# Error Invariants for Concurrent Traces

Andreas Holzer[1] ⋆, Daniel Schwartz-Narbonne[2]⋆⋆, Mitra Tabaei Befrouei[3] ⋆⋆⋆, Georg Weissenbacher[3]⋆⋆⋆, and Thomas Wies[4] †

[1] University of Toronto
[2] Amazon
[3] TU Wien
[4] New York University

**Abstract.** Error invariants are assertions that over-approximate the reachable program states at a given position in an error trace while only capturing states that will still lead to failure if execution of the trace is continued from that position. Such assertions reflect the effect of statements that are involved in the root cause of an error and its propagation, enabling slicing of statements that do not contribute to the error. Previous work on error invariants focused on sequential programs. We generalize error invariants to concurrent traces by augmenting them with additional information about hazards such as write-after-write events, which are often involved in race conditions and atomicity violations. By providing the option to include varying levels of details in error invariants—such as hazards and branching information—our approach allows the programmer to systematically analyze individual aspects of an error trace. We have implemented a hazard-sensitive slicing tool for concurrent traces based on error invariants and evaluated it on benchmarks covering a broad range of real-world concurrency bugs. Hazard-sensitive slicing significantly reduced the length of the considered traces and still maintained the root causes of the concurrency bugs.

## 1 Introduction

Debugging is notoriously time consuming. Once a program failure has been observed, the developer must identify a cause-effect chain of events that led to it. This task is complicated by the fact that the underlying failing execution trace can contain a large number of events that do not contribute to the failure.

Error invariants [6, 2, 22] are (automatically generated) annotations of a given failing execution trace that can support the developer in his endeavor to narrow down the statements involved in the failure. Error invariants provide, for each point in the trace, an over-approximation of the reachable states that will produce a failure if execution of the trace is continued from that point (cf. Definition 7). Consequently, two subsequent

⋆ Funded by the Erwin Schrödinger Fellowship J3696-N26 of the Austrian Science Fund (FWF).
⋆⋆ Research was performed at NYU.
⋆⋆⋆ Supported by the Austrian National Research Network S11403-N23 (RiSE), the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005.
† Funded in part by the National Science Foundation under grant CCF-1350574.

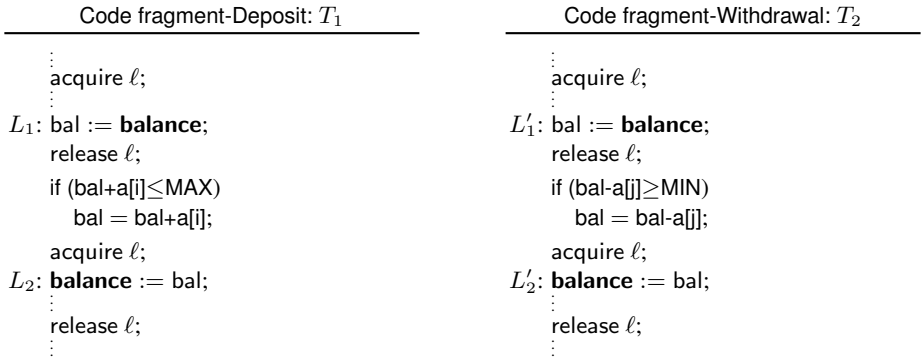



Fig. 1: Non-atomic update of bank account balance

error invariants in an erroneous execution reflect the relevance of the interjacent statement to the observed failure. Statements that leave the error invariant unchanged do not contribute to the failure and can be safely ignored during the failure analysis [22].

Intuitively, failure analysis with error invariants can be understood as a variant of dynamic slicing [28] that takes the semantics of the failure into account. Existing dynamic slicing techniques are based on data- and control-flow dependencies and remove statements which can not impact the failing state via any chain of dependencies. However, compared to error invariants the precision of these syntax-based slicing techniques is limited by the fact that the semantics of the erroneous trace is not taken into account.

Error invariants have been successfully deployed for constructing semantics-aware slices in sequential software. The enabling techniques for the automated generation of error invariants and slicing are *unsatisfiable cores* and *interpolation*. An error trace translated into an unsatisfiable first-order logical formula yields a proof of unsatisfiability from which interpolants can be extracted. These interpolants which correspond to assertions representing the error invariants can be used to construct a slice of the error trace that abstracts from the irrelevant statements and explains the faulty behavior. This approach produces a slice of the original trace annotated with assertions (the obtained error invariants) showing the relevant values and variables to the failure.

**Error Invariants for Concurrent Traces.** While error invariants faithfully reflect sequential control- and data-flow, concurrency aspects are ignored entirely. Consequently, a naive application of error invariants to concurrent traces leads to undesirable slices.

Consider, for example, the code fragments in Figure 1. At locations $L_2$ and $L_2'$, respectively, threads $T_1$ and $T_2$ update the balance of a bank account which is stored in the shared variable balance. The array a contains the sequence of 5 amounts to be transferred, partitioned into three deposits ($1 \leq i \leq 3$) and two withdrawals ($4 \leq j \leq 5$) executed by thread $T_1$ and $T_2$ in parallel, respectively. Figure 2 shows the suffix of a failing interleaved execution in which the third deposit is lost because of an atomicity violation. After three successful transactions (two deposits and one withdrawal) thread $T_2$ stores the current balance in a thread-local variable bal. At this point, $T_1$ interferes and updates the value of balance by performing the third deposit. Thread $T_2$, then, proceeds with the now stale value stored in bal and stores the result of the last with-

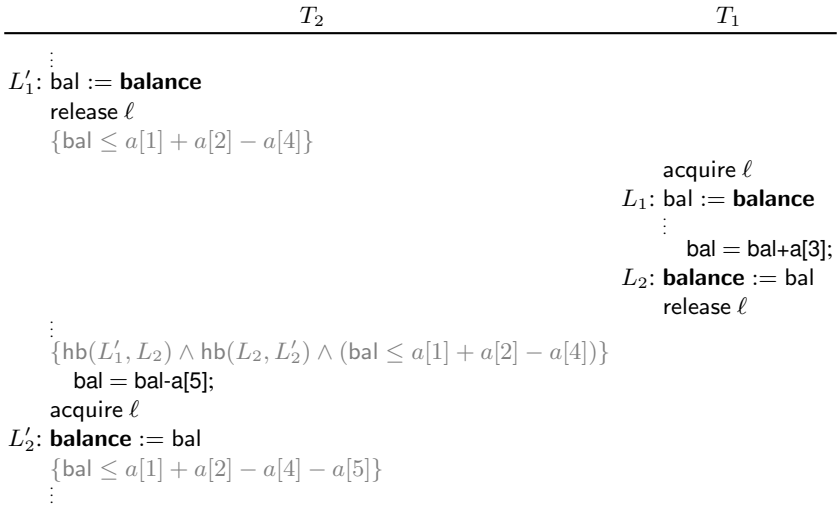



Fig. 2: Error trace with hazard-sensitive error invariants

drawal transaction in balance. Consequently, the execution results in a discrepancy of the expected and the actual balance on the account.

The problem is that the final value of balance depends on the sequence (or timing) of concurrently executed statements, i.e., the program contains a *data hazard*. As the statements are not executed in the order expected by the programmer, the hazard results in an erroneous state, which propagates to the end of the program where it surfaces as a failure. In this setting, the fault the programmer is looking for is the above-mentioned data hazard, in particular the write-after-write dependency between $L_2$ and $L_2'$.

The gray assertions in Figure 2 represent error invariants computed using the approach we propose in this paper. The assertion after $L_1'$ states that the local variable bal reflects at most two deposits and one withdrawal. At this point, the fault has not been triggered yet. The last conjunct in the error invariant after the context switch indicates that the value of bal is unchanged. The error invariants produced by previous techniques [6, 2, 22] track only the state information captured by this final conjunct. Therefore they would slice away all the statements of thread $T_1$ since the error invariants before and after the context switch would be identical. Thus, the resulting slice would not reflect the data hazard and not even the relevant interleaving.

