Abstract. A runtime analysis technique is presented, which can predict errors in multi-threaded systems by examining event traces generated by executions of these systems even when they are successful. The technique is based on a novel partial order relation on relevant events, called sliced causality, which loosens the obvious but strict “happens-before” relation by considering static structural information about the multi-threaded program, such as control-flow and data-flow dependence, and dynamic synchronization information, such as lock-sets. A vector clock based algorithm to encode the sliced causality is given, together with a procedure for generating all potential runs that are consistent with this partial order in a memory effective way. Then violations of properties can be “predicted” by running the corresponding monitor against potential runs that are consistent with the observed execution, i.e., permutations of (abstract) events that do not violate the sliced causal partial order. The monitors can be manually implemented or automatically synthesized from the desired properties, which can be given in any formalism that allows monitor synthesis algorithms. Our runtime analysis technique is sound, in the sense that it reports no false alarms. As expected, it is not complete; indeed, it cannot say anything about code that was not reached during the observed execution. A prototype system, called jPredictor, has been implemented and evaluated on several Java applications with promising results.

1 Introduction

A characteristic of concurrent systems in general and of multi-threaded systems in particular is that the same program with the same input may exhibit different behaviours when executed at different times. This inherent nondeterminism makes multi-threaded programs difficult to analyze, test and debug. This paper introduces a technique to correctly detect concurrency errors from observing execution traces of multithreaded programs. The program is automatically instrumented to emit “more than the obvious” information to an external observer, by means of runtime events. The particular execution that is observed needs not hit the error; yet, errors in other executions can be predicted without false alarms. The observer, which can potentially run on a different machine, will never need to see the code which generated those events but still be able to correctly predict errors that may appear in other executions, and report them to the user in a meaningful manner, by counter-example executions that explicitly reveal those errors.

There are several other approaches in the literature also aiming at detecting potential errors in concurrent systems by examining particular execution traces. Some of these approaches aim at verifying general purpose behavioral properties [27, 28], including temporal ones, and are inspired from efforts in debugging distributed systems based on Lamport’s “happens-before” causal partial ordering on runtime events [18]. Other approaches aim at dynamic behavior reduction and have been designed to work best for particular properties of interest, such as data-race and/or atomicity detection by means of lock-set algorithms [25, 14]. These previous efforts focus on either soundness or coverage: approaches based on the “happen-before” relation are sound but have limited coverage over interleavings, thus resulting in more false negatives (missing errors); lock-set based approaches produce fewer false negatives but suffer from false positives (false alarms). There are also works combining “happen-before” and lock-set techniques, e.g., [21], aiming at achieving a better balance between soundness and coverage, but these focus on particular properties to check, such as data-races, and do not use static information to increase coverage.

The approach presented in this paper focuses on improving the coverage without breaking the soundness. This is achieved by combining dynamic analysis, based on a special “happen-before” causal partial order, with static control-flow and dynamic data-flow dependency information of the multi-threaded program. This results in a much loosened causal partial order relation on events, which we call sliced causality, because, based on an apriori step of static analysis of the program’s code, it drastically cuts the usual “happen-before” causality by removing unnecessary dependencies; this way, a large number of consistent runs can be inferred and analyzed by the observer of the multithreaded execution. This novel causality relation leads to a practical technique for sound violation prediction of general-purpose properties, with significantly less coverage compromise than the other “happen-before” approaches.
One should not confuse the notion of sliced causality introduced in this paper with the existing notion of computation slicing [26]. The two slicing techniques are quite opposed in scope: the objective of computation slicing is to safely reduce the size of the computation lattice extracted from a run of a distributed system, in order to reduce the complexity of debugging, while our goal is to increase the size of the computation lattice extracted from a run, in order to strengthen the predictive power of our analysis by covering more consistent runs. Computation slicing and sliced causality do not exclude each other. Sliced causality can be used as a front end to increase the coverage of the analysis, while computation slicing can then remove redundant consistent runs from the computation lattice, thus reducing the complexity of analysis. At this moment we do not use computation slicing in our implementation, but it will be addressed in our future works to improve the performance of our analysis. Sliced causality can be used as a front end to increase the coverage of the analysis, while computation slicing can then remove redundant consistent runs from the computation lattice, thus reducing the complexity of analysis. At this moment we do not use computation slicing in our implementation, but it will be addressed in our future works to improve the performance of our analysis.

Our predictive runtime analysis technique can be understood as a hybrid of testing and model checking. Testing because one runs the system and observes its runtime behavior in order to detect errors, and model checking because the special causal partial order extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer. To avoid false alarms, the permutations of abstract events analyzed by the observer should be consistent with the semantics of the original program. Previous approaches based on the “happen-before” idea (such as [21, 27, 28]) extract causal partial orders from analyzing exclusively the dynamic thread communication in program executions. Since these approaches consider all interactions among threads, e.g., all reads/writes of shared variables, the obtained causal partial orders are rather restrictive, or rigid, in the sense of allowing a reduced number of linearizations and thus of errors that can be detected. Note that, in general, the larger the causal order (as a binary relation) the fewer linearizations it has, i.e., the more restrictive it is. By considering information about the static structure of the multi-threaded program in the computation of the causal partial order, we can filter out irrelevant thread interactions and thus obtain a more relaxed causality, allowing more consistent runs. Furthermore, we also take synchronization into account in our approach, in the sense that events protected by locks can only be permuted in blocks. This way, our approach borrows comprehensiveness from lock-set approaches without giving-up soundness. Moreover, our approach is fully generic: the possible linearizations that are consistent with the observed causal partial orders can be checked against any property on execution traces.

Figure 1 shows the classical example of the limitation of the happen-before technique [25]. When the execution proceeds as indicated by the arrow, the data race on y is masked by the protected accesses to x because the accesses to y are ordered by the write/read of x in the traditional happen-before approach. However, one can see that the accesses to y in the thread t2 do not depend on the accesses to x in the same thread; in other words, no matter what the value of x is, the accesses to y will occur anyway. Therefore, our approach drops the causal partial order caused by read/write of x and predicts the datarace on y in this successful execution.

Let us further explain our predictive runtime analysis technique on a more abstract example. Assume the threads and events in Figure 7, where e1 causally precedes, or “happens-before”, e2 (e.g., e1 writes a shared variable and e2 reads it right afterwards), and the statement generating e'1 is in the control scope (this notion will be formally defined in Section 4)1 of the statement generating e2, while the statement generating e3 is not in the control scope of e2. Then we say that e'1 is dependent upon e1, but that e3 is not dependent upon e1, despite the fact that e1 obviously happened before e3. The intuition here is that e3 would happen anyway, with or without e1 happening. Because of its combined static/dynamic flavor, we call this new dependence relation on events the hybrid dependence. Interestingly, if the events in the scope of e2 are not relevant for the property to check, then any permutation/linearization of relevant events consistent with the intra-thread total order and the hybrid dependence corresponds to some valid execution of the multithreaded system. Therefore, if any of these permutations violate the property, then the system can do so in a

---

1 For now, one can think of it as control-/data-flow dependence, e.g., e'1 may be generated by a statement in the “then” branch of a conditional statement that previously generated the read event e2 while evaluating its condition.
different execution. In particular, without any other dependencies but those in Figure 7, the property “e_1 must happen before e_3” can be violated by the program generating the execution in Figure 7, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e_1 should precede e_3, because e_3 would happen anyway. Also, note that a standard, purely dynamic “happens-before” approach would not work in this example.

We implemented our approach in a prototype system called jPredictor. The multi-threaded program to test is automatically instrumented to generate detailed execution traces and save them into log files. The log files are further filtered and finally analyzed by the tool. The prototype has been evaluated on several non-trivial applications and the results are promising. We were able not only to find known errors in large systems, but also reveal a wrong patch in the latest version of the Tomcat webserver.

More Motivating Examples

Below we first discuss the enhanced predictive ability of our technique by means of a more example, shown in Figure 3.

In this program, a mutex lock is implemented to allow the customizable light-weighted synchronization among threads. It provides two methods, namely, tryAcquiring to test and acquire the lock if possible and release to release the lock. tryAcquiring does not block the program; instead, it returns false if the acquiring fails, otherwise true is returned. This way, the thread can choose to wait for lock or continue to do other work first. This lock also counts the number of successful acquiring. However, this flexible implementation may cause tricky concurrent bugs if its behavior is not clearly specified. In this example, MyThread tries to count the number of executed threads using the shared variable MyThread.threadCount that should be protected by the lock MyThread.lock. However, the implementation expects tryAcquiring to block the execution as a normal lock acquiring does. This misuse essentially causes a datarace of MyThread.threadCount.

It is not straightforward to accurately catch this bug using existing techniques. The lockset algorithm can detect this datarace, will also report dataraces of lock.count since the update is not protected by any lock, but in fact accesses to lock.count only occur when the lock is successfully acquired, which means that it is well protected from dataracing. For “happen-before” technique, since the execution of the thread finishes in a short time, it is very likely that the threads run in order without interleaving. In such case, there is a causal partial order from the previous release of the lock to the following acquiring due to the write and read on the variable lock.flag. Therefore, no datarace can be detected. (CF: once I make JMPaX work, we can give some more details here.)

If we take the dependence into consideration, we can see that the accesses to count in tryAcquiring depend on the previous condition checking of variable flag, since the if statement can choose to return from the method without update count. Therefore, the causal order caused by writes and reads of flag should be considered when detecting dataraces of count, which then enforces a total order among accesses to count. This way, our approach knows that there is no dataraces of MyLock.count. On the other hand, there is no dependence from the update of MyThread.threadCount to the checking of lock.flag, since not matter the value of lock.flag is, the update will be executed. So we can ignore
the accesses to lock.flag when we try to detect the data race on MyThread.threadCount. Based on this observation, our approach shows that there are no causal relationship among those updates even when the threads are executed sequentially. The warning of potential dataraces is then reported.

Now let us look into a practical example to show the complexity of reality. Daisy [23] is a small file system that was devised to serve as a concrete system to stimulate and challenge the various software verification tools. Figure 4 shows the write functions on disk and the implementation of Mutex locks in Daisy. The disk is simulated using the RandomAccessFile class in Java. The whole disk is logically divided into blocks, which are used to store files. Every block or file is protected by a specific Mutex lock for performance reasons.

Although efforts have been made to assure the well-defined synchronization of the disk operations, there is a data-race reported [23]. The data-race is caused by the usage of the seek function of the disk, which sets a pointer for the next file operation. Since the pointer is unique for the whole file, when two threads try to read/write different blocks, they will compete for the pointer without synchronization locks. Since the read/write operations on the disk are not protected by the system lock, the lock-set based algorithms will produce a large number of false alarms for disk accesses, overwhelming the actual data-race. In our approach, because the loop in the lock implementation is regarded as a non-terminating one, the subsequent accesses to the disk have a control-flow dependence on the loop condition. Therefore, accesses to the same disk block (a continuous area in the underlying array) are ordered by the reads/writes on the shared Mutex lock, as shown in Figure 5. So no races are reported for accesses to the array. But when two threads try to access different blocks, they use different Mutex locks and therefore a race on the file pointer will be reported.

Let us further consider the buggy implementation of Mutex in Figure 6. In this code, the while loop is replaced by an if statement, which will cause errors if multiple threads are waiting for the same lock. However, because of the causal partial order caused by read/write on the locked field, the potential data-race will not be detected by the traditional causal partial order based approaches. In our approach, because accesses to the shared memory, including the file pointer and the specific block in the disk, are out of the control scope of the if statement, there is no causal partial order between the two writes in Figure 5. Therefore races will be reported.

The point we are trying to make here is that one can significantly improve over existing work in predictive runtime analysis if one considers structural information about the program, obtained via static analysis of its source code. Observers of multi-threaded runs can generate other potential valid runs that can yield sequences of states that are complete only up-to-relevant events. This way observers can explore more potential interleavings, increasing their predictive power.

2 Parametric Framework for Control Dependence

Before we can technically discuss the dependence-sliced causality in our approach, we need to define the important concept of control scope of a statement. Briefly, the scope of a statement $S$ is the set of statements whose reachability depends upon the potential choice at $S$. For example, in Figure 8 (A), the choice made at $C_1$ decides whether $S_1$ and $S_2$ are executed or not, but does not affect the execution of $S_3$. So scope ($C_1$) = \{ $S_1$, $S_2$ \} and scope ($S_1$) = scope ($S_2$) = $\emptyset$. 

![Fig. 4. Implementation of Disk and Mutex locks in Daisy](image1)

![Fig. 5. Writes on a block](image2)

![Fig. 6. Buggy lock acquiring](image3)
This can be computed entirely statically by examining the structure of the program. One may also notice that the concept of control scope can be connected to those of control dependences ([13, 22]) which have been extensively studied in program analysis. In what follows we introduce a generic parametric framework, which can be instantiated to capture the existing control dependences and also a new control dependence, named termination-sensitive control dependence.

Control dependence plays a fundamental role in program analysis, e.g., in program slicing [15, 30], in compiler optimization [13, 1], in total program correctness [22], in security (of information flows) [11], etc. Intuitively, a statement \( S \) control depends on a choice statement \( C \) iff the choice made at \( C \) determines whether \( S \) is executed or not. Because of the significance and broad range of applications of control dependence, related definitions and algorithms have been extensively investigated: [13] gives an efficient algorithm to compute the control dependence; [22] introduces the strong control dependence, also called the range of the control statement in [33], as well as the weak control dependence; [3] defines a generalized notion of control dependence to capture both the classic and the weak direct control dependencies, together with their afferent algorithms.

Although all these notions of control dependence are related to each other, there is no adequate unifying framework for all of them, not even a uniform or consistent terminology. This often results in confusion and difficulty in understanding existing works, and may slow future developments in the area, in particular defining new, or domain-specific control dependence relations. For example, the strong control dependence in [22] is the transitive closure of the control dependence in [13], contradicting common sense in formal terminology, because the former is actually weaker than the latter; the generalized control dependence in [3] addresses only the direct control dependencies (classic and weak), but omits the word “direct” in definitions and proofs, and it also proposes the terminology “loop control dependence” for (direct) weak control dependence; [22] claims that the strong control dependence is included in the weak control dependence, which appears quite intuitive, but it is non-trivial to prove rigorously. We believe that a rigorous development of a unifying framework for the various control dependences would enhance understanding and terminology in this area.

A first important step in this direction is made in [3], where a generalized control dependence is defined, parametrized by a property on paths. It generalizes both the classic and the weak direct control dependences. A linear time algorithm is also given in [3], to detect all the statements that depend upon a given choice statement. However, the parametric approach in [3] covers only the direct control dependence. The first part of our work, on parametric control dependence (Section 2.3), can be regarded as an extension of the work in [3] to also include indirect control dependencies, as well as comparisons of different instances. Our compact prefix-invariance property of the parameter is equivalent to the intersection of all the constraints on the parameter in [3] required by the different results, though we do not need to add a self-looping edge in the terminal node of the control-flow graph in order to capture the weak control dependence. We also develop a rigorous mathematical theory in Section 2.3, capturing formally many of the “folklore” results about different control dependencies.

