Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. TOPLAS 18(3), 268–299 (1996)
10. Knoop, J.: Parallel constant propagation. In: Pritchard, D., Reeve, J.S. (eds.) Euro-Par 1998. LNCS, vol. 1470, pp. 445–455. Springer, Heidelberg (1998)
11. Krinke, J.: Static slicing of threaded programs. SIGPLAN Not. 33(7), 35–42 (1998)
12. Lammich, P., Müller-Olm, M.: Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 205–220. Springer, Heidelberg (2008)
13. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. ACM Commun. 18(12), 717–721 (1975)
14. Masticola, S.P., Ryder, B.G.: Non-concurrency analysis. In: PPOPP, New York, NY, USA, pp. 129–138 (1993)
15. Muchnick, S.S.: Advanced Compiler Design and Imlementation. Morgan Kaufmann, San Francisco (1997)
16. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that happen in parallel. In: FSE, pp. 24–34 (1998)
17. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An efficient algorithm for computing mhp information for concurrent java programs. In: ESEC/FSE-7, pp. 338–354 (1999)
18. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
19. Nielson, F., Nielson, H.: Type and effect systems. In: Correct System Design, pp. 114–136 (1999)

20. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
21. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP (2001)
22. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 351–365. Springer, Heidelberg (2000)