To address this shortcoming, we lift interpolation-based slicing techniques to a concurrency setting by taking into account control and data dependencies between threads. The second assertion in $T_2$ (after the context switch) already reflects this adaptation: the expression $\mathsf{hb}(L_1', L_2) \wedge \mathsf{hb}(L_2, L_2')$ indicates that the statement at $L_1'$ happened before the statement at $L_2$, which in turn happened before the one at $L_2'$. This specific order is crucial to the failure. A slicing algorithm taking this information into account cannot safely slice the statement at $L_2$ in thread $T_1$ anymore. Note that, unlike previous techniques, error invariants in our approach not only reflect a set of states but also the execution order of critical statements via the happens-before relation (cf. Section 3.2).

Inter-thread data dependencies enable us to isolate (among other bugs) race conditions and atomicity violations which constitute the predominant class of non-deadlock

concurrency bugs [18]. Contrary to other concurrency debugging tools [5, 25, 8, 9, 23, 24] which target specific kinds of bugs, we provide a general framework for concurrency bug explanation. We applied an implementation of our approach to error traces generated from concurrent C programs using the directed testing tool ConCrest [7]. We evaluate our approach on benchmarks that contain bugs found in real-world software such as Apache, GCC, and MySQL [17]. On average, our slices yield a significant reduction of the number of variables and the length of the considered traces while maintaining information that is crucial to understand the underlying concurrency bug.

## 2 Preliminaries

**Syntax of Concurrent Programs.** A concurrent program comprises multiple threads each represented by its control-flow graph (CFG) [21, §7].

**Definition 1 (Control-Flow Graph).** *A CFG $\langle N, E\rangle$ comprises nodes $N$ and edges $E$. Each node $n \in N$ corresponds to a single programming construct from a simple imperative language comprising assignments* x:=$e$ *and conditions $R$.*

*Nodes representing conditional statements have two outgoing edges labeled* Y *and* N*, respectively, corresponding to the positive and negative outcome of the condition. All other nodes – except the exit node, which has no successors – have out-degree one.*

If a node $m$ is control dependent on a node $n$ and $n$ represents a condition, its outcome can determine whether $m$ is reached:

**Definition 2 (Dominators and Control Dependency).** *A node $m$ post-dominates a node $n$ if all paths to the exit node starting at $n$ must go through $m$. Node $m$ is control dependent on $n$ (where $n \neq m$) if $m$ does not post-dominate $n$ and there exists a path from $n$ to $m$ such that $m$ post-dominates all nodes (other than $n$) on that path.*

Based on Definition 2, we introduce our notion of a scope:

**Definition 3 (Scope).** *A node $m$ is in scope of the condition at node $n$ if $m$ is control dependent on $n$ or in scope of a condition that is control dependent on $n$.*

A CFG is in Static Single Assignment (SSA) form [3] if each variable is assigned exactly once. The standard mechanism to translate CFGs into SSA form is to subscript each definition of a variable with a unique version number; consequently, each definition is uniquely identified by the corresponding SSA variable. Conflicting definitions at a control-flow merge point $m$ in a CFG are resolved by introducing an arbiter node $n$ (with sole successor $m$) to which we divert the incoming edges of $m$. The arbiter node $n$ is annotated with a $\phi$-function which switches between the definitions from different incoming paths (see Figure 3). Algorithms to convert a program into SSA form are described in [3] and [21, §8.11].

**Definition 4 (Program Path).** *Let $\langle N_t, E_t\rangle$ be a CFG representing a thread $t$. A path $P_t$ of thread $t$ is a sequence $n_1, \langle n_1, n_2\rangle, n_2, \ldots, \langle n_{k-1}, n_k\rangle, n_k$ of nodes $n_i \in N_t$ and edges $\langle n_i, n_{i+1}\rangle \in E_t$. A program path $P \stackrel{\text{def}}{=} n_1, \langle n_1, n_2\rangle, n_2, \ldots, \langle n_{k-1}, n_k\rangle, n_k$*

*corresponds to an interleaving of paths of threads (starting at their respective initial nodes) such that for each $i$ with $1 \leq i < k$ either $n_i, n_{i+1} \in N_t$ and $\langle n_i, n_{i+1} \rangle \in E_t$ for some thread $t$, or $n_i$ and $n_{i+1}$ belong to different threads and $\langle n_i, n_{i+1} \rangle$ is an inter-thread edge representing a context switch.*

Given a (program) path $P$, let $[n_i, n_j]$ denote the sub-path $n_i$, $\langle n_i, n_{i+1} \rangle$, $n_{i+1}$, ..., $n_{j-1}$, $\langle n_{j-1}, n_j \rangle, n_j$ of $P$ including the nodes $n_i$ and $n_j$ and $(n_i, n_j)$ the sub-path $\langle n_i, n_{i+1} \rangle, n_{i+1}$, ..., $n_{j-1}, \langle n_{j-1}, n_j \rangle$ excluding the nodes $n_i$ and $n_j$. We use $P\downarrow_t$ to denote the projection of a program path $P$ to thread $t$ in which only nodes $n_i \in N_t$ and edges $\langle n_i, n_{i+1} \rangle \in E_t$ are retained and any sub-path $(n_i, n_j)$ with $n_i, n_j \in N_t$ and $n_l \notin N_t$ for $i < l < j$ is replaced with the edge $\langle n_i, n_j \rangle$, i.e., $P\downarrow_t$ is a path of the thread $t$. Consequently, for each program path $P$, $P\downarrow_t$ is either empty (if $P$ does not visit thread $t$) or a path of thread $t$ starting at the initial node of $t$. Finally, $P\downarrow_N$ and $P\downarrow_E$ denote the projection of $P$ to the sequence of nodes $N$ and edges $E$ in $P$, respectively.

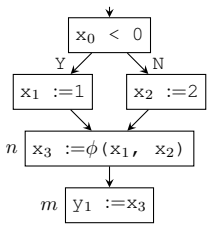


Fig. 3: SSA form of: `if (x<0) then x:=1 else x:=2; y:=x`

**Semantics, Feasible Executions and Error Traces.** The variables of a program are partitioned into *global* and *thread-local* variables. A state $s$ maps each variable to a value, and $s(e)$ denotes the value of expression $e$ in state $s$.

A program path $P$ corresponds to a sequence of statements. We require that each statement refers to at most one global variable, and hence statements execute atomically.

**Definition 5 (Execution).** *An execution of a path $P$ corresponds to an execution of the statements of $P$ in order (starting in an initial state). We use $\mathsf{stmt}_P(n_i)$ to denote the statement represented by node $n_i$ in a path $P$. In particular, if node $n_i$ represents the condition $R$, let $t$ be such that $n_i \in N_t$ and let $\langle n_i, n_j \rangle$ be the first edge in $P\downarrow_t$ succeeding $n_i$. Then $\mathsf{stmt}_P(n_i)$ is $R$ if $\langle n_i, n_j \rangle$ is labeled* Y*, $\neg R$ if the edge is labeled* N*. If $n_i$ is the last node of a thread $t$, then $\mathsf{stmt}_P(n_i) = \mathsf{true}$.*

*The execution of one statement in the current program state $s$ is defined as follows:*

- *If $\mathsf{stmt}_P(n_i)$ is the assignment $\mathtt{x}{:=}e$, the successor state of $s$ is updated such that $\mathtt{x}$ evaluates to $s(e)$ and all other variables are unchanged.*
- *If $n_i$ is a conditional statement $R$, the execution proceeds iff $s(\mathsf{stmt}_P(n_i))$ is true.*

A path $P$ is *feasible* if there exists an initial state $s$ for which the execution of $P$ is not blocked by a condition which is false. Given a path $P$, we use $\mathsf{stmts}_P$ to denote the sequence of statements represented by $P$. Abusing our notation, we sometimes call $\mathsf{stmts}_P$ a path and will use $P$ and $\mathsf{stmts}_P$ interchangeably.

We use $\mathsf{stmts}_P[i]$ to denote the $i^{\text{th}}$ statement $\mathsf{stmt}_P(n_i)$ of a path $P$, and $\mathsf{stmts}_P[i, j]$ to denote the sub-path $\mathsf{stmts}_P[i]$; ...; $\mathsf{stmts}_P[j]$ ($[n_i, n_j]$, respectively). We drop the subscript $P$ if it is clear from the context.