As an instance of the parametric framework, we define a new control dependence relation that we call termination-sensitive control dependence, because it is sensitive to the termination information of loops, which can be given as annotations. If all loops are annotated as terminating then the termination-sensitive control dependence becomes the classic control dependence, while if all loops are annotated as non-terminating then it becomes the weak control dependence. Since in practice some loops are terminating and others are not, termination-sensitive control dependence is expected to improve the precision of analysis tools using it.

Our motivation for the termination-sensitive control dependence came from efforts in improving the accuracy and the coverage of predictive runtime analysis [7] of multithreaded systems. Since we refer back to it later in the paper, we explain this runtime analysis on a very simple example. Assume the threads and events in Figure 7, where \( e_1 \) causally precedes, or “happens-before”, \( e_2 \) (e.g., \( e_1 \) writes a shared variable and \( e_2 \) reads it right afterwards), and the statement generating \( e'_3 \) is in the control scope (this notion will be formally defined in Section 3) of the statement generating \( e_2 \), while the statement generating \( e_3 \) is not in the control scope of \( e_2 \). Then we say that \( e'_3 \) is dependent upon \( e_1 \), but that \( e_3 \) is not dependent upon \( e_1 \), despite the fact that \( e_1 \) obviously happened before \( e_3 \). The intuition here is that \( e_3 \) would happen anyway, with or without \( e_1 \) happening. Because of its combined static/dynamic flavor, we call this new dependence relation on events the hybrid dependence. Interestingly, if the events in the scope of \( e_2 \) are not relevant for the property to check, then any permutation/linearization of relevant events consistent with the intra-thread total order and the hybrid dependence corresponds to some valid execution of
the multithreaded system. Therefore, if any of these permutations violate the property, then the system can do so in a different execution. In particular, without any other dependencies but those in Figure 7, the property "e₁ must happen before e₃" can be violated by the program generating the execution in Figure 7, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e₁ should precede e₃, because e₃ would happen anyway.

The control scope of a statement is determined statically, as the set of statements that control depend on it. Unfortunately, the classic control dependence does not consider non-terminating loops, thus leading to false positives in the analysis, while the weak control dependence makes the worst case assumption (all loops are not terminating), resulting in over-restricted dependence among events and thus false negatives. The termination-sensitive control dependence takes the termination information of loops into account in order to build a more precise control dependence. We define this new control dependence as an instantiation of the parametric framework and show that it has properties similar to other control dependences. Interestingly, it follows as a corollary of the parametric framework that the termination-sensitive control dependence is stronger than the weak control dependence but weaker than the classic control dependence.

Finally, we describe an \(O(|V|^2)\) algorithm to compute the control scopes in the context of higher level programming languages, such as Java and C#. The detailed explanation and proof of the algorithm is out of the scope this paper. Due to space limitations, proofs are all omitted in this paper. The interested reader is referred to [7] for detailed proofs, as well as for more details on predictive runtime analysis.

2.1 Preliminaries.

A directed graph \(G\) is a pair \(\langle V, E \rangle\), where \(E \subseteq V \times V\). The elements of \(V\) are called nodes and those of \(E\) are called edges. A finite path of \(G\) is a finite sequence of nodes \(u_1u_2...u_{m+1}\) such that \((u_i, u_{i+1}) \in E\) for all \(0 < i \leq m\), where \(m > 0\) is its length. If \(u = u_1\) and \(v = u_{m+1}\) then we call this path a \(u \to v\) path. For any node \(u\), we let \(\lambda_u\) be the empty path from \(u\) to itself; its length is 0. An infinite path is an infinite sequence \(u_1u_2...\) such that \((u_i, u_{i+1}) \in E\) for all \(i > 0\). A \(u\)–path is a (finite or infinite) path starting with \(u\). We let \(\text{Paths}(G)\) be the set of all paths of \(G\), finite or infinite. For a path \(\pi\) either infinite or finite of length larger than or equal to \(k \geq 0\), we let \(\pi|_k\) be the path containing the first \(k\) edges of \(\pi\), i.e., \(u_1u_2...u_{k+1}\). We also define the concatenation of paths: if \(\alpha = u_1u_2...u_m\) finite and \(\pi = u_mu_{m+1}...\) finite or infinite, then \(\alpha\pi\) is the finite or infinite path \(u_1u_2...u_mu_{m+1}...\). A property of paths in a graph \(G\) is a set \(\mathcal{P} \subseteq \text{Paths}(G)\).

For any \(\pi \in \text{Paths}(G)\), we say that \(\mathcal{P}(\pi)\) holds, or simply \(\mathcal{P}(\pi)\), iff \(\pi \in \mathcal{P}\).

Definition 1. [13] A control flow graph \(\text{CFG} = \langle V, E, \text{START}, \text{END} \rangle\) is a directed graph \(\langle V, E \rangle\), together with an entry node, \(\text{START}\), from which every other node is reachable, and an exit node, \(\text{END}\), which is reachable from any other node. We make the standard assumption that nodes in \(V\) except \(\text{END}\) can have either one or two successors. Let \(V_C \subseteq V\) denote the set of nodes with two successors and call them choice nodes.

Intuitively, nodes in \(V\) correspond to statements in the program, edges in \(E\) indicate possible flows of control in the program, and choice nodes correspond to choice statements, such as conditionals, e.g., \(C_1\) in Figure 8 (A). Conditionals can also be parts of loops, e.g., \(C_1\) and \(C_2\) in Figure 8 (C). Because of the assumption on the bounded number of successors, \(|E| = O(|V|)\). Note that, in this paper, we tend to use letters at the beginning of the Greek alphabet, such as \(\alpha, \beta, \gamma, \text{ etc.},\) for \(u \to v\) paths, and letters \(\pi, \pi'\) and so on, for infinite or \(u \to \text{END}\) paths, though this convention is not strictly obeyed.

![Fig. 8. Some control flow graphs](image-url)
2.2 Control Dependence Revisited

There have been many studies on control dependence. We here discuss some of the major known results, introducing at
the same time a uniform notation and terminology. Some of the results in this section are mentioned in other works as
"folklore"; however, we were not able to find them proved formally in the literature; we will show that all these results
follow as corollaries of the general results in the next section. The structure of the results in this section anticipates the
structure of those for parametric control dependence in the next section. From here on we fix a CFG.

Classic Control Dependence

Definition 2. ([13, 11]) Node $u$ is post-dominated by node $v$, written $u \triangleright v$, iff all $u - END$ paths contain $v$. Let
PostDom($u$) be the set of post-dominators of $u$ except $u$.

We use the notation $u \triangleright v$ to symbolize the fact that no matter how we leave $u$ (the first two edges of the diamond),
we eventually converge (the other two edges of the diamond) and reach (the arrow) $v$. For example, in Figure 8 (A),
$C_1 \triangleright S_3$, while $S_1$ and $S_2$ do not post-dominate $C_1$; and in Figure 8 (B), $C_1 \triangleright S_2$, while $S_1$ is not a post-dominator
of $C_1$; however, note that in this figure there is no guarantee that $S_2$ will be reached once $C_1$ is reached, because of the
potentially infinite loop through $C_1$. In our context of predictive runtime analysis, this reflects a serious limitation of
the classic notion of control dependence; we will discuss this issue shortly.

Lemma 1. The post-domination relation, $\triangleright$, is a partial order on the nodes of the CFG.

Proof: The reflexivity is immediate. For transitivity, assume that $u \triangleright v$ and $v \triangleright w$. Then any $u - END$ path passes
through $v$, and since any $v - END$ path passes through $w$, it follows that any $u - END$ path passes through $w$, that is,$u \triangleright w$. For anti-symmetry, assume that $u \triangleright v$ and $v \triangleright u$, that is, that $v$ belongs to any $u - END$ path and $u$ belongs
to any $v - END$ path. If $u \neq v$, then one can immediately see the contradiction, because only one of $u$ or $v$ can appear
last on any finite path. Therefore, $u = v$. \hfill $\Box$

One can prove the following properties of post-dominance, while here they are just immediate corollaries of more
general results on parametric control dependence in Section 2.3.

Corollary 1. If $v_1 \neq v_2 \in$ PostDom($u$) then either $v_1 \triangleright v_2$ or $v_1 \triangleright v_2$, i.e., \langle PostDom($u$), $\triangleright$ \rangle is a total order. As a
consequence, for any $u$, if PostDom($u$) $\neq \emptyset$ then PostDom($u$) has a unique first element w.r.t. $\triangleright$.

Proof: It follows by Definition 12, Lemma 4 and Proposition 3. \hfill $\Box$

Definition 3. Let ipd($u$) be the first element of \langle PostDom($u$), $\triangleright$ \rangle, called the immediate post-dominator of $u$; let
$u \triangleright v$ iff $v = ipd(u)$.

The immediate post-dominator is the post-dominator that appears first on any $u - END$ path. For example, in Figure
8 (A), $C_1 \triangleright S_3$ since $S_3$ appears before any other post-dominators of $C_1$ on any path from $C_1$ to $END$; in Figure 8
(B), $C_1 \triangleright S_2$.

Proposition 1. $\triangleright$ is an inverted tree rooted by END.

Therefore one can represent $\triangleright$ as a post-dominance tree [20, 13] with END at its root. An $O(|V| \cdot \alpha(|V|))$ algorithm
to compute such trees in given in [20], where $\alpha$ is the inverse Ackerman function. Based on post-dominance, direct
control dependence can be defined as in [13]. Note that what we call “direct control dependence” below was called just
“control dependence” in [13]; we believe, for reasons given shortly, that its transitive closure should be called control
dependence.

Definition 4. Node $v$ is directly control dependent on node $u$, written $u \overset{dcd}{\sim} v$, iff

1. there exists a $u - v$ path $\alpha$ such that $v$ post-dominates every node in $\alpha$ different from $u$; and
2. $u$ is not post-dominated by $v$. 
For example, in Figure 8 (A), $S_1$ and $S_2$ are directly control dependent on $C_1$ but $S_3$ is not; while in Figure 8 (B), $S_1$ is directly control dependent on $C_1$ but $S_2$ is not. In Figure 8 (C), $S_1$ is directly control dependent on $C_1$ but not on $C_2$ (because $S_1$ does not post-dominate $C_1$). Note, however, that once $C_2$ is reached, the execution of $S_1$ depends on the control decision made at $C_2$! Therefore, $S_1$ control depends on $C_2$ by all practical, theoretical and intuitive means, suggesting that the terminology in [13] for control dependence is, perhaps, not the most appropriate one. We will shortly see that $S_1$ is in the transitive closure of the direct control dependence on $C_2$; for some reason, this transitive closure of the direct control was misleadingly called “strong control dependence” in [22]. We will call it simply “control dependence” in what follows, because we think it captures best the dependence of some statements on the control decision made by others. Note that direct control dependence is not a partial order on nodes: in Figure 8 (C), $C_1$ and $C_2$ are directly control dependent on each other.

The notion of direct control dependence has been widely used in program analysis, e.g., in program slicing [15, 30] and compiler construction [13], where it was called control dependence. As we have already mentioned, we prefer to call it “direct”, because it only considers direct dependence among statements; it does not reflect indirect dependence, e.g., the dependence from $C_2$ to $S_1$ in Figure 8 (C). Due to the hybrid dynamic/static setting of predictive runtime analysis, we need to also consider the “indirect” control dependence in the analysis. For example, in Figure 8 (C), if $C_1; C_2; C_1; S_1; S_2$ is an execution, the analysis needs to know that $S_1$ also depends on the choice made at $C_2$ to not exit the loop, which is caused by an indirect control dependence in the CFG.

In fact, even before the direct control dependence was introduced by Ferrante et al. [13], Dennings [11] already discussed the indirect influence of control statements on the program flow. Besides, Weiser [33] introduced a similar notion, called the range of branches, which is nothing but the transitive closure of the direct control dependence, as informally mentioned in [13, 24] without proof. Podgurski and Clarke [22] called it “strong control dependence”, to emphasize that it was stronger than their “weak” dependence, still without proving that it was the transitive closure of the direct control dependence—otherwise, they would have probably noticed the inconsistent terminology: for a relation $R$, which is the control dependence with the terminology in [22], “strong $R$” ended up strictly including $R$. For reasons explained above, we prefer to drop the adjective “strong” and call it just control dependence:

**Definition 5.** Node $v$ is control dependent on $u$, written $u \sim^c v$, if there exists some $u \overset{\alpha}{\rightarrow} v$ path that does not contain $ipd(u)$, the immediate post-dominator of $u$.

For example, in Figure 8 (C), $C_2 \sim^c S_1$. One can prove the following properties of the control dependence, all of which following from our parametric framework:

**Corollary 2.** For $\sim^cd$ and $\sim$, the following hold:
1. If $u \sim^cd v$ then $PostDom(u) \subseteq PostDom(v)$; in particular, $ipd(v) \sim id \mapsto ipd(u)$;
2. If $u \sim v$ then $PostDom(u) \subseteq PostDom(v)$; in particular, $ipd(v) \mapsto id \mapsto ipd(u)$;
3. $u \sim^c v$ iff there exists some $u \rightarrow v$ path $\alpha$ such that $\alpha \cap PostDom(u) = \emptyset$;
4. $\sim^c \subseteq \sim$, that is, $u \sim^c v$ implies $u \sim v$;
5. $\sim^c$ is transitive, that is, $u \sim^c v$ and $v \sim w$ implies $u \sim^c w$; and
6. $\sim = \sim^c$, that is, $u \sim v$ iff $u \sim^c v$.

**Proof:**
1. It follows by Definition 12, Lemma 4 and Lemma 5.
2. It follows by Definition 12, Lemma 4 and Lemma 6.
3. It follows by Definition 12, Lemma 4 and Lemma 7.
4. It follows by Definition 12, Lemma 4 and Lemma 6 (1).
5. It follows by Definition 12, Lemma 4 and Lemma 6 (2).
6. It follows by Definition 12, Lemma 4 and Lemma 6 (3).

Therefore, control dependence is nothing but the transitive closure of the direct control dependence, so it is weaker than the direct control dependence. Driven by common sense in mathematical terminology (“stronger” means more
restrictive and “weaker” means more general), we therefore took the liberty and brevity to suggest what we believe is a more appropriate terminology for control dependencies than the one in [22].

**Weak Control Dependence**

Although control dependence as defined above captures the “indirect” dependence as well, it still has another important limitation: it is insensitive to (non-terminating) loops; e.g., in Figure 8 (C), $S_2$ is *not* control dependent on $C_1$ because the former is the post-dominator of the latter. This may lead to incorrect predictive analysis of multi-threaded systems.