A state $s_j$ is reachable from a state $s_{i-1}$ via a sub-path $\mathsf{stmts}_P[i, j]$ if an execution of $\mathsf{stmts}_P[i, j]$ starting in $s_{i-1}$ does not block and results in state $s_j$.

We assume that the correctness of a path is determined by an assertion $\psi$ expected to hold after the execution of the path. Error traces which result in the violation of $\psi$ are defined as:

**Definition 6 (Error Trace).** *A path $P$ is an* error trace *for the assertion $\psi$ if $P$ is feasible and always results in a state $s$ such that $s(\psi)$ is false.*

Intuitively, an error trace is an execution of a failing test case that does not satisfy the specification $\psi$. We assume (w.l.o.g.) that path $P$ in Definition 6 reaches the end of the main thread, where $\psi$ is asserted. Consequently, $\psi$ is not in scope of any condition.

## 3 Error Explanation

In this section, we first recall the interpolation-based slicing approach presented in [6, 2] for sequential software. We then explain how we extend it to concurrent executions.

### 3.1 Interpolation-based Slicing for Sequential Traces

Ermis et al. [6] and Christ et al. [2] use *error invariants* to identify statements that do not contribute to the assertion violation in sequential traces.

**Definition 7 (Error Invariant).** *Given an error trace $P$ of length $k$ for assertion $\psi$, an error invariant for position $i$ (with $i \leq k$) is a set of states $E$ such that*
*(a) $E$ contains (at least) all states reachable from an initial state via $\mathsf{stmts}_P[1, i]$, and*
*(b) every feasible execution of $\mathsf{stmts}_P[i+1, k]$ starting from a state in $E$ results in a state in which $\psi$ is false.*
*An error invariant $E$ is* recurring[5] *for positions $i \leq j$ if $E$ is an error invariant for $i$ as well as for $j$.*

Intuitively, an error invariant $E$ represents an over-approximation of the states that are reachable via the path $\mathsf{stmts}[1, i]$ such that $\mathsf{stmts}[i+1, k]$ if executed from a state in $E$ still results in failure. According to [6, 2], statements between a recurring error invariant are "not needed to reproduce the error."

Error invariants can be derived using Craig interpolation (defined below) and a symbolic encoding of a path $P$ [6, 2]. In the following, we derive a symbolic encoding $\mathsf{enc}(P)$ similar to the one in [6] from a straight-line program in SSA form, which represents the path $P$ to be encoded. This straight-line program is obtained by traversing the CFG along $P$. If a node is visited repeatedly (via a cycle in one of the CFGs), a new version of the variable is introduced; for straight-line programs (which do not contain control-flow merge points) it suffices to increase the version number of a variable each time it is assigned and refer to the latest version of each variable in conditions and right-hand sides of assignments.

Given a path $P$ in SSA form as described above, the formula $\mathsf{enc}(P)$ is a conjunction $\bigwedge_{i=1}^{k} \mathsf{enc}_P(n_i)$ of the encodings of the individual statements:

$$\mathsf{enc}_P(n_i) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{x}_i = e) & \text{if } \mathsf{stmt}_P(n_i) \text{ is } \mathsf{x}_i := e \\ \mathsf{stmt}_P(n_i) & \text{if } \mathsf{stmt}_P(n_i) \text{ is a condition} \end{cases} \tag{1}$$

[5] To avoid confusion with inductive interpolant sequences (Definition 8), we replace the notion of *inductive error invariants* [6, 2] with recurring error invariants.

Variable assignments that satisfy formula $\mathsf{enc}(P)$ correspond to executions; note that if all variables in $P$ are initialized before being used, $\mathsf{enc}(P)$ has only one unique satisfying assignment. In this context, interpolants (as defined in [19]) are a symbolic representation of sets of states. Let $\mathrm{Var}(A)$ be the set of (free) variables occurring in a formula $A$. An interpolant $I$ is a predicate that encodes all states $s$ for which $s(I)$ is true. We define $\mathsf{states}(I) \stackrel{\text{def}}{=} \{s \mid s(I) = \mathsf{true}\}$. The following definition is a generalization of interpolants as defined in [19] under the assumption that all non-logical symbols in $A$ and $B$ are interpreted:

**Definition 8 (Inductive Interpolant Sequence).** *Let $A_1, \ldots, A_n$ be a sequence of first-order formulas whose conjunction is unsatisfiable. Then $I_0, \ldots I_n$ is an inductive interpolant sequence if*

- $I_0 = \mathsf{true}$ *and* $I_n = \mathsf{false}$,
- *for all* $1 \leq i \leq n$, $I_{i-1} \wedge A_i \Rightarrow I_i$, *and*
- *for all* $1 \leq i < n$, $\mathrm{Var}(I_i) \in (\mathrm{Var}(A_1 \wedge \ldots \wedge A_i) \cap \mathrm{Var}(A_{i+1} \wedge \ldots \wedge A_n))$.

Given a path $P \stackrel{\text{def}}{=} n_1, \langle n_1, n_2\rangle, n_2, \ldots, \langle n_{k-1}, n_k\rangle, n_k$ in SSA form, a sequence interpolant $I_0, \ldots, I_{k+1}$ derived from the formulas $\mathsf{enc}_P(n_1), \ldots, \mathsf{enc}_P(n_k), \psi$ is inductive in the sense that $\mathsf{states}(I_i)$ contains all states reachable from $\mathsf{states}(I_{i-1})$ via $\mathsf{stmt}(n_i)$ (and potentially more) [20, 22]. Moreover, $I_k \wedge \psi$ is not satisfiable, i.e., all states represented by $I_k$ violate assertion $\psi$. If $I_i$ represents an error invariant for positions $i$ and $j$ (i.e., $\mathsf{states}(I_i)$ is an error invariant for $j$ and $I_i$ implies $I_j$) then $I_i$ is inductive with respect to the sub-path $\mathsf{stmts}_P[i+1, j]$. Accordingly, slicing $[n_{i+1}, n_j]$ away (i.e, replacing it with an edge $\langle n_{i+1}, n_j\rangle$) preserves the assertion violation.

A trace obtained by removing statements between recurring error invariants from $P$ is sound in the sense of Definition 9 below:

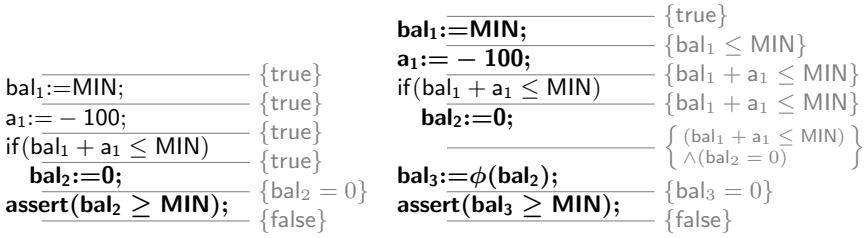
**Definition 9 (Sound Slice).** *A slice of path $P$ of length $k$ is a path $Q$ of length $m$ with $\mathsf{stmts}_Q[1] = \mathsf{stmts}_P[i_1]$, $\mathsf{stmts}_Q[2] = \mathsf{stmts}_P[i_2]$, …, $\mathsf{stmts}_Q[m] = \mathsf{stmts}_P[i_m]$ with $1 \leq i_1 < i_2 < \ldots < i_m \leq k$. Given an error trace $P$ for $\psi$, a slice $Q$ of $P$ is* sound *if $Q$ is also an error trace for $\psi$.*

### 3.2 Interpolation-based Slicing for Concurrent Traces

In the following, we enhance and extend the interpolation-based slicing technique discussed in Section 3.1 to take control dependency as well as concurrency into account.

**Control Dependencies.** The following example shows that the encoding $\mathsf{enc}(\mathsf{stmts})$ fails to capture control dependence (Definition 2).