Re-consider the execution in Figure 7. Suppose it is generated by the program in Figure 8 (C). More specifically, suppose that $e_1$ is a write on the shared variable $j$, $e_2$ is the following read on $j$ generated by $C_1$, $e_3'$ is the write on $j$ generated by $S_1$, and $e_3$ is the write on $z$ generated by $S_2$. One may think that $e_3$ is *not* control dependent on $e_2$ by definition, that is, that $e_3$ will happen regardless of $e_2$. However, we can see that, since the loop is potentially non-terminating, $S_2$ may *never be executed* at runtime. Thus, the observed existence of $e_3$ is a consequence of a fortunate control choice made by $C_1$ when $e_2$ took place. So $e_3$ *should be control dependent* on $e_2$. Podgurski and Clarke [22] introduced strong post-dominance to handle control dependence in the presence of loops:

**Definition 6.** Node $u$ is strongly post-dominated by $v$, written $u \leadsto v$, iff

1. $u \iff v$ and
2. there is some integer $k \geq 1$ s.t. every $u-$path of length larger than or equal to $k$ passes through $v$.

Node $v$ is a proper strong post-dominator of $u$ if $u \leadsto v$ and $u \neq v$.

In other words, $u$ is strongly post-dominated by $v$ iff $u$ is post-dominated by $v$ and there is no infinite $u-$path that does not pass through $v$; e.g., in Figure 8 (B), $S_2$ does not strongly post-dominate $C_1$, because there is an infinite path from $C_1$ that will not pass through $S_2$, while in Figure 8 (D), $S_1$ is strongly post-dominated by $C_2$ but $C_2$ is not strongly post-dominated by $S_3$. There may be no proper strong post-dominators for some nodes; e.g., in Figure 8 (C), neither $C_1$ nor $C_2$ have proper strong post-dominators, since they can choose to either stay in the loop forever or jump out of it. Based on strong post-dominance, weak control dependence is defined in [22] as follows:

**Definition 7.** Node $v$ is directly weakly control dependent on $u$, written $u \dwcd v$, iff $u$ has successors $u'$ and $u''$ in the CFG such that $u' \dwcd wcd v$ but $u''$ is not strongly post-dominated by $v$; weak control dependence, written $\wcd$, is the transitive closure of $\dwcd$.

In Figure 8 (D), $C_1 \dwcd S_4$ because $S_2 \leadsto S_4$ but not $S_1 \leadsto S_4$. Weak control dependence is a generalization of control dependence, that is, every control dependence is a weak control dependence. This was informally mentioned in [22], but it is not straightforward to prove rigorously using their original definitions. However, it will follow as a corollary of more general results in our parametric framework, as shown at the end of Section 2.3. What makes this result even more interesting is that direct weak control dependence is not a generalization of direct control dependence. E.g., in Figure 8 (D), $S_3$ is directly control dependent but not directly weak control dependent on $C_1$, while it is directly weak control dependent but not directly control dependent on $C_2$. This may suggest that the terminology “direct weak control dependence” versus “direct control dependence” is also inappropriate, because the former is not weaker than the latter. However, this is not not problematic here because the adjective “weak” does not qualify the relation “direct control dependence”, but the relation “control dependence”; the terminology “weak direct control dependence” would have been indeed problematic. Like control dependence, weak control dependence is not a partial order either; e.g., in Figure 8 (C), both $C_1 \dwcd S_2$ and $C_2 \dwcd C_1$.

The (direct) weak control dependence makes the worst-case assumption that all loops are non-terminating, which is very rarely the case in practice. In fact, most loops in real programs do terminate. We next propose a parametric framework to define and reason about control dependence, which incorporates both the direct control dependence and the direct weak control dependence, as well as their transitive closures, as special cases. This framework can be easily instantiated to define other control dependence relations, such as the termination-sensitive control dependence discussed in Section 2.4 that we need for predictive runtime analysis.

### 2.3 Parametric Control Dependence

**Definition 8.** A set $P \subseteq \Paths(CFG)$ is a prefix-invariant property on paths iff
1. $P(\lambda_{\text{END}})$; and
2. $P(\alpha \pi) \Rightarrow P(\pi)$ for any $\alpha \pi \in \text{Paths}(\text{CFG})$ (\(\alpha\) is obviously finite).

From now on in this section, we fix a prefix-invariant property $P$. One can show that $P$ contains all $u - \text{END}$ paths, that is, that $P(\alpha)$ holds for any $u - \text{END}$ path $\alpha$.

**Definition 9.** A $u - \pi$ path is any $u - \pi$ path in $P$.

For any node $u$, there exists at least one finite $u - \pi$ path (END is reachable from $u$).

**Definition 10.** Node $u$ is $P$-post-dominated by node $v$, written $u \triangleright v$, iff all $u - \pi$ paths contain $v$. Let $\text{PostDom}_{P}(u)$ denote the set of $P$-post-dominators of $u$ different from $u$.

Note that for some nodes $u$, $\text{PostDom}_{P}(u)$ can be empty. For example, as shown after Definition 6, some nodes may not have strong post-dominators, which will be proved shortly to be a special case of $P$-post-dominators for a well chosen property $P$. The following says that $P$-post-dominance is stronger than classical post-dominance:

**Proposition 2.** $\triangleright \subseteq \triangleright p$, that is, $u \triangleright v$ implies $u \triangleright v$.

**Proof:** Suppose that $u \triangleright v$. Since any (finite) $u - \text{END}$ path is a $u - \pi$ path (by Definition 9), it follows that any $u - \text{END}$ path contains $v$. Therefore, $u \triangleright v$.

**Lemma 2.** $\triangleright$ is a partial order.

**Proof:** The reflexivity is immediate. For transitivity, assume that $u \triangleright v$ and $v \triangleright w$. Then any $u - \pi$ path passes through $v$. Since $P$ is prefix-invariant and any $v - \pi$ path passes through $w$, it follows that any $u - \pi$ path passes through $w$, that is, $u \triangleright w$. For anti-symmetry, assume that $v \triangleright u$ and $u \triangleright v$. Then we have $v \triangleright u$ and $u \triangleright v$ by Proposition 2, so $u = v$ by the anti-symmetry of $\triangleright$ (Lemma 1).

**Lemma 3.** If $u \triangleright v$ and there is a $u - u'$ path that does not contain $v$, then $u' \triangleright v$.

**Proof:** Suppose that $u \triangleright v$ and $\alpha$ is a $u - u'$ path that does not contain $v$. Let $\pi$ be a $u'$ path. Since $P$ is prefix-invariant, $\alpha \pi$ is a $u - \pi$ path. Therefore, $v \in \alpha \pi$, that is, $v \in \pi$.

**Proposition 3.** If $v_1 \neq v_2 \in \text{PostDom}_{P}(u)$, then either $v_1 \triangleright v_2$ or $v_2 \triangleright v_1$; in other words, $\langle \text{PostDom}_{P}(u), \triangleright \rangle$ is a total order. As a consequence, if $\text{PostDom}_{P}(u) \neq \emptyset$ then $\text{PostDom}_{P}(u)$ has a unique first element w.r.t. $\triangleright$.

**Proof:** As mentioned, there exists at least one $u - \pi$ path. For a $u - \pi$ path $\pi$, since $v_1, v_2 \in \text{PostDom}_{P}(u)$, $\pi$ contains both $v_1$ and $v_2$. Suppose that $v_1$ appears before $v_2$ on $\pi$, that is $\pi$ has the form $\alpha_1 v_1 \alpha_2$, where $\alpha_2$ is a $u - v_1$ path that does not contain $v_2$. Then $v_1 \triangleright v_2$ by Lemma 3. If $v_2$ appears before $v_1$ on $\pi$ then one can similarly show that $v_2 \triangleright v_1$.

**Definition 11.** Let $ipd_{P}(u)$ be the first element of the total order $\langle \text{PostDom}_{P}(u), \triangleright \rangle$, called the immediate $P$-post-dominator of $u$; let $u \triangleright v$ iff $v = ipd_{P}(u)$.

**Proposition 4.** $\triangleright$ is a forest of inverted trees.

**Proof:** According to Lemma 3, for any node $u$ with $\text{PostDom}_{P}(u) \neq \emptyset$, $u$ has only one successor w.r.t. $\triangleright$, namely $ipd_{P}(u)$. Therefore, each node in the CFG has at most one successor w.r.t. $\triangleright$.

One can show that post-dominance and strong post-dominance are two special cases of $P$-post-dominance by choosing appropriate parameters $P$. 


Definition 12. Let \( \mathcal{P}_\perp \) denote the set of all finite paths ending with END and let \( \mathcal{P}_{\perp \infty} \) be the union of \( \mathcal{P}_\perp \) with all infinite paths.

Lemma 4. Both \( \mathcal{P}_\perp \) and \( \mathcal{P}_{\perp \infty} \) are prefix-invariant.

Proof: Both \( \mathcal{P}_\perp \) and \( \mathcal{P}_{\perp \infty} \) contain \( \lambda_{\text{END}} \). \( \mathcal{P}_\perp \) is clearly prefix-invariant because, for any \( u \to v \) path \( \alpha \), \( \alpha v \in \) a \( u \to \text{END} \) path if and only if \( \pi \) is a \( v \to \text{END} \) path. Also, \( \mathcal{P}_{\perp \infty} \) is prefix-invariant because, for any \( u \to v \) path \( \alpha \), \( \alpha v \in \) a \( u \to \text{END} \) path or an infinite path if and only if \( \pi \) is a \( v \to \text{END} \) path or an infinite path. \( \square \)

Proposition 5. \( \Diamond \to = \Diamond \to \) and \( \Diamond \leftarrow = \Diamond \leftarrow \).

Proof: \( \Diamond \to = \Diamond \to \) follows by Definition 2, Definition 10 and Definition 12. For \( \Diamond \leftarrow = \Diamond \leftarrow \), suppose first that \( u \Diamond \to v \) and consider a \( u \Diamond \to \) path \( \pi \). If \( \pi \) is finite, i.e., a \( u \to \text{END} \) path, then \( v \in \pi \) because \( u \Diamond \to v \) by Definition 6. If \( \pi \) is infinite, then there is some \( k \geq 1 \) such that \( v \in \pi_k \), so \( v \in \pi \). Therefore, \( u \Diamond \to v \). Conversely, suppose \( u \Diamond \to v \). In particular, this means that any \( u \to \text{END} \) path contains \( v \), so \( u \Diamond \to v \). Now suppose that there is no \( k \geq 1 \) such that \( v \in \pi \) for any finite \( u \to \) path \( \pi \) with \( |\pi| \geq k \). In other words, for any \( k \geq 1 \), either there is no path longer than or equal to \( k \) or there is some path \( \pi \) such that \( |\pi| \geq k \) and \( v \notin \pi \). The first case means that there are only finite \( u \to \) paths, in which case \( \Diamond \to \) and \( \Diamond \leftarrow \) coincide, so \( u \Diamond \leftarrow v \). For the second case, since the CFG has a finite number of nodes, one can choose a large enough \( k \) such that, by the pigeon-hole principle, any finite \( u \to \) path \( \pi \) must contain a duplicate of some node \( w \) when \( |\pi| \geq k \). So we can have such a \( \pi \) in the form of \( owbfv \gamma \) and \( v \notin \pi \). We can then build an infinite \( u \to \) path \( \alpha(wbf)^\infty \) which does not contain \( v \). This contradicts the hypothesis. \( \square \)

We will discuss a third special case of \( \mathcal{P} \)-post-dominance in Section 2.4, where additional termination information of loops will be taken into account.

Definition 13. \( v \) is directly \( \mathcal{P} \)-control dependent on \( u \), written \( u \preceq_P v \), iff

1. there exists some \( u \to v \) path \( \alpha \) such that \( v \mathcal{P} \)-post-dominates all nodes in \( \alpha \) except \( u \), and
2. \( v \) does not \( \mathcal{P} \)-post-dominate \( u \).

Note that \( \preceq_P \) is not a partial order. For example, \( \preceq_d \) and \( \preceq_{\text{ord}} \), which will be shortly proved to be special cases of \( \preceq_P \), are not partial orders. This means that, in the worst case, the time needed to compute the transitive closure of \( \preceq_P \) using the standard transitive closure algorithms is \( O(|V|^3) \) [10]. In Section 3 we give an alternative \( O(|V|^2) \) algorithm that works on more restrictive CFGs, such as those obtained from programs in modern languages, e.g., Java and C#.

Lemma 5. If \( u \preceq_P v \) then \( \text{PostDom}_\mathcal{P}(u) \subseteq \text{PostDom}_\mathcal{P}(v) \); hence, \( \text{ipd}_\mathcal{P}(v) \preceq_P \text{ipd}_\mathcal{P}(u) \).

Proof: By Definition 13, there exists a \( u \to v \) path \( \alpha \), such that \( v \mathcal{P} \)-post-dominates any node in \( \alpha \) except \( u \). For any node \( u' \in \text{PostDom}_\mathcal{P}(u) \), \( u' \) cannot belong to \( \alpha \); otherwise, \( u' \Diamond \to v \), because \( v \mathcal{P} \)-post-dominates all nodes on \( \alpha \) except \( u \), and thus \( u \Diamond \to v \) by Lemma 2, which contradicts \( u \preceq_P v \). Suppose, by contradiction, that \( u' \) does not \( \mathcal{P} \)-post-dominate \( v \); then there exists a \( v \mathcal{P} \)-path \( \pi \) that does not contain \( u' \). Therefore, we can build a \( u \mathcal{P} \)-path, namely \( \alpha v \), that does not contain \( u' \), contradicting the fact that \( u' \in \text{PostDom}_\mathcal{P}(u) \). Hence \( \text{PostDom}_\mathcal{P}(u) \subseteq \text{PostDom}_\mathcal{P}(v) \). Then \( \text{ipd}_\mathcal{P}(u) \in \text{PostDom}_\mathcal{P}(v) \), so \( \text{ipd}_\mathcal{P}(v) \preceq_P \text{ipd}_\mathcal{P}(u) \). \( \square \)

Definition 14. Node \( v \) is \( \mathcal{P} \)-control dependent on \( u \), written \( u \sim_P v \), iff there exists some \( u \to v \) path that does not contain \( \text{ipd}_\mathcal{P}(u) \).