*Example 1.* Figure 4a shows the statements of a path $P$ (in SSA form) and a corresponding interpolant sequence on the right. The example is a sequential variation of the bank account example which fails if the required minimum balance MIN is larger than zero. The resulting slice (indicated in bold) contains only the last assignment to bal and the assertion $\psi$. It does not reflect the fact that the Y-branch of the conditional statement has to be taken for the failure to occur.




(a) Control-insensitive slice

(b) Control-sensitive slice

Fig. 4: Slicing sequential trace with Error Invariants

We present a (modular) extension to the encoding defined in Section 3.1 that enables the inclusion of control dependencies. Unlike prior work [2], which addresses this problem using a custom-tailored control-sensitive encoding, our technique is based on the SSA representation. As in Section 3.1, the starting point of our approach is a straight-line representation of the error trace $P$. Unlike before, however, we include the $\phi$-nodes from the SSA presentation of the program in $P$:

**$\phi$-functions** at $n \in N_t$ for a variable $\mathsf{x}$, take as parameters the subscripted variable versions representing definitions of $\mathsf{x}$ in thread $t$ that reach $n$.

Consequently, when generating the straight-line presentation of $P$, we include all $\phi$-nodes of the SSA presentation of the program that are traversed by $P$. As we are encoding a single path $P$, however, $\phi$ takes only one parameter, since only one definition of each variable $\mathsf{x}$ reaches $n$ in $P$. Our extension $\mathsf{csenc}(\mathsf{stmts})$ of the encoding $\mathsf{enc}(\mathsf{stmts})$ is based on assignments $\mathsf{x}_i{:=}\phi(\mathsf{x}_j)$, which make control dependencies in an error trace $P$ explicit. In order for $\mathsf{x}_i$ to take the value of $\mathsf{x}_j$, the outcomes of the conditional statements preceding the assignment of $\mathsf{x}_j$ in $P$ have to permit the assignment to be executed.

Let $\mathsf{stmt}(n_j)$ be the statement assigning $\mathsf{x}_j$, and note that control dependency coincides with our notion of a scope (as defined in Definition 3). We define

$$\mathsf{guard}(n_j) \stackrel{\text{def}}{=} \bigwedge \{\mathsf{enc}(n_i) \mid n_j \text{ is in scope of } n_i\} \ . \tag{2}$$

In order for the definition of $\mathsf{x}_j$ in $n_j$ to be reachable along $P$, $\mathsf{guard}(n_j)$ needs to evaluate to $\mathsf{true}$. Moreover, since trace $P$ does not traverse alternative branches, the value of $\mathsf{x}_i$ is unknown if $\mathsf{guard}(n_j)$ does not hold. Based on this insight, we define a *control-sensitive* encoding $\mathsf{csenc}(P)$ as follows:

$$\mathsf{csenc}(n_i) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathsf{guard}(n_j) \Rightarrow (\mathsf{x}_i = \mathsf{x}_j) & \text{if } \mathsf{stmt}(n_i) \text{ is } \mathsf{x}_i{:=}\phi(\mathsf{x}_j) \\ & \quad \text{and } n_j \text{ assigns } \mathsf{x}_j \\ \mathsf{enc}(n_i) & \text{if } n_i \text{ is an assignment} \\ \mathsf{true} & \text{if } n_i \text{ is a condition} \end{cases} \tag{3}$$

An inductive error invariant for the encoding $\mathsf{csenc}(P)$ induces a control-sensitive slice (cf. Definition 4 of flow-sensitivity and Theorem 6 in [2]):

**Definition 10 (Control-sensitive Slice).** *Let $P$ be an error trace for the assertion $\psi$. A (sound) slice $Q$ is* control-sensitive *if for every statement $\mathsf{stmts}_Q[k] = \mathsf{stmts}_P[i]$ and every assumption $\mathsf{stmts}_P[j]$ such that $\mathsf{stmts}_P[i]$ is in scope of $\mathsf{stmts}_P[j]$, there is some prefix $\mathsf{stmts}_Q[1,h]$ of $\mathsf{stmts}_Q[1,k]$ (with $h < k$ such that $\mathsf{stmts}_Q[h]$ precedes and $\mathsf{stmts}_Q[h+1]$ succeeds or equals $\mathsf{stmts}_P[j]$ in $P$) such that $\mathsf{stmts}_Q[1,h]$ is an error trace for $\neg(\mathsf{stmts}_P[j])$.*

Intuitively, the definition requires that $Q$ justifies that every branch containing a relevant statement will be taken.

**Theorem 1.** *Let $P$ be a (concurrent) error trace for $\psi$ of length $k$ and let $I_0$, $I_1$ , ...,$I_{k-1}$, $I_{k+1}$ be error invariants (with $I_0 = \mathsf{true}$ and $I_{k+1} = \mathsf{false}$) obtained from an inductive sequence interpolant for $\mathsf{csenc}(n_1), \ldots, \mathsf{csenc}(n_k), \psi$. Let $Q$ be the slice obtained from $P$ by removing each sub-path $P[i,j]$ for which $I_{i-1}$ is inductive. Then $Q$ is a sound control-sensitive slice for $P$. (Proof in [11])*

Note that the interpolants in Theorem 1 may contain different versions of a variable x, since the encoding of $\phi$-nodes may refer to conditions in the "past". This corresponds to *history* or *ghost* variables used in Hoare logic and does not affect soundness.

*Example 2.* Figure 4b shows the path $P$ from Example 1 sliced using a control-sensitive encoding $\mathsf{csenc}(P)$ based on $\phi$-nodes. Note that the statements initializing bal and amount, which guarantee that the Y-branch is taken, are included in the slice.

**Synchronization.** In the simple interleaving semantics deployed in this paper, locks can be modeled using an integer variables $\ell$ and atomicity constraints. Lock $\ell$ is available if its value is 0. Any other value $t$ indicates that the lock $\ell$ is held by thread $t$. Let $n$ be a node of thread $t$ with a self-loop waiting for $(\ell = 0)$ to become true, and $m$ its successor node assigning $t$ to $\ell$. By constraining the execution such that no thread other than $t$ can execute between $n$ and $m$, we guarantee that lock acquisition is performed atomically. Analogously, a lock $\ell$ held by the current thread (guaranteed by condition $\ell = t$) is released by the statement $\ell{:=}0$. Control-sensitive slices also take into account lock acquisition statements, as relevant statements executed in a locked region are in the scope of the corresponding condition $(\ell = 0)$.

**Hazards.** A trace contains a *data hazard* if its outcome depends on the sequence (or timing) of concurrently executed statements. As explained for the sub-trace in Figure 2 discussed in Section 1, applying error invariants in their original form [6] to sequential paths results in slices that ignore important characteristics of concurrent traces. While $\mathsf{csenc}(P)$ reflects control-flow, it fails to capture *data dependencies*, which are constraints arising from the flow of data between statements [21]:

**Read-after-write** If statement $\mathsf{stmt}(n)$ writes a value read by statement $\mathsf{stmt}(m)$, then the two statements are *flow dependent*.

**Write-after-read** An *anti dependence* occurs when statement $\mathsf{stmt}(n)$ reads a value that is later updated (over-written) by $stmt(m)$.

| $T_2$ | $T_1$ |
|---|---|
| | $L_2$: $\mathbf{balance_1} := \mathsf{bal} + \mathsf{a}[3];$ |
| $L_2'$: $\mathbf{balance_2} := \mathsf{bal} - \mathsf{a}[5]$<br>$\mathbf{balance_3} := \pi(\mathbf{balance_1}, \mathbf{balance_2});$<br>$\mathsf{assert}(\mathbf{balance_3} = \mathsf{a}[1] + \mathsf{a}[2] + \mathsf{a}[3] - \mathsf{a}[4] - \mathsf{a}[5]);$ | |

Fig. 5: Part of a path with hazard and $\pi$-node

**Write-after-write** An *output dependence* exists if $\mathsf{stmt}(n)$ as well as $\mathsf{stmt}(m)$ set the value of the same variable.

While this definition also applies to single threads, we concern ourselves exclusively with *inter-thread* data dependencies. In a path $P$, a data dependency between different threads can indicate a conflicting access (i.e., a *race condition* or *hazard*).