Lemma 6. If \( u \sim_P v \) then \( \text{PostDom}_\mathcal{P}(u) \subseteq \text{PostDom}_\mathcal{P}(v) \); hence, \( \text{ipd}_\mathcal{P}(v) \preceq_P \text{ipd}_\mathcal{P}(u) \).
Proof: We first prove that $ipd_P(u) \in PostDom_P(v)$. By Definition 14, there exists a $u \rightarrow v$ path $\alpha$ that does not contain $ipd_P(u)$. If $ipd_P(u)$ does not $P$-post-dominate $v$, then there exists a $v \rightarrow \pi$ path $\pi$ that does not contain $ipd_P(u)$. Therefore, we can build a $u \rightarrow \pi$-path, namely $\alpha \pi$, that does not contain $ipd_P(u)$, contradicting the definition of $ipd_P(u)$. Therefore $ipd_P(u) \in PostDom_P(v)$. For any node $u' \in PostDom_P(u)$, $ipd_P(u') \not\rightarrow* u'$; therefore, by Lemma 2, $v \not\rightarrow* u'$.

Lemma 7. $u \not\rightarrow v$ iff there exists some $u \rightarrow v$ path $\alpha$ such that $\alpha \cap PostDom_P(u) = \emptyset$.

Proof: It suffices to show that if $\alpha$ is a $u \rightarrow v$ path that does not contain $ipd_P(u)$ then $\alpha$ does not contain any $P$-post-dominator of $u$. Suppose, by contradiction, that $\alpha$ does contain some proper $P$-post-dominator $u'$ of $u$ different from $ipd_P(u)$, that is, that $\alpha$ has the form $\alpha_1 \cup \alpha_2$, where $\alpha_1$ does not contain $ipd_P(u)$. Since $ipd_P(u)$ does not $P$-post-dominate $u'$ (otherwise $u' = ipd_P(u)$ by Lemma 2), there is some $u' \rightarrow \pi$ path $\pi$ that does not contain $ipd_P(u)$. Since $ipd_P(u) \not\in \alpha_1$, it follows that $ipd_P(u) \not\in \alpha_1 \pi$, contradiction.

Proposition 6. The following hold:

1. $dP \subseteq \sim$;
2. $\sim$ is transitive; and
3. $dP = \simP$.

Proof:

1. Suppose that $u \not\rightarrow v$. In other words, there exists a $u \rightarrow v$ path $\alpha$ such that $v$ $P$-post-dominates all nodes in $\alpha$ except $u$. Then $ipd_P(u)$ cannot appear in $\alpha$ since otherwise $ipd_P(u) \not\rightarrow v$, implying that $u \not\rightarrow v$ by Lemma 2, which contradicts the definition of $dP$. Therefore $u \not\rightarrow v$.

2. Suppose that $u \not\rightarrow v$ and $v \not\rightarrow w$. Then there exists a $u \rightarrow v$ path $\alpha$ that does not contain $ipd_P(u)$ and a $v \rightarrow w$ path $\beta$ that does not contain $ipd_P(v)$. By Lemma 6, $ipd_P(v) \not\rightarrow ipd_P(u)$. If $ipd_P(u) = ipd_P(v)$, then $ipd_P(u)$ cannot appear in $\beta$. If $ipd_P(u) \neq ipd_P(v)$, then according to Lemma 2, $ipd_P(v)$ does not post-dominate $ipd_P(u)$. Thus, there exists an $ipd_P(u) \rightarrow \pi$ path $\pi$ that does not contain $ipd_P(v)$. Suppose that $ipd_P(u)$ appears in $\beta$, that is, that $\beta$ has the form $\beta_1 ipd_P(u) \beta_2$. Then we can build a $v \rightarrow \pi$ path $\beta \pi$ that does not contain $ipd_P(v)$, contradicting the definition of $ipd_P(v)$. So $ipd_P(u)$ cannot appear in $\beta$. Therefore, we have found a $u \rightarrow \beta \rightarrow w$ path $\alpha \beta$ that does not contain $ipd_P(u)$, that is, $u \not\rightarrow w$.

3. The first two items imply immediately that $dP \subseteq \sim$. For the other implication, suppose that $u \not\rightarrow v$ and let $\alpha$ be a $u \rightarrow v$ path such that $ipd_P(u) \not\in \alpha$. We prove by well-founded induction on the length of $\alpha$ that $u \not\rightarrow v$. Let $w$ be the last node on $\alpha$ which is not $P$-post-dominated by $v$. By Definition 13, it follows that $w \not\rightarrow v$. If $w = u$ then we are done. If $w \neq u$ then $u \not\rightarrow w$ by the induction hypothesis, so $u \not\rightarrow v$.

One can also show that direct control dependence and direct weak control dependence are two special cases of direct $P$-control dependence, while control dependence and weak control dependence are two special cases of $P$-control dependence:

Proposition 7. $dcd \sim \sim = \simP \rightarrow \simP = \simP$.

Proof: $dcd \sim = dP \sim$ follows by Definition 4, Definition 13, and Proposition 5. For $dwc \sim = dP \rightarrow$, since by Proposition 5, $\sim \rightarrow \rightarrow \rightarrow$, we use only $\rightarrow \rightarrow$ in this proof. Suppose that $u \sim v$. Then $u$ has two successors $u', u''$, such that $u' \not\rightarrow v$ and $u''$ is not strongly post-dominated by $v$. The latter implies that $u$ is not strongly post-dominated by $v$. The former first implies that there is some $u \rightarrow v$ path that does not contain $v$ except at its end, and then, by Lemma 3, that $v$
Proposition 8. \( cd \subseteq \sim \) \( \land \) \( \{ wcd \subseteq \sim \} \)

Proof: \( \sim = \sim \) \( \land \) \( \{ wcd \subseteq \sim \} \). 

The following proposition will allow us to compare control dependencies, based on just a simple comparison of their corresponding parameters:

Proposition 9. If \( \mathcal{P} \subseteq \mathcal{P}' \) are prefix-invariant properties then:

1. \( \sim \subseteq \sim \);
2. \( \sim \subseteq \sim \); 
3. \( \sim \subseteq \sim \); 
4. \( \sim \subseteq \sim \).

Proof:

1. If \( u \sim v \) then all \( u \sim v \) paths contain \( v \). Since \( \mathcal{P} \subseteq \mathcal{P}' \), all \( u \sim v \) paths are \( u \sim v \) paths. Then all \( u \sim v \) paths contain \( v \), that is, \( u \sim v \).
2. For any \( v \in PostDom_\mathcal{P}(u) \), that is, \( u \sim v \), by the first item, \( u \sim v \), that is, \( v \in PostDom_\mathcal{P}(u) \).
3. By the first item, \( u \sim ipd_\mathcal{P}(u) \). By Definition 11, \( ipd_\mathcal{P}(u) \sim ipd_\mathcal{P}(u) \).
4. By Lemma 7, we only need to prove that, for a \( u \sim v \) path \( \alpha \), if \( \alpha \cap PostDom_\mathcal{P}(u) = \emptyset \) then \( \alpha \cap PostDom_\mathcal{P}(u) = \emptyset \).

This follows by the second item.

Corollary 3. \( \sim \subseteq \sim \) for any prefix-invariant property \( \mathcal{P} \); in particular, \( cd \subseteq wcd \).

Proof: Since every finite path ending with \( END \) is a \( \mathcal{P} \) path, \( \mathcal{P}_3 \subseteq \mathcal{P} \). By Proposition 9 and Proposition 8, \( \sim \subseteq \sim \), and in particular \( cd \subseteq \sim \).

Interestingly, the inclusion of the direct versions of the dependences in the corollary above does not hold. For example, it is not the case that \( \sim \subseteq \sim \) (see the discussion following Definition 7).

2.4 Termination-Sensitive Control Dependence

Weak control dependence takes loops into account using strong post-dominance, which is more suitable for proving total correctness of programs [22] than the classic control dependence. However, weak control dependence unfortunately makes the worst-case assumption about the termination of loops in the program, namely, all loops are assumed to be potentially infinite. Considering the fact that most loops terminate in real programs, this assumption is too conservative in practice. Let us look at the example in Figure 8 (D). The loop containing \( S_1 \) and \( C_2 \) obviously terminates, so \( S_3 \) will be eventually executed once \( C_2 \) is reached. In other words, the execution of \( S_3 \) does not depend on the choice made at \( C_2 \). However, by Definition 7, \( C_2 \sim \rightarrow S_3 \). Such over-restrictive assumptions may bring false positives to static program analysis, while for our runtime predictive analysis, they may generate over-restrictive control dependences on events,
reducing the number of potential permutations of events when investigating possible actual executions, resulting in more false negatives, i.e., a reduced coverage.

In this section, we introduce a new control dependence relation, named termination-sensitive control dependence, as another instantiation of the parametric control dependence framework presented in Section 2.3. As indicated by its name, this control dependence takes the termination information of loops into account in order to improve the precision of program analyses that make use of control dependence. Although termination analysis is an undecidable problem, there exist some effective algorithms to approximately determine termination of programs, e.g., [9,4] (more discussion on these algorithms is out of the scope of this paper). Besides, termination information can also be provided by users (e.g., using special annotations) or detected by heuristics-based criteria (for example, a loop whose condition is \( i < n \) and in which \( i \) is increased at each iteration will always terminate). Here we only focus on defining a more precise control dependence relation using existing termination information, which is assumed to be correct.

First, we extend the CFG with termination information:

**Definition 15.** A termination-sensitive control flow graph \( \langle V, E, \text{START}, \text{END}, V_\infty \rangle \) is a CFG \( \langle V, E, \text{START}, \text{END} \rangle \) together with a distinguished set of nodes \( V_\infty \subseteq V \).

The nodes in \( V_\infty \) can be thought of as nodes that can lead to non-terminating executions. In practice, one would like to annotate as few statements as possible to provide the termination information; if that is the case, then \( V_\infty \) can contain precisely the conditions of those loops that may not terminate in some executions. Theoretically, one can add to \( V_\infty \) all the unavoidable statements in such loops, but this is not necessary. Besides, some of these statements can be themselves loops, but ones which terminate. From here on, we fix an arbitrary termination-sensitive CFG and define complete paths as follows:

**Definition 16.** A complete path \( \pi \) is a path either finite and ends with \( \text{END} \), or infinite and \( \text{inf}(\pi) \cap V_\infty \neq \emptyset \), where \( \text{inf}(\pi) \) gives those nodes visited infinitely often in \( \pi \). Let \( P_\uparrow \) denote the set of complete paths of the termination-sensitive CFG.

Note that infinite paths generated by “nested” loops in which the outer ones are annotated as “non-terminating” (in \( V_\infty \)), while the inner ones are “terminating”, are considered complete as far as the outer loop is executed infinitely often. One may be tempted to instead annotate the “terminating” nodes as a subset \( V_\uparrow \subseteq V \) and then require the complete path to satisfy \( \text{inf}(\pi) \cap V_\uparrow = \emptyset \); however, such an approach would be less precise, because it would exclude common paths as the ones generated by nested loops as above. There is an interesting similarity between termination-sensitive CFG and Buchi automata [5], where the role of accepting states is played by \( V_\infty \) and that of accepted words by complete paths.

One can show that \( P_\uparrow \) is also a prefix-invariant property on paths. Indeed, for any \( u \prec v \) path \( \alpha \) and \( v \prec \) path \( \pi \), \( \alpha \pi \) is a \( u \prec \text{END} \) path iff \( \pi \) is a \( v \prec \text{END} \) path. Besides, if \( \alpha \pi \) is infinite, then since \( \alpha \) is finite, \( \text{inf}(\alpha \pi) = \text{inf}(\pi) \). Therefore, \( \text{inf}(\alpha \pi) \cap V_\infty = \text{inf}(\pi) \cap V_\infty \); in particular, \( \text{inf}(\alpha \pi) \cap V_\infty = \emptyset \) iff \( \text{inf}(\pi) \cap V_\infty = \emptyset \). Based on the parametric framework for control dependence introduced in Section 2.3, we can define corresponding post-dominance and dependence notions: \( P_\uparrow \prec \text{-post-dominance} \) (\( \prec \)), immediate \( P_\uparrow \prec \text{-post-dominance} \) (\( \leadsto \)), direct \( P_\uparrow \prec \text{-control dependence} \) (\( \leadsto \)), and \( P_\uparrow \prec \text{-control dependence} \) (\( \tilde{\leadsto} \)). The following results follow immediately from the generic framework in the previous section:

**Corollary 4.** For \( \tilde{\leadsto} \), the following hold:
1. \( \tilde{\leadsto} \subset \tilde{\leadsto} \), that is, \( u \tilde{\leadsto} v \) implies \( u \tilde{\leadsto} v \);
2. \( \tilde{\leadsto} \) is a partial order;
3. If \( v_1 \not\in v_2 \in \text{PostDom}_{\prec} (u) \), then either \( v_1 \tilde{\leadsto} v_2 \) or \( v_2 \tilde{\leadsto} v_1 \); in other words, \( \langle \text{PostDom}_{\prec} (u) \rangle, \tilde{\leadsto} \rangle \) is a total order;
4. If \( \text{PostDom}_{\prec} (u) \neq \emptyset \) then \( \text{PostDom}_{\prec} (u) \) has a unique first element w.r.t. \( \tilde{\leadsto} \);
5. \( \tilde{\leadsto} \) is a forest of inverted trees;

**Proof:**
1. It follows by Proposition 2.
2. It follows by Lemma 2.
3. It follows by Proposition 3.
4. It follows by Proposition 3.
5. It follows by Proposition 4.

\[ \square \]

**Corollary 5.** For \( \sim_{\text{cd}} \) and \( \sim_{\text{tscd}} \), the following hold:
1. If \( u \sim v \) then \( \text{PostDom}_\alpha(u) \subseteq \text{PostDom}_\alpha(v) \); in particular, \( \text{idp}_\alpha(v) \sim \text{idp}_\alpha(u) \);
2. If \( u \sim v \) then \( \text{PostDom}_\alpha(u) \subseteq \text{PostDom}_\alpha(v) \); in particular, \( \text{idp}_\alpha(v) \sim \text{idp}_\alpha(u) \);
3. \( u \sim v \) iff there exists some \( u - v \) path \( \alpha \) such that \( \alpha \cap \text{PostDom}_\alpha(u) = \emptyset \);
4. \( \sim_{\text{cd}} \subseteq \sim_{\text{tscd}} \);
5. \( \sim_{\text{cd}} \) is transitive; and
6. \( \sim_{\text{tscd}} = \sim_{\text{cd}}^{-1} \).

**Proof:**
1. It follows by Lemma 5.
2. It follows by Lemma 6.
3. It follows by Lemma 7.
4. It follows Lemma 6 (1).
5. It follows Lemma 6 (2).
6. It follows Lemma 6 (3).

\[ \square \]

Now we are ready to define the termination-sensitive control dependence and to compare this new control dependence with the classical and weak control dependence:

**Definition 17.** Let \( \sim_{\text{tscd}} := \sim_{\text{cd}} \) be the termination-sensitive control dependence.

**Proposition 10.** \( \sim_{\text{cd}} \subseteq \sim_{\text{tscd}} \subseteq \sim_{\text{wd}} \).

**Proof:** Since \( \mathcal{P}_\perp \subseteq \mathcal{P}_\top \subseteq \mathcal{P}_{\text{wd}} \), by Proposition 9 and Definition 17, \( \sim_{\text{cd}} \subseteq \sim_{\text{tscd}} \subseteq \sim_{\text{wd}} \).