Unlike flow dependence (which is taken into account by $\mathsf{enc}(P)$ and $\mathsf{csenc}(P)$, since the SSA form represents use-definition pairs and therefore also flow dependence explicitly), anti and output dependencies are not explicit in the SSA-based encoding of $P$ used in Sections 3.1 and 3.2. Similar to merge points in sequential programs, inter-thread dependencies in $P$ give rise to conflicting definitions of global variables. The Concurrent SSA (CSSA) form of paths presented in [29, 26] introduces $\pi$-functions to resolve dependencies between accesses to global variables in different threads.

To convert an error trace into CSSA form, we introduce an arbiter node before every read access to a global variable $\mathsf{x}$ in an error trace $P$ (analogously to the arbiter nodes for $\phi$-functions in Section 2). The arbiter node is annotated with a $\pi$-function that selects from all definitions of the global variable $\mathsf{x}$ in $P$ the most recent definition:

**$\pi$-functions** at $n \in N_t$ for a global variable $\mathsf{x}$, take as parameters the subscripted variables representing definitions of $\mathsf{x}$ in all threads.[6]

Figure 5 shows a simplified suffix of the trace in Figure 2. The simplified trace consists of two threads with a $\pi$-node (arbitrating between the definitions $\mathsf{balance}_1$ and $\mathsf{balance}_2$) inserted before an assertion $\psi$ that states the expected outcome. Note that unlike the degenerate $\phi$-functions used in Section 3.2, a $\pi$-function for $\mathsf{x}$ has as many parameters as there are definitions of $\mathsf{x}$ in $P$.

To encode WAR and WAW dependencies, we introduce an irreflexive, transitive, and anti-symmetric relation $\mathsf{hb}(n_i, n_j)$ which indicates that node $n_i$ is executed before node $n_j$. This happens-before relation enables us to encode the edges of a program trace, reflecting the program order and the schedule.

In addition, $\mathsf{rd}(\mathsf{x}, n_i)$ and $\mathsf{wr}(\mathsf{x}, n_j)$ indicate that $\mathsf{x}$ is read at node $n_i$ and written at node $n_j$. These primitives allow for an explicit encoding of data dependencies:

$$\begin{aligned} \mathsf{wr}(\mathsf{x}, n_i) \wedge \mathsf{hb}(n_i, n_j) \wedge \mathsf{rd}(\mathsf{x}, n_j) &\Leftrightarrow \mathsf{raw}_\mathsf{x}(n_i, n_j) \\ \mathsf{rd}(\mathsf{x}, n_i) \wedge \mathsf{hb}(n_i, n_j) \wedge \mathsf{wr}(\mathsf{x}, n_j) &\Leftrightarrow \mathsf{war}_\mathsf{x}(n_i, n_j) \\ \mathsf{wr}(\mathsf{x}, n_i) \wedge \mathsf{hb}(n_i, n_j) \wedge \mathsf{wr}(\mathsf{x}, n_j) &\Leftrightarrow \mathsf{waw}_\mathsf{x}(n_i, n_j) \end{aligned} \tag{4}$$

The hazard-sensitive encoding presented below incorporates data dependencies into the encoding of a trace. The encoding is derived directly from a program path $P$, taking

[6] As an optimization, only the *last* definition of $\mathsf{x}$ in thread $t$ before $n$ is added.

advantage of the information encoded in the edges. Assignments (without $\pi$-functions) are encoded as follows:

$$\mathsf{hsenc}(n_i) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathsf{wr}(\mathsf{x}, n_i) \wedge \mathsf{enc}(n_i) & \text{if } n_i \text{ writes global var. } \mathsf{x} \\ \mathsf{rd}(\mathsf{x}, n_i) \wedge \mathsf{enc}(n_i) & \text{if } n_i \text{ reads global var. } \mathsf{x} \\ \mathsf{enc}(n_i) & \text{otherwise} \end{cases} \tag{5}$$

Nodes $n_i$ with $\pi$-functions incorporate happens-before information. Let $n_i$ be a $\pi$-node assigning $\mathsf{x}_i$, let $n_j$ be an assignment to $\mathsf{x}_j$ and the last node before $n_i$ in $P$ updating the global variable $\mathsf{x}$. Then $\mathsf{hsenc}(n_i)$ is:

$$\mathsf{rd}(\mathsf{x}, n_i) \wedge (\mathrm{DEP}(n_i, n_j) \Rightarrow (\mathsf{x}_i = \mathsf{x}_j)) \tag{6}$$

where $\mathrm{DEP}(n_i, n_j)$ is the following condition:

$$\mathsf{raw}_{\mathsf{x}}(n_j, n_i) \wedge \bigwedge_{\substack{m \in \{n \in P \,|\, \mathsf{wr}(\mathsf{x}, n)\} \\ m \neq n_j}} (\mathsf{waw}_{\mathsf{x}}(m, n_j) \vee \mathsf{war}_{\mathsf{x}}(n_i, m)) \tag{7}$$

Intuitively, $\mathrm{DEP}(n_i, n_j)$ states that $\mathsf{x}_j$ is written before $\mathsf{x}_i$ is read, and no other definition of $\mathsf{x}$ interferes.

Finally, edges are encoded as happens-before relations:

$$\mathsf{hsenc}(\langle n_i, n_{i+1} \rangle) \stackrel{\text{def}}{=} \mathsf{hb}(n_i, n_{i+1}) \tag{8}$$

Given a path $P \stackrel{\text{def}}{=} n_1, \langle n_1, n_2 \rangle, n_2, \ldots, \langle n_{k-1}, n_k \rangle, n_k$, applying sequence interpolation to the formulas $\mathsf{hsenc}(n_1)$, $\mathsf{hsenc}(\langle n_1, n_2 \rangle)$, $\mathsf{hsenc}(n_2)$, …, $\mathsf{hsenc}(\langle n_{k-1}, n_k \rangle)$, $\mathsf{hsenc}(n_k)$, $\psi$ yields a sequence $\mathsf{in}_1, \mathsf{out}_1, \ldots, \mathsf{in}_k, \mathsf{out}_k$ of formulas such that

$$\mathsf{in}_i \wedge \mathsf{hsenc}(n_i) \Rightarrow \mathsf{out}_i \text{ and } \mathsf{out}_i \wedge \mathsf{hsenc}(\langle n_i, n_{i+1} \rangle) \Rightarrow \mathsf{in}_{i+1} .$$

Unlike before, $\mathsf{in}_i$ and $\mathsf{out}_i$ propagate facts about states as well as execution order. We can slice sub-path $[n_i, n_j]$ if $\mathsf{in}_i \Rightarrow \mathsf{out}_j$, sub-path $(n_i, n_j)$ if $\mathsf{out}_i \Rightarrow \mathsf{in}_j$, sub-path $[n_i, n_j)$ if $\mathsf{in}_i \Rightarrow \mathsf{in}_j$, and sub-path $(n_i, n_j]$ if $\mathsf{out}_i \Rightarrow \mathsf{out}_j$. The resulting sliced path $Q$ corresponds to a sequence of statements $\mathsf{stmts}_Q$ and a set of edges $Q\downarrow_E$ representing context switches and program order constraints relevant to the error.

**Definition 11 (Hazard-sensitive slice).** *Given an error trace $P$, a (sound) slice $Q$ is* hazard-sensitive *if for every statement $\mathsf{stmts}_Q[k] = \mathsf{stmts}_P[j]$ and statement $\mathsf{stmts}_P[i]$ such that there is an inter-thread data dependency between $\mathsf{stmts}_P[i]$ and $\mathsf{stmts}_P[j]$, there is an $h$ such that $\mathsf{stmts}_Q[h] = \mathsf{stmts}_P[j]$.*

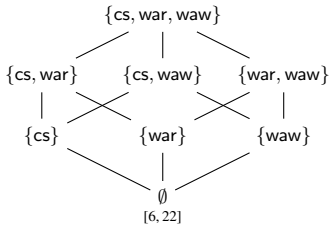
**Theorem 2.** *Let $P$ be a concurrent error trace and let $Q$ be the slice obtained from $P$ as explained above. Then $Q$ is a sound hazard-sensitive slice of $P$. (Proof in [11])*

*Example 3.* Consider the path in Figure 5. A hazard-insensitive slice would contain the statement at node $L_2'$ but not the statement at node $L_2'$ (as explained in Section 1) since $L_2$ has no influence on the state after $L_2'$. Encoding (6) and (7) of the $\pi$-node require the interpolant before the $\pi$-node to imply $\mathsf{waw}_{\mathsf{balance}}(L_2, L_2')$, and consequently $\mathsf{wr}(\mathsf{balance}, L_2)$, $\mathsf{wr}(\mathsf{balance}, L_2')$, and $\mathsf{hb}(L_2, L_2')$ (as indicated in Figure 2). Nodes $L_2$ and $L_2'$ as well as the edge $\langle L_2, L_2' \rangle$ are included in the resulting slice.

### 3.3 Fine-Tuning Explanations

The encodings presented in Section 3.2 can be combined in a straightforward manner, providing us with a choice of control WAR , and WAW dependencies reflected by the resulting explanation. Control-flow or hazard-sensitivity can be added (or removed) by (dis-)regarding $\pi$-nodes and $\phi$-nodes in $P$. Control-flow dependency can be incorporated into $\pi$-nodes in Equation (6) by prefixing the assignment $\mathsf{x}_i = \mathsf{x}_j$ with the guard of the definition of $\mathsf{x}_j$ at node $n_j$: $\mathsf{guard}(n_j) \Rightarrow (\mathrm{DEP}(n_i, n_j) \Rightarrow (\mathsf{x}_i = \mathsf{x}_j))$, similar to the guard in the definition of $\mathsf{csenc}(n_i)$ in Encoding (3). Moreover, Encoding (6) can be made insensitive to WAR dependencies by restricting $m$ to predecessors of $n_i$ and by dropping the disjunct $\mathsf{war}_\mathsf{x}(n_i, m)$ from (7) (and similarly for WAW dependencies). Note that flow dependency has a special role, since use-definition chains are explicit in the SSA representation.

The partial order given by the subset relation $\subseteq$ over the power-set of the remaining dependencies $\{\mathsf{cs}, \mathsf{war}, \mathsf{waw}\}$ reflects possible levels of detail of explanations, as illustrated by the Hasse diagram to the right. As indicated in the diagram, the configuration $\emptyset$ corresponds to the basic approach presented in [6, 22], whereas $\{\mathsf{cs}\}$ represents control-flow sensitive approach.




While we see interpolants as an inherent part of the explanation, the level of detail provided by these annotations cannot be related or formalized as easily as it is the case for dependencies: changing the underlying encoding typically has an unpredictable effect on the structure and strength of interpolants [4, 20].

## 4 Experiments

We implemented our approach as an extension of the directed testing tool ConCrest [7]. We generate error traces of concurrent programs and then produce slices as described in Section 3. While all slices provided by our tool are sound in the sense of Definition 9, the level of detail might not be sufficient to reflect the underlying bug: for example, the hazard-sensitive slice for the account benchmark readily reveals the atomicity violation. Therefore, it is not necessary to compute a more detailed control-sensitive slice.

The results from Section 3.3 enable the developer to gradually increase the detail in an iterative manner until the bug can be understood. This section provides an empirical evaluation of the size and accuracy of slices with varying levels of detail.

**Effectiveness of the Method.** To evaluate our method, we applied it on a collection of faulty C programs to show how effective the different dependency encodings are at revealing different types of concurrency bugs. We used four different encodings to track data and control dependencies: **hs** refers to hazard-sensitive encoding for tracking inter-thread data dependencies, **cs** refers to control-sensitive encoding for tracking control dependencies, and **ds** denotes the basic encoding $\mathsf{enc}_P$ of Section 3. The symbol "+" indicates combinations of encodings.

Our definition of whether the bug was captured depends on the type of bug. For data race bugs, we required that the slice reflecting the bug contains both conflicting accesses. For atomicity violations, a slice reflecting the bug contains conflicting statements from another thread interrupting the desired atomic region. For order violations, a slice reflecting the bug contains conflicting statements in the problematic order.

Table 1 summarizes our empirical results. The benchmarks in this table are classified into two groups. The first group consists of 33 multithreaded C programs taken from [17].[7] These programs capture the essence of concurrency bugs reported in various versions of open source applications such as Mozilla, Apache, and GCC. The apache2 and bluetooth benchmarks in the second group are simplified versions of applications taken from [7]. The pool-simple-2 benchmark is a lock-free concurrent data structure with a linearizability bug. We discuss this benchmark in depth in [11]. The remaining two benchmarks in the second group are variants of the program discussed in Section 1. For each benchmark program, the name, the number of lines of code (LOC), the number of threads, and the type of bug are listed in Table 1. The number of error traces (#T) per benchmark varies due to specific assertions and ConCrest's ability to produce error traces. They do not reflect any preselection of traces. In total, ConCrest generated 90 error traces from the 38 programs all of which we considered in our evaluation.

We use ✓ to indicate that the explanations obtained using the corresponding encoding capture the bug, and – if the bug was not captured. By manually inspecting the slices we found that for all but two benchmarks, tracking all dependencies **ds+cs+hs** yields explanations that capture the corresponding concurrency bug. For most benchmarks there exists at least one additional encoding which provides smaller slices that still reveal the bug. This encoding is usually **hs** (68%) or **cs** (50%) depending on the nature of the bug and the assertions. Interestingly, our analysis revealed that boop, freebsd_auditarg and gcc-java-25530 from [17] contain sequential bugs already reflected in a **ds**-slice rather than concurrency bugs (even though in [17] they are classified as concurrency bugs).

In two of the three error traces of freebsd_auditarg the bug is triggered by non-interleaved executions of the threads. For these traces, any encoding yields an adequate explanation. In one error trace, however, the bug is triggered by an interference between two threads, which is only reflected by the encodings **ds+hs** and **ds+cs+hs**.

Only the programs hash_table, ms_queue02, and list_seq, which contain bugs in intricate concurrent data structures, require the full **ds+cs+hs** encoding.

Only for the two benchmarks apache-25520 and cherokee_01 the slices produced by our method failed to reveal the bugs. The problem is that the root cause of the assertion violation is that a specific branch of a conditional statement is not taken during the execution. Slices of single error traces cannot reveal the non-occurrence of an event as the cause for failure. Therefore, we plan to analyze merged error traces in future work.

**Running times.** The generation of the slices takes an average of 2.43s ($\sigma = 11.02s$) across all encodings with a maximum of 168.8s. As expected, the running times increase with the amount of detail captured by the encoding. Generating a **ds** explanation takes 0.43s on average ($\sigma = 0.18s$) whereas a **ds+cs+hs** explanation takes 7.3s ($\sigma = 21.25s$).

[7] ConCrest's search heuristic failed to generate an error trace for the fibbench_longer, a variant of fibbench with larger parameters. We emphasize that this failure is related to the generation of traces rather than slicing.