By Proposition 9, the set \( V_\infty \) acts as a “knob” tuning the precision of the control dependence relation. For example, if \( V_\infty = \emptyset \) then the termination-sensitive control dependence becomes precisely the classic control dependence. If \( V_\infty = V \) then it becomes the weak control dependence. In practice, \( V_\infty \) is somewhere in-between \( \emptyset \) and \( V \). However, the more nodes are added to \( V_\infty \), the more dependences are added, i.e., the weaker the dependence relation becomes. For example, in Figure 8 (C), suppose that \( C_2 \notin V_\infty \). Then \( S_2 \) is not termination-sensitive control dependent on \( C_2 \).

But if the user declares that \( C_2 \in V_\infty \) despite of the actual semantics of the program, we will have \( C_2 \sim S_2 \).

Ideally, one would like to pick a \( V_\infty \) which would generate a minimal set of complete paths \( \mathcal{P}_\top \) that includes all the actual execution paths of the program to analyze. Unfortunately, the selection of such an optimal \( V_\infty \) is difficult to achieve, because one would need to automatically prove termination of loops, an undecidable problem. A safe approach would be to start with \( V_\infty = V \), and then remove from it all the statements which are not loop conditions, then all those loop conditions controlling terminating loops which can be detected by heuristic criteria or declared so by users or code generators.

Interestingly, there are no inclusion relations between the direct versions of these control dependences, that is, between \( \sim_{\text{cd}} \) (or \( \sim_{\text{wd}} \)) and \( \sim_{\text{tscd}} \) or between \( \sim_{\text{tscd}} \) and \( \sim_{\text{wd}} \) (or \( \sim_{\text{cd}} \)). For example, consider the CFG in Figure 8 (D). Suppose first that \( C_2 \in V_\infty \) (i.e., the loop containing \( S_1 \) and \( C_2 \) is annotated as “non-terminating”). Then \( C_1 \sim S_3 \) but \( S_3 \) is not directly \( \mathcal{P}_\top \)-control dependent on \( C_1 \), while \( C_2 \sim S_2 \) but \( S_2 \) is not directly control dependent on \( C_2 \).

Suppose next that \( C_2 \notin V_\infty \) (i.e., the loop containing \( S_1 \) and \( C_2 \) is not annotated as “non-terminating”). Then \( C_1 \sim S_3 \) but \( S_3 \) is not directly weak control dependent on \( C_1 \), while \( C_2 \sim S_2 \) but \( S_2 \) is not directly \( \mathcal{P}_\top \)-control dependent on \( C_2 \).
3 Control Scope

The control scope of a conditional statement is the set of statements that control depend on it, where the control dependence relation is parametric (in the sense of the previous section) and indirect. In other words, a statement \( S \) is the control scope of \( C \) iff the execution of \( S \) depends upon a fortunate choice made by \( C \). Algorithms to compute the direct control dependence [13] and the direct weak control dependence [3] are well-known. These algorithms take linear time to detect all the statements that directly depend upon a given statement \( C \), and can be used to construct program dependence graphs (PDG) [15], which are widely adopted in program slicing. These linear algorithms to calculate control dependencies are sufficient in applications where high online speed is not crucial and where only the direct dependencies are necessary, such as debugging. However, there are applications that need the transitive versions of the control dependences. For example, in [22], the (indirect) weak control dependence is used to prove total correctness of programs. Also, in predictive runtime analysis, one prefers to calculate all the dependencies statically and then spend constant time at runtime to check whether the statements generating two events depend upon each other, to reduce the runtime overhead.

Calculating all the direct dependencies for all the statements statically can therefore be achieved in \( O(|V|^2) \). This also works for the termination-sensitive control dependence introduced in the previous section, because its parameter instance fits the framework in [3]. However, it is not clear how to effectively calculate the indirect control dependencies.

A blind application of the transitive closure of the direct control dependence would yield an \( O(|V|^3) \) algorithm (since the direct \( P \)-control dependence is not a partial order), which can be inapplicable even on relatively small programs. Without any additional information about the program which generates the CFG, it seems that there is nothing that one can do to decrease the complexity of calculating the \( P \)-control dependence. However, CFGs are typically generated from actual code that is stored as lines of sequences of characters in files. In what follows, we augment the CFG with code references and show that, under some rather common restrictions, we can calculate the entire \( P \)-control dependence relation in \( O(|V|^2) \), which is the same as the complexity of calculating the direct \( P \)-control dependence. It may seem that \( O(|V|^2) \) is still impractical in large applications; however, in the case of predictive runtime analysis or unit testing, we only need to calculate the control scopes for relatively small units, e.g., only intra-procedurally.

The nodes of a CFG generally correspond to either simple statements (i.e., statements that do not contain sub-statements) or to conditions that are parts of compound statements (i.e., statements that contain sub-statements). We only consider two types of compound statements in the sequel, namely conditionals and loops; note that although a programming language may also support other kinds of compound statements, e.g., try..catch, such statements are decomposed into simple statements when constructing the CFG. So they need not appear explicitly in the CFG (they appear only implicitly, encoded by corresponding edges). Even though CFGs capture faithfully the control flow of a program, unfortunately, precious structural information about the program, such as where a compound statement starts and where it ends, is generally not reflected in a CFG. In what follows we augment CFGs with structural information by adding to each node a corresponding unique line, or code reference number, which can be thought of as the position in the program where the statement corresponding to that node is located. The reference numbers of all nodes are assumed distinct. Since there is a one-to-one correspondence between (simple and compound) statements in the program and nodes in the CFG, we can identify statements with the reference numbers of their corresponding nodes in the CFG. Since the corresponding node in the CFG of a loop is its condition, the reference number corresponding to a statement is not necessarily the line number where that statement starts! For example, the reference number of the do...while loop in Fig. 10 (B) is 3. Let us formalize this:

**Definition 18.** A **sequential CFG** \((V, E, \text{START}, \text{END}, \#, b)\), abbreviated \( \text{SCFG} \), is a CFG together with injective functions \( \# : V \to \mathbb{N} \) and \( b : V_C \to \text{Intervals}(\mathbb{N}) \), such that

1. \( \#(C) \in b(C) \) for any \( C \in V_C \) and
2. \( b(C') \subseteq b(C) \) for any \( C \neq C' \in V_C \) with \( \#(C') \in b(C) \).

The function \( \# \) associates to each node in the CFG, corresponding to either a simple statement (whose out-degree is 1) or a condition (whose out-degree is 2) that is part of a compound (i.e., conditional or loop) statement, a unique reference number. The function \( b \) returns for each condition the code reference boundaries of its corresponding compound statement, given as an interval bounded by the smallest and the largest code reference numbers of nodes in the SCFG covered by that statement; some statements may include but not overlap other statements.

Fig. 10 shows the SCFGs for the programs in Fig. 9; we drew the nodes in ascending order of line numbers, and augmented every node with its reference number and each condition with its statement boundaries. The computation
can define a function \( \phi \) of the ple,

continue structured programming languages, such as Java and C#, impose additional restrictions on jump statements; for exam-

Definition 20. high-level structured programming languages:

time. We next define a corresponding version of SCFG that captures formally the restrictions on jumps encountered in

A Definition 20.

there is no proper sub-statement \( C \) of \( C \) that properly contains \( S \) then \( S \) is a direct sub-statement of \( C \).

The requirements of SCFGs are common to all programming languages that we are aware of. Most higher level structured programming languages, such as Java and C#, impose additional restrictions on jump statements; for example, continue, break, return, exception throwing, can only jump to specific positions determined statically at compile time. We next define a corresponding version of SCFG that captures formally the restrictions on jumps encountered in high-level structured programming languages:

Definition 20. A structured SCFG, abbreviated as SSCFG, is an SCFG \((V, E, START, END, \#, b)\) that satisfies:

1. Each compound statement has a unique entry point which is the lower bound of \( b(C) \), written \( entry(C) \); if \( \#(S) \not\in b(C) \) and \( next(S) \in b(C) \) then \( next(S) = entry(C) \);
2. Backward control flows can only be caused by loops: for any edge \((S, S') \in E\) with \( \#(S) > \#(S') \), there exists a compound statement \( C \) such that \( \#(S) \in b(C) \) and \( \#(S') = entry(C) \); in this case we call \( C \) a loop statement. All compound statements which are not loop statements are called conditional statements. For every loop statement \( L \), we also have a next function that points to the statement following \( L \). Formally, \( next(L) = max(\#(S_1), \#(S_2)) \) where \((L, S_1)(L, S_2) \in E\).

All the SCFGs in Fig. 10 are SSSCIF; nodes with references 3 in Fig. 10 (B) and 1 in Fig. 10 (C) are loop statements. Note that even though one could technically define loop statements in the context of (unstructured) SCFGs, they make full sense only in the context of SSSCIF, because in a SCFG one can construct a “loop statement” using a branch statement (e.g., a if statement) with an arbitrary jump (goto) statement.

We next focus on computing the control scope function of compound statements. Ideally, the control scope of a compound statement \( C \) would contain precisely the statements that are control-dependent on \( C \). Unfortunately, such
statements can be spread all over the program, thus making their precise bookkeeping rather challenging. However, in what follows we show that in the context of SSCFG, the statements that are control dependent on a compound statement \( C \) are all located into a window, or interval, of references, say \( \text{scope} (C) \), with the property that \( \text{scope} (C) \) contains no (reference of) statements that are forward-reachable (see Definition 22) from but not control dependent on \( C \). In other words, the interval, \( \text{scope} (C) \), characterizes unambiguously all the statements that are control-dependent on \( C \); if one wants to find precisely those statements control-dependent on \( C \), all one needs to do is to perform a simple (linear) reachability analysis from \( C \) and then all the statements in \( \text{scope} (C) \) that are not control-dependent on \( C \) can be easily filtered out. Moreover, in many applications one will only be interested in checking dependency for statements \( S \) that are, for external reasons, known to be reachable from \( C \). For example, in our runtime analysis approach, we need to check for control dependence of the statements that generated an observed event \( e \), upon the compound statement \( C \) that previously generated an event \( e' \); therefore, the very existence of the event \( e' \), as after the event \( e \), is a proof of reachability of \( S \) from \( C \). For such applications, our technique below to calculate control scopes can be very effective, because the checking control dependence reduces to checking membership to an interval. Moreover, we show, that one can calculate all the control scopes of a SSCFG in \( O(|V|^2) \), instead of \( O(|V|^3) \) that is needed for an unrestricted CFG.

We can easily see that all sub-statements of a compound statement are control dependent on it. Besides, a jump statement from within a compound statement \( C \) may extend the control scope of \( C \). For example, in Fig. 10 (C), the break statement extends the scope of the if-statement to the end of the loop, therefore statement 5 is control-dependent on the compound statement 2. This can be formalized as follows:

**Definition 21.** Suppose that \( C \) is a compound statement with \( \overline{b}(C) = [b_1, b_2] \). Then we define the pre-scope of \( C \), written \( \text{pre-scope}(C) \), as follows:

1. If \( C \) is a conditional statement then \( \text{pre-scope}(C) = [b_1, \max(b_2, \text{next}(J_1) - 1, \ldots, \text{next}(J_n)) - 1] \), where \( J_i \), for \( i \in [1, n] \), are all the direct sub-statements of \( C \); and
2. If \( C \) is a loop statement, \( \text{pre-scope}(C) = \overline{b}(C) \).

For example, in Fig. 10 (C), the pre-scope of the loop is \([1, 6]\) while the pre-scope of the if-statement is \([2, 6]\). The pre-scopes of loop statements are not extended by their direct sub-statements (when, e.g., an exception is thrown or a break/continue for an outer loop) because, as we discuss below, the backward edges of loops cause a different situation to handle. The pre-scopes of statements can be easily calculated by at no additional cost at parse time because the targets of jump statements are known statically (we focus on intra-procedure analysis here; if a statement throws an exception that is not caught in the analyzed procedure, it is assumed to jump to the end of the procedure). Note that the pre-scope of \( C \) may already contain statements that are not control-dependent on \( C \). Considering the example in Fig. 11: the pre-scope of the conditional statement 3 is \([3, 8]\) because of the continue statement in one of its branches. So statement 8 is within its pre-scope, but obviously not control-dependent on it. To filter out such statements, we next introduce a new relation between statements:

**Definition 22.** Statement \( S' \) is forward-reachable from \( S \) iff there exists an \( S - S' \) path that contains no loop statement containing both \( S \) and \( S' \).

For example, in Fig. 10 (C), node 3 is reachable but not forward-reachable from 4, and in Fig. 11, statement 8 is reachable but not forward-reachable from statement 3. Although the intuition for forward-reachability is that from \( S \) “one can go forward and reach” \( S' \), it is not always the case that one can find an \( S - S' \) path with increasing reference numbers. For example, in Fig. 11, statement 8 is forward-reachable from statement 2, but the path between them always contains statement 1, in other words, there is no path from 2 to 8 with increasing reference numbers. Forward-reachability can be determined by the following proposition:

**Proposition 11.** Given an SSCFG \( G \) and statements \( S \) and \( S' \), \( S' \) is forward-reachable from \( S \) iff \( S' \) is reachable form \( S \) in a graph \( G' \) constructed by transforming \( G \) as follows: for every (back) edge \( e = (n_1, n_2) \) in \( G \) with \( n_1 > n_2 \), corresponding to loop \( L \) (that is, \( \text{entry}(L) = n_2 \)), replace \( e \) by \( (n_1, @ - (L)) \).
First, it is obvious that all paths in $G'$ contain only increasing reference numbers and $G$ and if $S'$ is reachable from $S$ in $G'$ then $S'$ is reachable from $S$ in $G$. Suppose that $S'$ is not forward-reachable from $S$ in $G$. If $S'$ is not reachable from $S$ in $G$ then $S'$ is not reachable from $S$ in $G'$ either. If there exist $S - S'$ paths then all of them contain some loop $L$ which contains both $S$ and $S'$. In other words, all $S - S'$ paths contain an edge $e = (n_1, #(L)), n_1 > #(L)$. This edge is replaced with $(n_1, n_3)$ in $G'$ where $n_3 > #(S')$, which means that $S'$ is not reachable from statement $n_3$ in $G'$. So one cannot find a $S - S'$ path in $G'$, that is, to say, $S'$ is not reachable from $S$ in $G'$.