| Benchmark | #T | LOC | | AIT | Threads | Bugs | ds+cs+hs | | | | | ds+hs | | | | | ds+cs | | | | | ds | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | S[%] | | V[%] | | | S[%] | | V[%] | | | S[%] | | V[%] | | | S[%] | | V[%] | | |
| | | | | | | | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | RB | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | RB | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | RB | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | RB |
| account | 3 | 43 | (58) | 51.7 | 4 | AV | **62** | **12** | **77** | **6** | ✓ | **42** | **10** | **68** | **6** | ✓ | 43 | 8 | 68 | 6 | – | 29 | 5 | 59 | 5 | – |
| apache-21287 | 2 | 30 | (79) | 43 | 3 | AV | **72** | **0** | **87** | **0** | ✓ | 28 | 0 | 53 | 0 | – | **51** | **0** | **87** | **0** | ✓ | 9 | 0 | 40 | 0 | – |
| apache-25520 | 1 | 88 | (192) | 34 | 3 | AV | 38 | – | 50 | – | – | 9 | – | 33 | – | – | 26 | – | 50 | – | – | 9 | – | 33 | – | – |
| barrier_vf_false | 12 | 57 | (85) | 27 | 4 | AV | **70** | **0** | **80** | **0** | ✓ | 19 | 0 | 40 | 0 | – | **67** | **0** | **80** | **0** | ✓ | 15 | 0 | 40 | 0 | – |
| boop | 1 | 58 | (98) | 40 | 3 | SB | **38** | – | **47** | – | ✓ | **30** | – | **40** | – | ✓ | **35** | – | **47** | – | ✓ | **28** | – | **40** | – | ✓ |
| cherokee_01 | 1 | 88 | (188) | 28 | 3 | AV | 46 | – | 60 | – | – | 11 | – | 40 | – | – | 32 | – | 60 | – | – | 11 | – | 40 | – | – |
| counter_seq | 1 | 28 | (41) | 29 | 3 | DR | **72** | – | **90** | – | ✓ | **38** | – | **70** | – | ✓ | 52 | – | 80 | – | – | 31 | – | 60 | – | – |
| fibbench | 2 | 34 | (47) | 34 | 3 | AV | **94** | **3** | **97** | **3** | ✓ | **94** | **3** | **97** | **3** | ✓ | **88** | **3** | **97** | **3** | ✓ | **88** | **3** | **97** | **3** | ✓ |
| freebsd_auditarg | 3 | 52 | (104) | 37 | 4 | SB | **67** | **7** | **86** | **0** | ✓ | **32** | **5** | **64** | **0** | ✓ | **57** | **10** | **79** | **10** | ✓ (2/3) | **30** | **8** | **57** | **10** | ✓ (2/3) |
| gcc-java-25530 | 2 | 36 | (86) | 17 | 3 | SB | **35** | **0** | **40** | **0** | ✓ | **35** | **0** | **40** | **0** | ✓ | **24** | **0** | **40** | **0** | ✓ | **24** | **0** | **40** | **0** | ✓ |
| gcc-libstdc++-3584 | 1 | 40 | (104) | 37 | 3 | AV | **62** | – | **79** | – | ✓ | **35** | – | **64** | – | ✓ | 46 | – | 71 | – | – | 30 | – | 57 | – | – |
| gcc-libstdc++-21334 | 1 | 36 | (86) | 27 | 3 | OV | **63** | – | **78** | – | ✓ | **22** | – | **33** | – | ✓ | 48 | – | 78 | – | – | 15 | – | 33 | – | – |
| gcc-libstdc++-40518 | 2 | 40 | (104) | 23 | 3 | AV | **43** | **0** | **56** | **0** | ✓ | 30 | 0 | 56 | 0 | – | **39** | **0** | **56** | **0** | ✓ | 22 | 0 | 56 | 0 | – |
| glib-512624_02 | 2 | 50 | (94) | 27.5 | 3 | AV | **84** | **2** | **100** | **0** | ✓ | **47** | **3** | **80** | **0** | ✓ | 60 | 4 | 85 | 5 | – | 38 | 5 | 65 | 5 | – |
| hash_table | 1 | 51 | (114) | 69 | 3 | AV | **41** | – | **61** | – | ✓ | 4 | – | 21 | – | – | 29 | – | 54 | – | – | 4 | – | 21 | – | – |
| jetty-1187 | 1 | 24 | (98) | 26 | 3 | AV | **81** | – | **100** | – | ✓ | **35** | – | **78** | – | ✓ | 58 | – | 89 | – | – | 27 | – | 67 | – | – |
| lazy01_false | 2 | 39 | (55) | 23 | 4 | OV | **91** | **0** | **100** | **0** | ✓ | **65** | **0** | **100** | **0** | ✓ | **87** | **0** | **100** | **0** | ✓ | **61** | **0** | **100** | **0** | ✓ |
| lineEq_2t_01 | 1 | 35 | (58) | 52 | 3 | AV | **69** | – | **81** | – | ✓ | **46** | – | **71** | – | ✓ | **52** | – | **76** | – | ✓ | **37** | – | **67** | – | ✓ |
| linux-iio | 1 | 54 | (87) | 55 | 3 | DR | **40** | – | **59** | – | ✓ | **20** | – | **50** | – | ✓ | 27 | – | 41 | – | – | 16 | – | 32 | – | – |
| linux-tg3 | 1 | 93 | (115) | 167 | 3 | DR | **19** | – | **38** | – | ✓ | **13** | – | **36** | – | ✓ | **8** | – | **11** | – | ✓ | 2 | – | 9 | – | – |
| list_seq | 1 | 59 | (122) | 53 | 3 | AV | **58** | – | **95** | – | ✓ | 6 | – | 30 | – | – | 40 | – | 75 | – | – | 6 | – | 30 | – | – |
| llvm-8441 | 2 | 149 | (244) | 32.5 | 3 | AV | **74** | **4** | **92** | **0** | ✓ | **18** | **0** | **33** | **0** | ✓ | 55 | 7 | 83 | 8 | – | 12 | 0 | 33 | 0 | – |
| mozilla-61369 | 1 | 19 | (68) | 6 | 1 | OV | **67** | – | **100** | – | ✓ | **67** | – | **100** | – | ✓ | **67** | – | **100** | – | ✓ | **67** | – | **100** | – | ✓ |
| ms_queue02 | 1 | 67 | (97) | 66 | 3 | AV | **44** | – | **52** | – | ✓ | 5 | – | 20 | – | – | 35 | – | 48 | – | – | 5 | – | 20 | – | – |
| mysql5 | 1 | 21 | (27) | 28 | 3 | AV | **82** | – | **89** | – | ✓ | **46** | – | **67** | – | ✓ | 46 | – | 89 | – | – | 25 | – | 67 | – | – |
| mysql-644 | 1 | 68 | (165) | 16 | 3 | AV | **38** | – | **33** | – | ✓ | **38** | – | **33** | – | ✓ | 25 | – | 33 | – | – | 25 | – | 33 | – | – |
| mysql-3596 | 1 | 30 | (83) | 6 | 3 | DR | **100** | – | **100** | – | ✓ | **100** | – | **100** | – | ✓ | **67** | – | **100** | – | ✓ | **67** | – | **100** | – | ✓ |
| mysql-12848 | 1 | 51 | (142) | 14 | 2 | AV | **71** | – | **67** | – | ✓ | 43 | – | 50 | – | – | **50** | – | **67** | – | ✓ | 29 | – | 50 | – | – |
| read_write_false | 1 | 78 | (140) | 58 | 5 | AV | **17** | – | **27** | – | ✓ | **17** | – | **27** | – | ✓ | **17** | – | **27** | – | ✓ | **17** | – | **27** | – | ✓ |
| reorder2_false | 8 | 50 | (105) | 10.5 | 5 | AV | **86** | **14** | **100** | **0** | ✓ | **86** | **14** | **100** | **0** | ✓ | **62** | **8** | **100** | **0** | ✓ | **62** | **8** | **100** | **0** | ✓ |
| testconc02 | 1 | 15 | (19) | 9 | 2 | AV | **89** | – | **100** | – | ✓ | **89** | – | **100** | – | ✓ | 56 | – | 100 | – | – | 56 | – | 100 | – | – |
| transmission-1.42 | 1 | 25 | (78) | 5 | 3 | DR | **100** | – | **100** | – | ✓ | **100** | – | **100** | – | ✓ | **80** | – | **100** | – | ✓ | **80** | – | **100** | – | ✓ |
| VectPrime02 | 1 | 97 | (183) | 115 | 3 | AV | **25** | – | **68** | – | ✓ | **9** | – | **45** | – | ✓ | 18 | – | 59 | – | – | 7 | – | 36 | – | – |
| apache2 | 8 | 719 | (–) | 235.5 | 3 | AV | **8** | **2** | **9** | **2** | ✓ | 1 | 0 | 1 | 0 | – | **7** | **2** | **9** | **2** | ✓ | 1 | 0 | 1 | 0 | ✓ |
| bankaccount-lock-for-loop | 5 | 103 | (–) | 247 | 3 | AV | **46** | **2** | **44** | **2** | ✓ | **12** | **1** | **30** | **2** | ✓ | 40 | 2 | 42 | 2 | – | 9 | 1 | 23 | 3 | – |
| bankaccount-simple-lock | 2 | 50 | (–) | 45 | 3 | AV | **71** | **0** | **80** | **0** | ✓ | **31** | **0** | **60** | **0** | ✓ | 62 | 0 | 73 | 0 | – | 24 | 0 | 53 | 0 | – |
| bluetooth | 5 | 87 | (–) | 35.8 | 3 | AV | **42** | **0** | **63** | **0** | ✓ | 14 | 0 | 31 | 0 | – | **36** | **0** | **63** | **0** | ✓ | 11 | 0 | 31 | 0 | – |
| pool-simple-2 | 8 | 298 | (–) | 885.5 | 3 | LV | **30** | **1** | **58** | **2** | ✓ | 0 | 0 | 2 | 0 | – | **29** | **1** | **56** | **2** | ✓ | 0 | 0 | 2 | 0 | – |
| Total | 90 | | | | | | 58.8 | | 72 | | 88 | 35 | | 54 | | 47 | 45 | | 67.7 | | 61 | 27 | | 50.5 | | 22 |