Now suppose that $S'$ is forward-reachable from $S$ in $G$ and $S$ is a $S - S'$ path that contains no loop that contains both $S$ and $S'$. Then for every loop $L$ contained in $S$, we keep only one iteration of $L$; if $L$ contains $S$ or $S'$ then the iteration to keep should go through $S$ or $S'$ correspondingly. If the loop exits at its entry, i.e., the while loop, then the path contains a sequence of edges $(n_1, #(L), #(L), n_2)$ where $n_1 > #(L)$ and $n_2$ is the reference number of the statement following $L$. We then replace these two edge by $(n_1, n_2)$ in $G'$. This way, we construct a $S - S'$ path in $G'$, that is to say, $S'$ is reachable from $S$ in $G'$.

Proposition 11 gives a very simple and effective to compute forward-reachability. Now we can have the following property for the pre-scope:

**Proposition 12.** For a (simple or compound) statement $S$ and a compound statement $C$, if $#(S) \in \text{pre-scope}(C)$ and $S$ is forward-reachable from $C$, then $C \leadsto S$.

**Proof:** If $C$ is a loop statement, since the pre-scope of $C$ is $b(C)$, any statement in the pre-scope of $C$ is control-dependent on $C$. Suppose that $C$ is a conditional statement and $S$ falls in the pre-scope of $C$ and is forward-reachable from $C$. If $#(S) \in b(C)$, then $C \leadsto S$. If $S$ is out of $b(C)$ (so the pre-scope of $C$ is larger than $b(C)$) and $S$ is not control-dependent on $C$ then all $C - S$ paths contain $ipd_p(C)$. Obviously, $ipd_p(C)$ is outside of $b(C)$. Let $b$ be the upper bound of $b(C)$, then, by Definition 21, there exists a statement $S'$ such that $#(S') = b$ and we can find a $C - S'$ path that does not contain any node within the pre-scope of $C$ but out of $b(C)$. If $#(ipd_p(C)) < b$, then there exists an $S' - ipd_p(C)$ path $\pi$ which contains a loop $L$ that contains both $S'$ and $ipd_p(C)$. Then $ipd_p(C)$ must be the node corresponding the loop; otherwise, the loop can choose to exit and skip $ipd_p(C)$ which is impossible. Moreover, $L$ should contain $S'$; otherwise, there exists a jump from outside of $b(L)$ into $b(L)$, contradicting to our assumptions on SSCFG. So every $C - S$ path contains $L$, contradicting to the hypothesis. If $#(ipd_p(C)) \geq b$ then any $ipd_p(C) - S$ path contains a loop $L$ that contains $ipd_p(C)$ and $S$. Similarly, $L$ contains $C$ because of our assumptions on SSCFG. So any $C - S$ path contains $L$, contradiction.

**Definition 23.** For a compound statement $C$, a control scope of $C$ is a an interval of reference numbers with the following properties:

1. all statements that are control dependent on $C$ are contained in the interval;
2. if a statements $S$ is within the interval and not control dependent on $C$ then $S$ is not forward-reachable from $C$.

For every compound statement, there can be multiple control scopes. In what follows we show that such control scopes exist and give an $O(|V|^2)$ algorithm to compute one of them. And by control scope, we mean the one computed by our algorithm from now on. The control scope of a compound statement can be larger than its pre-scope because of pre-scopes may overlap. Considering the control scope of the if statement in Fig. 10 (C), its control scope is $[1, 6]$ because its pre-scope overlaps the one of the outer loop.

To facilitate the following discussion, we abstract the sequential CFG to emphasize the pre-scopes of statements, as shown in Fig. 12. In this figure, the ranges of arrows give the pre-scopes of the statements, while the directions of the arrows distinguish the branch statement and the loop statement, that is, the forward arrow represents the branch statement and the backward one for the loop statement. According to the assumptions that we have, there are only two cases for overlapped pre-scopes, as shown in Fig. 12 (A) and (B). In the first case, $C_2$ is forward reachable from $C_1$. Then the control scope of $C_1$ is extended by that of $C_2$, because of the transitivity of control dependence. Consider the statement $S_1$ that resides out of the pre-scope of $C_1$. $C_1$ may choose to go into the branch containing $C_2$ and then $S_1$ will be skipped, that is, $C_1 \leadsto S_1$. In the second case $C_1$ and $C_2$ have the same control scope because of the backward jump caused.
by the loop $C_2$. For example, consider the statement $S_1$ before $C_1$. Its execution in the second iteration of the loop is dependent on the choice made at $C_1$ in the first iteration.

Based on the above observation, Fig. 13 shows the algorithm to compute the $\text{scope}_P$ function in $O(|V|^2)$ time based on the above observations. The computation process consists three steps. The first step is to extend pre-scopes of statements as in Fig. 12 (A) by a backward scanning. The second step is to compute equivalence classes of statements that have the same control scope by checking overlapped forward and backward conditionals (Fig. 12 (B)). We first build a graph to represent the overlapping among loops and other conditionals and then calculate connected components in the graph, which are essentially the desired equivalence classes. At the end, we compute the scopes of obtained equivalence classes. Note that this step can be adapted to compute different $P$-control scopes. For example, for the classical control dependence, the endln will never be set to infinity; while for the weak control dependence, the endln will be always infinity whenever the equivalence class contains a loop.

procedure ComputeScope()
ComputeFWReachability();
ExtendPreScope();
BuildEquivalentClasses();
ComputeEquivalentClassScope();
endProcedure

procedure ComputeFWReachability()
transform the original CFG into the corresponding non-loop CFG;
for every statement $S$ in the program do
    use depth-first search to compute the set of forward-reachable statements of $S$
    and the set of statements which can forward-reach $S$;
endFor
endProcedure

procedure ExtendPreScope()
for $S$ = the last statement downto the first statement do
    if ($S$ is a non-loop conditional) then
        for every non-loop conditional $S'$ that can forward-reach $S$ do
            if prescope($S$) overlaps prescope($S'$) then
                prescope($S'$) = prescope($S'$) U prescope($S$);
            endIf
        endFor
    endIf
endFor
endProcedure

procedure ComputeEquivalentClasses()
create a graph $G$ containing nodes corresponding to conditionals;
for every loop $L$ do
    for every non-loop conditional $C$ in prescope($L$) do
        if (prescope($L$) overlaps prescope($C$)) then
            create an edge between $L$ and $C$ in $G$;
        endIf
    endFor
endFor
compute connected component in $G$;
for every connected component $Cls$ do
    for every statement $S$ in $Cls$ do
        set($S$) = $Cls$;
    endFor
endFor
endProcedure

procedure ComputeEquivalentClassScope()
for every connected component $Cls$ do
    beginln = the smallest lower bound of pre-scopes of statements in $Cls$;
    if $Cls$ contains at least one non-terminating loop then
        endln = infinity; //infinity is the maximum integer in the system
    else
        endln = the largest upper bound of pre-scopes of statements in $Cls$;
    endIf
    scope($Cls$) = [beginline, endline];
endFor
endProcedure

Fig. 13. Compute the scope function

The output of this algorithm includes a $\text{prescope}_P$ function that maps an equivalence class into its control scope, and a function $\text{set}$ that maps a statement into the corresponding equivalence class. One can show that:

Lemma 8. For a conditional statement $C$ and a statement $S$, if $S$ is outside of scope (sets($C$)), then $S$ is not $P$-control dependent on $C$.

Proof: If $S$ is not reachable from $C$ then $S$ is not control-dependent on $C$. Suppose that $S$ is reachable from $C$, then there exists a $C - S$ path $\pi$. If $S$ is before $C$ then $\pi$ contains a loop $L$ containing both $C$ and $S$. Since $S$ is outside of
Proof: If program usually produces multiple events. Events need to store enough information about the program state in order to be a write on location \( \ell \), which is a write on location \( \ell \).

Events play a crucial role in our approach, representing atomic steps in the execution of the program. An event can be a write on location \( \ell \) or within a loop that has the same scope with \( C \) then by the algorithm, \( \#(S) \in \text{scope}(\{S\}) \), contradiction. So any \( C \sim S \) path \( \pi'' \) should contain a loop \( L \) containing both \( C \) and \( L \) is outside of \( \text{scope}(\{S\}) \), which means that \( \pi'' \) contains an edge that jumps from inside of \( \text{scope}(\{S\}) \) to the end of \( L \). If \( S \) is in \( L \) then by the algorithm, \( \#(S) \in \text{scope}(\{S\}) \), contradiction; but if \( S \) is outside of \( L \) then any \( C \sim S \) path should contain \( L \), so \( S \) is not control dependent on \( C \).

\( \square \)

Proposition 13. Under the assumptions of statements above, for a conditional statement \( C \) and a statement \( S \), \( C \leadsto S \) iff \( \#(S) \in \text{scope}(\{S\}) \) and one of the following holds:

1. \( S \) is forward-reachable from \( C \) or
2. there exists a loop \( L \) such that \( \text{sets}[C] = \text{sets}[L] \) and \( S \in \delta(L) \).

Proof: If \( S \) is control dependent on \( C \) then \( \#(S) \in \text{scope}(\{S\}) \) by Lemma 8. If there exists no loop \( L \) such that \( \text{sets}[C] = \text{sets}[L] \) and \( S \in \delta(L) \), \( S \) is not forward-reachable from \( C \). Then any \( C \sim S \) path \( \pi \) contains a loop \( L' \) containing \( C \) and \( L' \) is outside of \( \text{scope}(\{S\}) \). Then \( L' \) is a post-dominator of \( C \), so \( S \) is not control dependent on \( C \), contradiction.

Suppose that \( \#(S) \in \text{scope}(\{S\}) \). If there exists a loop \( L \) such that \( \text{sets}[C] = \text{sets}[L] \) and \( S \in \delta(L) \) then \( S \) is obviously control dependent on \( C \). Otherwise, if \( S \) is forward-reachable from \( C \) and not control dependent on \( C \) then any \( C \sim S \) path contains \( \text{ipd}(C) \). So \( \#(\text{ipd}(C)) \in \text{scope}(\{S\}) \), which is impossible according to the algorithm. \( \square \)

The complexity of \( \text{ComputeFWReachability}() \), \( \text{ComputePreScope}() \) and \( \text{ComputeEquivalentClasses}() \) is \( O(|V|^2) \) and \( \text{ComputeEquivalentClassScope}() \) is \( O(|V|) \). So the overall complexity of this algorithm is \( O(|V|^2) \).

4 Sliced Causality

Based on the control scope function, we are able to define the hybrid dependence on events, which are then used to slice the communication among threads. This way, one can achieve a more relaxed causal partial order on events, which we call sliced causality.

4.1 Events and Traces

Events play a crucial role in our approach, representing atomic steps in the execution of the program. An event can be a write/read on a location, the beginning/ending of a function invocation, acquiring a lock, etc. A statement in the program usually produces multiple events. Events need to store enough information about the program state in order for the observer to perform its analysis. Therefore, we define the notion of events in our approach as follows:

Definition 1 An event is a mapping of attributes into corresponding values. Let Events be the set of all events. A trace is a finite sequence of events. We assume an arbitrary but fixed trace \( \tau \), let \( \xi \) denote the set of events in \( \tau \) (also called concrete events), and let \( \prec \), be the total order on \( \xi \): \( e \prec e' \) iff \( e \) occurs before \( e' \) in \( \tau \).

For example, one event can be \( e_1 : (id = 17897, \text{thread} = t_1, \text{stmt} = L_{11}, \text{type} = \text{write}, \text{target} = a, \text{state} = 1) \), which is a write on location \( a \) with value 1, produced at statement \( L_{11} \) by thread \( t_1 \). One can easily include more information into an event by adding new attribute-value pairs. We use \( \text{attribute}(e) \) to refer to the value of \( \text{attribute} \) of event \( e \). To distinguish events with identical attributes, events are assigned unique identifiers when generated.

When the trace \( \tau \) is checked against a property \( \varphi \), most likely not all the attributes of the events in \( \xi \) are needed; some events may not even be needed at all. For example, to check data races on a variable \( x \), the states, i.e., the values of \( x \), of the events of type \text{write} and \text{read} on \( x \) are not important; also, updates of other variables or function call events are not needed at all. We next assume a generic filtering function that can be instantiated, usually automatically, to concrete filters depending upon the property \( \varphi \) under consideration:

Definition 2 Let \( \alpha_\varphi : \xi \to \text{Events} \) be a partial function, called a filtering function. The image of \( \alpha_\varphi \), that is \( \alpha_\varphi(\xi) \), is written more compactly \( \xi_\varphi \); its elements are called abstract relevant events, or simply just relevant events.
This abstraction will play a crucial role in increasing the predictive power of our analysis approach. That is because, in contrast to $\xi$, the more abstract $\xi_\phi$ will allow many more valid permutations of abstract events: instead of calculating permutations of $\xi$ and then abstracting them into permutations of $\xi_\phi$ like in [28], we will calculate directly valid permutations of $\xi_\phi$. Our goal is therefore to compute the precise causal partial order on abstract events in $\xi_\phi$ by analyzing the dependence among concrete events in $\xi$.

### 4.2 Hybrid Dependence

Without additional information about the structure of the program that generated the event trace $\tau$, the least restrictive causal partial order that an observer can extract from $\tau$ is the one which is total on the events generated by each thread and in which each write event of a shared variable precedes all the corresponding subsequent read events. This is investigated and discussed in detail in [29]. In this section we show that one can do much better than that if one uses appropriately control-flow and data-flow dependence information that can be obtained via static and dynamic analysis of the original program.

The notion of dependence discussed below somehow resembles that of program slicing [15, 30], but we focus on finer grained units here, namely events, instead of occurrences of statements as in program slicing. Our analysis keeps track of actual memory locations in every event, available at runtime, which avoids complicated inter-procedural analysis and eases the computation of the dependence relation. Also, we do not need to maintain the entire dependence relation among all the events, since we only need to compute the causal partial order among events that are relevant to the property to check. This leads to an effective vector clock (VC) based algorithm (Section 5.1).

Intuitively, event $e$ depends upon event $e'$ in $\tau$, written $\exists_e e'$, if a change of $e'$ may change or eliminate $e$. This tells the observer that $e'$ should occur before $e$ in any consistent permutation of $\tau$. There are two kinds of dependence: (1) control-flow dependence, written $e' \prec_{\text{ctrl}} e$, when a change of the state of $e'$ may eliminate $e$; (2) data-flow dependence, written $e' \prec_{\text{data}} e$, when a change of the state of $e'$ may lead to a change in the state of $e$. While the control-flow dependence relates events generated by the same thread of the multi-threaded program, the data-flow dependence relates to events generated by different threads: $e'$ may write some shared variable in a thread $t'$, whose new value is used for the computation of the value of $e$ in another thread $t$. Our notion of data-flow dependence in multi-threaded systems captures the intuition of dynamic dependence in distributed systems. It is this smooth combination of static control-flow dependence and dynamic dependence in the context of multi-threaded systems that suggested us the terminology hybrid dependence.

**Control-flow Dependence.** Informally, if a change of state($e$) may affect the occurrence of $e'$, then we say that $e'$ has a control-flow dependence on $e$, and write $e \prec_{\text{ctrl}} e'$. Control-flow dependence occurs inside of a thread, so we first define the total order within one thread:

**Definition 24.** Let $< \text{ denote the union of the total orders on events of each thread, i.e., } e < e' \text{ iff thread}(e) = \text{thread}(e')$ and $e <_{\tau} e'$.

This relation is extended by convention to abstract relevant events (when these are defined): if $e < e'$ then we also write $\alpha_\phi(e) < e'$ and $e < \alpha_\phi(e')$ and $\alpha_\phi(e) < \alpha_\phi(e')$. Then, with the help of the scope function discussed above, we can define the control-flow dependence on events as follows:

**Definition 25.** We write $e \prec_{\text{ctrl}} e'$ iff $e < e'$ and stmt($e'$) $\in$ scope(stmt($e$)), and $e'$ is smallest with this property, i.e., there is no $e''$ such that $e < e'' < e'$ and stmt($e'$) $\in$ scope(stmt($e''$)).

In other words, if $e$ and $e'$ are events occurring within the same thread in an execution trace $\tau$ of some multi-threaded system, we say that $e'$ has a control-flow dependence on $e$, written $e \prec_{\text{ctrl}} e'$, iff $e$ is the latest event occurring before $e'$ with the statement that generated $e'$ in the control scope of the statement that generated $e$. Obviously, the events generated in the branches of some conditional statement or in the body of a loop have the control-flow dependence on the events determining the choice of the condition statement. Consider the two example programs in Figure 8. In (A), the write on $x$ at $S_1$ and the write on $y$ at $S_2$ have a control-flow dependence on the read on $i$ at $C_1$, while the write on $z$ at $S_3$ does not have such control-flow dependence; in (B), the write on $y$ at $S_1$ control-flow depends on the read on $i$ at $C_1$. But for the write on $z$ at $S_2$, the situation is more complicated. Briefly, if the loop always terminates, events produced outside of the loop do not control-flow depend on the condition of the loop; otherwise, the loop condition control-flow precedes all the subsequent events.
The purpose of control-flow dependence, say \( e' \subseteq_{\text{ctrl}} e \), is to show that the existence of an event \( e \) is determined by the existence of all the events \( e' \). To distinguish among different occurrences of events with the same attribute values, let us add a new attribute to every event, \textit{counter}, collecting the number of previous events with the same attribute-value pairs (other than the \textit{counter}). Event \( e \) is said to \textit{occur} in a partial trace \( \beta \) iff there is an event \( e_{\text{abs}} \) in \( \alpha_{\beta}(\beta) \), such that for any attribute \textit{key}, either \( \text{key}(e) = \text{key}(e_{\text{abs}}) \) or both are undefined. Event \( e \) is said to \textit{occur regardless of attribute key} in \( \beta \) iff there is some \( e_{\text{abs}} \) in \( \alpha_{\beta}(\beta) \), such that for any attribute \textit{key} other than \textit{key}, either \( \text{key}'(e) = \text{key}'(e_{\text{abs}}) \) or both are undefined. Suppose an incomplete execution of the program that generated partial trace \( \beta \) and a relevant event \( e \) that has not occurred yet but has \textit{counter}(e)\( - 1 \) occurrences regardless of state in \( \beta \). Also, suppose that for any event \( e' \) with \( e' \subseteq_{\text{ctrl}} e, e' \) has already occurred in \( \beta \). Then we claim that \( e \) will occur regardless of its \textit{state} and \textit{counter} when the execution continues, independently of thread scheduling choices. The detailed formalization of these intuitions seems technically intricate and probably not worth the effort.

**Data-flow Dependence.** If a change of \textit{state}(e) may affect the \textit{state}(e') then we say \( e' \) has a \textit{data-flow dependence} on \( e \) and write \( e \sqsubseteq_{\text{data}} e' \).

**Definition 26.** For two events \( e \) and \( e' \), \( e \sqsubseteq_{\text{data}} e' \) iff \( e <_r e' \) and one of the following situations happens:

1. \( e < e', \ \text{type}(e) = \text{read and stat}(e') \) uses \textit{target}(e) to compute \textit{state}(e');
2. \( \text{type}(e) = \text{write, type}(e') = \text{read, target}(e) = \text{target}(e'), \) and there is no other \( e'' \) with \( e <_r e'' <_r e' \), \( \text{type}(e'') = \text{write}, \) and \( \text{target}(e'') = \text{target}(e') \);
3. \( e < e', \ \text{type}(e') = \text{read, stat}(e') \notin \text{scope}(\text{stat}(e)), \) and exists a statement \( S \) in scope (\text{stat}(e)) s.t. \( S \) can change the value of \textit{target}(e')

One can see in the definition that, in most cases, the data-flow dependence is straightforward: for an assignment statement, the write on the left hand side has the data-flow dependence on the reads on the right hand side; and a read data-flow depends on the most recent write on the same memory location. For example, in Figure 8 (A), if an execution is \( C_1S_1S_3 \), then the read on \( x \) at \( S_3 \) has data-flow dependence on the write on \( x \) at \( S_1 \). However, some cases are a little more intricate. Assuming another execution of Figure 8 (A), say \( C_1S_2S_3 \), one will not see a direct data-flow dependence. If the value of \( i \) changes then \( S_1 \) could be executed instead of \( S_2 \), so the value of the write at \( S_3 \) would be different. Therefore, there is a data-flow dependence from the write at \( S_3 \) to the read at \( C_1 \). Similarly, in Figure 8 (B), the write on \( z \) at \( S_2 \) data-flow depends on the read at \( C_1 \). Therefore, we say that an event \( e \) data-flow depends on \( e' \), if \( e' \) is an affecting event at a choice statement \( s \) and the value of \( e \) can be changed by some statement in the control scope of \( s \). By affecting, we mean that the value of the event may change the choice of the statement. To correctly determine such data-flow dependence, aliasing information among variables is required, which one can achieve using any available techniques.

Note that there are no write-write, read-read, read-write data dependence. Case (2) above only considers the write-read data dependence, enforcing the read to depend upon only the latest write of the same variable. This way, a write and the following reads of the same shared variable form an \textit{atomic} block of events. This captures entirely the work presented in [29], in the much more general setting of this paper.

Similarly to the control-flow dependence, the data-flow dependence also extends by convention to abstract relevant events (when defined) as expected: if \( e \sqsubseteq_{\text{data}} e' \) then \( e \sqsubseteq_{\text{data}} \alpha_{\phi}(e'), \) \( \alpha_{\phi}(e) \sqsubseteq_{\text{data}} e' \), and \( \alpha_{\phi}(e) \sqsubseteq_{\text{data}} \alpha_{\phi}(e') \). One can now show that an event \( e \) is uniquely determined by all the events \( e' \) with \( e' \sqsubseteq_{\text{data}} e \). Suppose an incomplete execution of the program that generated partial trace \( \beta \) and a relevant event \( e \) that has not occurred yet but which will occur regardless of its \textit{state} attribute, which also has the property for any event \( e' \) with \( e' \sqsubseteq_{\text{data}} e, \) \( e' \) has already occurred in \( \beta \). Then \( e \) (including the value of its \textit{state}) will also occur when this execution continues, independently of thread scheduling. Note that if the abstract event \( e \) does not contain a state attribute, then the data-flow dependence is not taken into account.

**Hybrid Dependence.** Now we can define the notion of hybrid dependence on events by merging the control-flow and the data-flow dependences:

**Definition 3** Event \( e \) \textit{depends upon} \( e' \) if and only if \( e' \subseteq e \), where \( \subseteq \) is the relation \((\sqsubseteq_{\text{data}} \cup \sqsubseteq_{\text{ctrl}})^+\).

As indicated by the discussion above, to compute this dependence relation on events some static structural information about the program is required. The most important piece of information that we collect statically is the \textit{control...
scope of every conditional statement, which is formally discussed in [7]. Besides, termination information of loops and aliasing relationship among variables are also needed. Termination and aliasing analyses are difficult problems by themselves and out of the scope of this paper. We are trying to make use of off-the-shelf analysis tools in our approach to accumulate static information about the program, which is further conservatively used by the subsequent dynamic analysis components, guaranteeing the soundness of our approach. Some heuristic assumptions may be adopted in implementations of the technique to improve performance, but these may introduce false alarms (see Section 6).

Thanks to the dynamic/static combined flavor of our approach, we only need to carry out intra-procedural static analysis. Method invocations will be expanded at runtime and the dependence relation can be propagated along method invocations easily: if a method call control-flow depends on an event, then all events produced by the method call control-flow depend on. Moreover, since our actual analysis takes place at runtime, we keep track of actual memory locations appearing in events, so the inter-procedural data-flow dependence can be computed similarly as the intra-procedural one using memory locations instead of variable names.

It is worth noting that the above discussion about dependence is independent from the particular definition of an event. The hybrid dependence can be computed on either the concrete events generated by the execution, or on the abstract relevant events discussed in the previous section. The latter usually results in more relaxed (less constrained) dependence relationships. For example, if some abstract event does not contain/need information about the state of an event (e.g., for data-race analysis we only care that there is a write of at in Figure 8 (A), but not the concrete value written to z), then only the control-flow dependence is considered and the data-flow dependence can be ignored.

4.3 Sound Permutations and Sliced Causality

One can show that any linearization of events that is consistent with, or preserves, the hybrid dependence partial-order guarantees the occurrence of relevant events and also preserves their state. Our goal is to generate and analyze permutations of relevant events that correspond to possible executions of the system.

Definition 4 A permutation of $\xi$ is sound if there is some execution whose trace can be abstracted to this permutation.

The most appealing aspect of predictive runtime analysis is that one does not need to re-execute the program to generate sound traces; instead, we define an appropriate notion of causal partial order and then prove that any permutation consistent with it is sound. Intuitively, a sound permutation preserves relevant events as well as events upon which relevant events depend.

Definition 5 Let $\overline{\xi}$ be the set extending $\xi$ with events $e \in \xi$ such that $e \preceq e'$ for some $e' \in \xi$. We then let $\preceq \subseteq \overline{\xi} \times \overline{\xi}$ be the sliced causal partial order relation, or the sliced causality, defined as $(< \cup \subseteq)^*$. If $\xi$ is a partial order relation, or the sliced causality, defined as $(< \cup \subseteq)^*$.

Unless otherwise specified, from now on by “causal partial order” we mean the sliced one. Therefore, the sliced causality is nothing but the dependence relation extended with the total order on the events generated by each thread; or, in other words, it can be regarded as the slice of the traditional causal partial order based on the dependence relation extracted statically. The causal partial order was defined on more events than those in $\xi$, but in order to generate sound permutations of relevant events we only need its projection onto the relevant events:

Theorem 1. A permutation of $\xi$ is a sound abstract trace whenever it is consistent with the sliced causality $\prec$.

Proof: Let $e_1 e_2 \cdots$ be a permutation of the events in $\xi$ that is consistent with $\prec$, or in other words a linearization of $\prec$, and let $\Sigma_i = \{e_1, \ldots, e_i\}$ denote the set of the first $i$ events of this abstract trace. Then one can easily show by induction on $i$ that if $e < e_i$ for some event $e$, then $e \in \Sigma_i$. Such sets $\Sigma_i$ are also called consistent cuts and will be further discussed in Section 5.2. Then we can construct an abstract execution of the program for this permutation by induction (same steps are followed to generate a counter-example when the property is violated):

1. For $e_1$, we simply start the thread $\text{thread}(e_1)$ and pause it after $e_1$ is generated;
2. For $e_i$, by the induction hypothesis we have constructed an execution of the program which produces $e_{i-1}$. Since all the events upon which $e_i$ depends are already preserved in the execution, we can safely start the thread $\text{thread}(e_i)$ to produce $e_i$ and pause it.

We can therefore simulate an execution of the system that generates the original permutation of relevant events as an abstract trace.
5 Generating Sound Permutations

We next describe the generation of sound event permutations, that is, ones that are consistent with the sliced causality relation discussed above. First, a vector clock (VC) based algorithm that encodes the sliced causality is introduced. Then we show that the causal order can be further relaxed by considering the particular atomicity semantics of synchronization objects (locks), rather than a generic read/write semantics. Finally, an algorithm is given which generates all the sound permutations of relevant events in parallel, following a level-by-level (in terms of the associated computation lattice) or breadth-first strategy.

5.1 Computing Causal Partial Order

A VC-based algorithm was presented in [28] to encode a “happen-before” causal partial ordering on events that was extracted entirely dynamically, ignoring any static information about the program that generated the execution trace. We next non-trivially extend that algorithm to consider static information, transforming it into a VC algorithm encoding the slicing causality relation.

Definition 6 A vector clock (VC) is a function from threads to integers, VC : T → Int, where T is the set of threads. VC ≤ VC’ iff ∀t ∈ T, VC(t) ≤ VC’(t). And we have the max function for VCs: max(VC1,...,VCn)(t) = max(VC1(t),...,VCn(t)).

Every thread t has a VCt, which keeps the order within the thread as well as the information about other threads that it knows from their communication (read/write events on shared variables). Every variable x has a VCx that shows how the value of the variable is computed. Every shared variable x has a VC′x that accumulates the information about variable accesses. When a concrete event e is encountered, it will be abstracted using the filter function and then associated with a VCt, which encodes the causal partial order. We next show how to update these VCs when an event e is encountered during the analysis (the third case can overlap the first two cases):

1. type(e) = write, target(e) = x, thread(e) = t (the variable x is written in thread t). In this case, the VCx is updated using VCx′, VCt, and VCs of those events upon which e depends: VCx = max(VCx′, VCt, VC1,...,VCn) where e1,...,en ⊆ e. Then VCx = VCx′ = VCx.
2. type(e) = read, target(e) = x, thread(e) = t (the variable x is read in t), and x is a shared variable. The information of the thread is accumulated into VC′x: VC′x = max(VC′x, VCt), and VCx = VC′x.
3. e is a relevant event w.r.t. the desired property, thread(e) = t. For this case, VCt needs to be increased in order to keep the total order within the thread, and the corresponding relevant event will be issued to the observer with an up-to-date VC.

However, it is not straightforward to determine the relevant event if one tries to calculate the vector clocks online, when only the information up to some execution point is available (no look-up into the future). Figure 14 (A) and (B) illustrate two cases that require backtracing in the calculation, in which e, e′, e′′ ∈ ξϕ and e1, e2 ∈ ξϕ. Basically, this is caused by some “delayed” dependence among events. For the case in (A), when e′ is processed, e1 seems unimportant to verify ϕ and is not taken into account by the algorithm. But when e″ is encountered, e1 becomes an important event and the algorithm has to re-compute VCϕ. (B) is similar but a little more complex: e1 and e2 are considered unimportant until e″ is hit. To recognize such cases, we can notice that, if e1 is not taken into account, the thread’s vector clock is not updated using VCe1. Therefore, we have VCe1 ̸∈ VCϕ. And for the same reason, if e′ has been processed and e1 < e′, VCe1 ̸∈ VCϕ. This way, we are able to go back and refine the VCs of previous events.