**#T**: No. of Traces in Benchmark
**LOC**: Lines of Code[a]
AIT: Average No. of Instructions in a Trace
**ds**: Basic Encoding
**cs**: Control-Sensitive Encoding
**hs**: Hazard-Sensitive Encoding
S: Slice Size / Trace Size
V: No. of Variables in Slice / No. of Variables in Trace
$\mu$: Average
$\sigma$: Standard Deviation
**RB**: Reflects Concurrency Bug
AV: Atomicity Violation
SB: Sequential Bug
DR: Data Race
OV: Order Violation
LV: Linearizability Violation

[a] LOC excluding comments and blank lines; LOC in parentheses are as stated in [17].

Table 1: Experimental comparison of sensitivity-configurations for slicing

**Quantitative Evaluation.** Table 1 shows the effect of tracking different dependencies on the size of the slices. $\mu$ refers to average percentage reduction as the quotient of the number of remaining and original instructions, so smaller numbers mean smaller slices. As expected, increasing the sensitivity of the algorithm by tracking more dependencies leads to smaller reductions. However, as we have seen previously, the hazard-sensitive explanations (**ds+hs**), which capture the concurrency bugs in 68% of the benchmarks, on average contain 35% of the original instructions and 54% of the original variables. We gained the maximum reduction with the encoding (**ds**), however the resulting explanations reflected the concurrency bugs in only 23% of the benchmarks. The amount of reduction differs across benchmarks with a maximum of 93% for the apache2 benchmark program. Slices which are hazard- but not control-flow sensitive tend to be much smaller than slices which are control-flow sensitive, but not data-hazard sensitive.

## 5 Related Work

The original work on error invariants [6, 2] is discussed in Sections 2 and 3. Murali et al. [22] relate error invariants to unsatisfiable cores and consistency-based diagnosis. The latter is also implemented in ConcBugAssist [17], a repair tool for concurrent programs, and BugAssist [15] for the diagnosis of sequential bugs. Both BugAssist and ConcBugAssist take into account multiple traces simultaneously and can yield better accuracy in certain cases (e.g., benchmarks apache-25520 and cherokee_01 in Section 4). Neither [15, 17] nor [22] report branch conditions (or statements explaining why they hold). On the benchmarks from [17], we found that ConcBugAssist yields similar reduction ratios as our tool using the **hs+ds** encoding. The dependency of ConcBugAssist on a bounded model checker for the constraint generation entails scalability issues: even on a simplified version of pool_simpl_2 for which we provided the minimal unwinding depth necessary to detect the bug, ConcBugAssist timed out after 45 minutes, while our approach generated a slice in 2.5 minutes for the non-simplified program.

Other static approaches for simplifying and summarizing concurrent error traces include [10], [12], [13], and [16]. In [10], an SMT solver and model enumeration is used to derive a symbolic representation of *all* reorderings of a given trace that violate a safety property, which is then used to explain the bug. Instead, we analyze a single failing trace, ensuring that our encoding explicitly captures which happens-before relations are relevant for the faulty behavior.

Tools that attempt to minimize the number of context switches, such as SIMTRACE [12] and TINERTIA [13], are orthogonal to the approach presented in this paper.

Many techniques for detecting race conditions or atomicity/serializability violations are geared towards specific bug characteristics [9, 30, 18]. Similarly, dynamic techniques such as Falcon [24] and Unicorn [23] rely on bug patterns. Our approach encodes data-dependencies rather than relying on bug patterns or specific bug characteristics. Recent work [27] uses mining of failing and passing traces to isolate erroneous sequences of statements. Our technique only considers failing traces.

AFIX [14] and CONCURRENCYSWAPPER [1] automatically fix concurrency-related errors. The latter uses error invariants to generalize a linear error trace to a partially ordered trace, which is then used to synthesize a fix. This approach may potentially benefit from our more fine-tuned trace encoding that enables error invariants to capture concurrent data dependencies.

## 6 Conclusion

We proposed to augment error invariants with information about inter-thread data dependency and hazards to capture a broad range of concurrency bugs. Our technique generates sound slices of concurrent error traces, enabling developers to quickly isolate and focus on the relevant aspects of error traces. We proved that the reported slices are sound and sufficient to trigger the failure. The experimental evaluation of our prototype implementation showed that the approach is effective and significantly reduces the amount of code that needs to be inspected.

## References


1. Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. Efficient synthesis for concurrency by semantics-preserving transformations. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 951–967. Springer, 2013.
2. Jürgen Christ, Evren Ermis, Matthias Schaef, and Thomas Wies. Flow-sensitive fault localization. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2013.
3. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.
4. Vijay D'Silva, Mitra Purandare, Georg Weissenbacher, and Daniel Kroening. Interpolant strength. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2010.
5. Dawson R. Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252. ACM, 2003.
6. Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In *Symposium on Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2012.
7. Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Foundations of Software Engineering (FSE)*, pages 37–47. ACM, 2013.
8. Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. *Communications of the ACM*, 53(11):93–101, 2010.
9. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation (PLDI)*, pages 338–349. ACM, 2003.
10. Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. Succinct representation of concurrent trace sets. In *POPL*, pages 433–444. ACM, 2015.
11. A. Holzer, D. Schwartz-Narbonne, M. Tabaei Befrouei, G. Weissenbacher, and T. Wies. Error Invariants for Concurrent Traces. *ArXiv e-prints*, abs/1608.08584, August 2016.
12. Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *Static Analysis Symposium (SAS)*, volume 6887 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2011.
13. Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *Foundations of Software Engineering (FSE)*, pages 57–66. ACM, 2010.
14. Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation (PLDI)*, pages 389–400. ACM, 2011.
15. M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Programming Language Design and Implementation (PLDI)*, 2011.
16. Sujatha Kashyap and Vijay K. Garg. Producing short counterexamples using "crucial events". In *Computer Aided Verification (CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2008.
17. Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In *ISSTA*, pages 165–176. ACM, 2015.
18. Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ACM Sigplan Notices*, volume 43, pages 329–339. ACM, 2008.
19. Kenneth L. McMillan. An Interpolating Theorem Prover. *Theoretical Computer Science*, 345(1):101–121, 2005.

20. Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Computer Aided Verification (CAV)*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.
21. Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann, 1997.
22. Vijayaraghavan Murali, Nishant Sinha, Emina Torlak, and Satish Chandra. A hybrid algorithm for error trace explanation. In *VSTTE*, 2014.
23. Sangmin Park, Richard Vuduc, and Mary Jean Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *Software Testing, Verification and Validation (ICST)*, pages 51–60. IEEE, 2012.
24. Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: fault localization in concurrent programs. In *International Conference on Software Engineering (ICSE)*, pages 245–254. ACM, 2010.
25. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
26. Nishant Sinha and Chao Wang. On interference abstractions. In *Principles of Programming Languages (POPL)*, pages 423–434. ACM, 2011.
27. Mitra Tabaei-Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *Runtime Verification (RV)*, 2014.
28. F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.
29. Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. 23(6):781–805, November 2011.
30. Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Programming Language Design and Implementation (PLDI)*, pages 1–14. ACM, 2005.