Because of backtracing, in the worst case, the complexity of the online algorithm is square in the number of events. For offline analysis (carried out after the execution finishes), on the other hand, one can first scan the execution trace backwards to figure out all important events and then compute VCs from the beginning of the trace, reducing the worst case complexity to linear. The online version of the algorithm is adopted in our prototype Predictor, although it presently works in the offline mode, because the experiments show that in practice backtracing appear very infrequently (Section 7).

We can show that the vector clocks encode the sliced causality ⊑:

Theorem 2. e < e′ ⇒ VCe ≤ VCe′
The proof of the important theorem above can be (non-trivially) derived from the one in [28]. The extension here is that the dependence is taken into account when computing the VCs of variables and relevant events. Note that in our case the partial order $\leq$ among VCs is stronger than the sliced causality $\prec$ among events. This is because when VCs are computed, the read-after-write order is also taken into account (the second case above), which the $\prec$ order does not need to encode. Theorem 1 yields the following immediately:

**Proposition 14.** Any permutation on events that is consistent with $\leq$ among events’ VCs is sound w.r.t. the sliced causality $\prec$.

### 5.2 Lock-Atomicity of Events

One may further loosen the causal partial order if more semantic information is obtained for the program. We next discuss how to incorporate the lock mechanism into our approach to construct more sound traces. Locks play a significant role in multi-threaded programs. In most causal order based approaches, locks are treated as shared variables, and acquiring and releasing locks are viewed as reads and writes of the associated lock objects. This way, blocks protected by the same lock are naturally ordered and kept exclusive to one another. However, this ordering is stronger than the actual lock semantics, which only imposes the mutual exclusion among blocks protected by the same lock. To better support lock semantics, we next extend our sliced causality with **lock related atomicity**. Using this concept, two sets of events that are atomic w.r.t. the same lock cannot be interleaved, but can be permuted if there are no other causal constraints on them.

Two new types of events are introduced for lock operations, acquire and release. The target of these events is the lock to be accessed. If there are embedded lock operations on the same lock (a thread can acquire the same lock multiple times), only the outmost acquire-release pair generates events. For example, the thread $t_1$ in Figure 1 may produce the event trace in Figure 15. The control-flow dependence is extended correspondingly:

**Definition 7** $e', e'' \in \xi$, type$(e') = acquire$ and type$(e'') = release$ of the same lock $l$. Then $e' \sqsubseteq_{ctrl} e$ for all $e' < e < e''$.

That is to say, an event $e$ protected by an acquiring of $l$ has the control-flow dependence on the acquiring event. For example, in Figure 15, $e_{13} \sqsubseteq_{ctrl} e_{14}$ since $e_{13} < e_{14} < e_{16}$; and $e_{13} \sqsubseteq_{ctrl} e_{15}$. Two events protected by the same lock are atomic w.r.t. the lock:

**Definition 8** Two events $e$ and $e'$ are $l$-atomic, written $e \mathbin{\#}_l e'$, iff $\exists e'' \in \xi, \text{type}(e'') = acquire, \text{target}(e'') = l, e'' \sqsubseteq_{ctrl} e$ and $e''' \sqsubseteq_{ctrl} e'$. $\mathbin{\#}_l$ is an equivalence relation on $\xi$. Let $[e]$ denote the corresponding equivalence class of an event $e \in \xi$.

For example, in Figure 15, $e_{13} \mathbin{\#}_l e_{15}$, meaning that they are atomic w.r.t. lock. To capture the lock-atomicity among events, we associate a counter $\text{counter}_l$ with every lock $l$. Let $LS_t$ denote the set of locks held by the thread $t$. 

![Fig. 14. Backtracing cases for VC Generation (solid nodes are relevant events and blank nodes are irrelevant ones.)](image)

![Fig. 15. Event trace containing lock operations](image)
A new attribute, $LS$, is also added into the event, whose value is a mapping on locks to corresponding counters. When an event $e$ is processed, the lock information is updated as follows:

1. if type($e$) = acquire, thread($e$) = $t$, target($e$) = $l$, then counter$_{t} = \text{counter}_{t} + 1$, $LS_{l} = LS_{l} \cup \{l\}$.
2. if type($e$) = release, thread($e$) = $t$, target($e$) = $l$, then $LS_{l} = LS_{l} - \{l\}$.
3. if $a(e)$ defined, then let $LS(e)(l) = \text{count}_{t}$ for any $l$ in $LS_{\text{thread}}$, and $LS(e)(l) = -1$ for any other $l$.

The following theorem states the correctness of this algorithm:

**Theorem 3.** $e \notin \emptyset$, $e'$ iff $LS(e)(l) = LS(e')(l) \neq -1$

**Proof:** Similar to the proof for Theorem 1, the definition of the consistent run actually gives the way to construct an execution of the program which can be represented by the permutation. \hfill \Box

### 5.3 Consistent Runs and Cuts

Every sound permutation can be viewed as an abstract run of the program. A run is called *consistent* from now on if it preserves not only the sliced causality, but also the lock-atomicity relation above. Let us first define the concept of *consistent cuts*:

**Definition 9** A cut $\Sigma$ is a set of events. $\Sigma$ is *consistent* if and only if for all $e, e' \in \Sigma$,

1. if $e \in \Sigma$ and $e' < e$, then $e' \in \Sigma$ and
2. if $e \notin \{e\}_l$ for some lock $l$, then either $[e]_l \subseteq \Sigma$ or $[e']_l \subseteq \Sigma$.

The first property says that for any event in $\Sigma$, all the events upon which it depends should also be in $\Sigma$. The second property states that there is *at most one* incomplete $l$-atomic set in $\Sigma$. Otherwise, the $l$-atomicity is broken. Essentially, $\Sigma$ contains the events in the prefix of a consistent run. When an event $e$ can be added to $\Sigma$ without breaking the consistency, $e$ is called *enabled* for $\Sigma$.

**Definition 10** An event $e$ is *enabled* for a consistent cut $\Sigma$ iff

1. for any event $e' \in \xi$, if $e' < e$, then $e' \in \Sigma$, and
2. for any $e' \in \Sigma$ and any lock $l$, either $e \in [e']_l$ or $[e']_l \subseteq \Sigma$.

This definition is equivalent to the following one:

**Definition 11** $e$ is *enabled* for a consistent cut $\Sigma$ if and only if $\Sigma \cup \{e\}$ is also consistent.

Now we can define a consistent run:

**Definition 12** A consistent multi-threaded run $e_1e_2...e_{|\xi|}$ is one which generates a sequence of consistent cuts $\Sigma_0|\ldots|\Sigma_{|\xi|}$: for all $1 \leq r \leq |\xi|$, $\Sigma_{r-1}$ is a consistent cut, $e_r$ is enabled for $\Sigma_{r-1}$, and $\Sigma_r = \Sigma_{r-1} \cup \{e_r\}$.

We can have the following theorem for the correctness of this algorithm. The proof is straightforward, so we ignore it here.

**Theorem 4.** Any consistent run of $\xi$ is sound.

### 5.4 Generation of Consistent Runs

Figure 16 gives an algorithm to generate and verify, on a level-by-level basis, consistent runs based on the causal partial order and the lock-atomicity. In this algorithm, $\xi$ is the set of relevant events, while $\text{CurrentLevel}$ and $\text{NextLevel}$ are sets of cuts. We do not store all the events for the cut $\Sigma$ in the algorithm; instead, $\Sigma$ is encoded using the following information: the VCs of threads and shared variables, lock sets held by threads, and the current state of the property monitor for this run. The property monitor is a program which verifies the run against the desired property. In our approach, the monitor is automatically generated from the specification of the property (see Section 6).

The algorithm first checks every event in $\xi$, and every cut in the current level to generate cuts of the next level by appending enabled events to the current cuts. After the next level is generated, redundant events, which are already processed in all runs, will be removed from $\xi$. The *enabled* procedure implements the definition of the consistent run: it first compares the VCs of the event with the current cut’s and then checks their lock-atomicity. If an event $e$ is enabled for a cut $\Sigma$, it will be added to $\Sigma$ to create a new cut $\Sigma'$. But first, it is sent to the monitor along with $\Sigma$ to verify against the desired property. Violations are reported as soon as detected. Otherwise, the vector clocks and lock set information of $\Sigma'$ will be computed and $\Sigma'$ is returned.
6 jPredictor

To evaluate the effectiveness of the proposed technique, we implemented a prototype predictive runtime analysis tool for multi-threaded Java programs, called jPredictor. Despite its yet unfriendly user interface and room for improving its performance, jPredictor was able to detect several concurrency bugs, some of them unknown yet, in non-trivial applications (Section 7), including Tomcat 5 [31]. We are continuously improving this prototype and applying it on new examples, and intend to transform it into a real, easy-to-use tool; we refer the reader to jPredictor’s website [17] for more information or to download its latest version. It is fair to mention here that, despite the theoretical soundness of our sliced causality technique, its implementation in jPredictor is not sound anymore, i.e., false alarms may be reported. Our decision to break its theoretical soundness was due to purely pragmatic reasons, explained in the sequel. However, in our experiments with jPredictor all the violations it reported were real violations.

Figure 17 shows the architecture of jPredictor. The system contains three major components: a static analyzer, a trace analyzer and a monitor synthesizer. The static analyzer instruments the program to issue events when executed; it also extracts static structural information from the program to be used later by the trace analyzer. The monitor synthesizer generates monitors from requirements specifications, which will be further used by the trace analyzer to verify the various permutations of relevant events against desired properties. For efficiency and modularity reasons, we distinguish two kinds of monitors in jPredictor: (1) specialized monitors that check well-defined, particular but important properties, such as data races for different variables; and (2) general purpose property monitors, automatically generated from formal specifications using the publicly available logic-plugins of the JavaMOP system [6]. By analyzing statically the property specification, the monitor synthesizer also provides the definition of relevant events. We do not discuss the monitor synthesizer here.

Once the program is instrumented and the monitors are generated, the user of jPredictor needs to run the instrumented program to gather execution traces, which are fed into the trace analyzer for the actual predictive analysis. The trace analyzer extracts the relevant events from the concrete trace(s), computes their VCs, and then constructs consistent runs by permuting relevant events, at the same time checking them against the corresponding monitors.

6.1 Static Analyzer

The static analyzer takes the original program as input and produces an instrumented program as output, together with static information needed for the trace analyzer. Figure 18 shows the three main components of the static analyzer: a program instrumentor, a control flow analyzer and an alias analyzer. All the outputs are stored in ASCII text files.
The program instrumentor is the core component of the static analyzer. It works at the byte code level. We are currently using the jTrek [8] package. The original program is instrumented with bytecode instructions that issue events at runtime, such as reads/writes on memory locations and begins/ends of function calls. The generated events are first placed in a global synchronized buffer and then flushed into a log file.

The soundness of our sliced causality technique is based on the assumption that all the code is instrumented. However, in practice, complete code instrumentation can cause an unacceptable runtime overhead; moreover, sometimes it is even impossible to achieve it, e.g., due to native methods. To keep its analysis practical and effective, jPredictor allows its users to specify which parts of the program to instrument. This way, the user can control the granularity and performance of the analysis by choosing different sets of classes to instrument according to the property of interest. There may be therefore uninstrumented methods invoked from the instrumented program. To avoid losing dependencies on variable updates, the un-tracked methods can be annotated with purity information: pure methods do not change the receiver object and will be regarded as reads on the object, while non-pure uninstrumented methods are regarded as writes on the receiver. Also, arguments that can be changed by the method can be annotated as out arguments. These method annotations can be reused and may be obtained by static analysis on the source code (if available), or even contained in interface specifications of classes (e.g., using JML [19]).

As mentioned in the previous section, the termination information of loops/recursion may also be taken into account to relax the control dependence relation. jPredictor allows the user to introduce annotations regarding termination in the code; one can produce these annotations either manually, or otherwise automatically by using some off-the-shelf static analysis tool. To relieve the user from producing termination annotations, a heuristic assumption for loops is implemented in jPredictor: when the condition of the loop involves no shared variables, the loop is assumed to terminate. This assumption brings unsoundness into the tool, but turned out to be so effective in our experiments (we did not need to further annotate any loops) that we decided to allow it anyway.

The control flow analyzer computes control scopes of statements using a simple algorithm discussed in [7]. The trace analyzer uses these control scopes to determine a refined control-flow dependence on events, as briefly explained in Section 4.2 and elaborated in depth in [7]. The alias analyzer implements a naive intra-procedural conservative alias analysis in our current implementation of jPredictor, which we are going to replace with some more powerful existing tool. By conservative we here mean that all variables not known to be unaliased are assumed aliased. This way, the lack of precision of the alias analyzer only affects the predictive power of our tool, not its soundness. The soundness of jPredictor is only affected by our heuristic regarding the termination of loops, which was not a source of false alarms in our experiments.

---

2 jTrek is not maintained anymore, so we are considering replacing it with the BCEL [2] library.
6.2 Trace Analyzer

The trace analyzer implements our runtime analysis approach based on sliced causality, and therefore has the capability of predicting potential bugs from concrete executions of the program that may not hit the bug. Its input includes the execution trace generated by the instrumented program together with the static information produced by the static analyzer, along with the monitor to check the desired property. Currently, for simplicity, predictor works in the offline mode, which means that the analyzer is not invoked at runtime, but after the execution, analyzing the generated trace log; however, the main VC generation algorithm is designed to work in the online mode, too. For efficiency reasons, the analysis process is divided into three phases, as depicted in Figure 19 (these will need to be changed in online mode).

Fig. 19. Architecture of trace analyzer

In the first phase, the pre-processor goes through the input execution trace and collects information about the usage of objects. Specifically, life cycle information about objects is collected, based on which the VC generator can minimize the usage of memory by discarding information about objects when they are dead. Besides, if the tool is requested to detect dataraces, the pre-processor will also detect the shared variables. The complexity of the pre-processor is linear in the length of the trace.

The VC generator extracts the relevant events from the execution trace and computes the corresponding VCs and the lock-atomicity using the algorithms in Section 5.1 and Section 5.2. Since the trace contains detailed runtime information, the analysis can be very fine-grained, e.g., every element in an array can be processed individually if desired. However, in many cases such a fine-grained analysis is not necessary. Because of the back-tracing step in the algorithm, the complexity of the VC generation algorithm is, in the worst case, square in the length of the trace. In the offline mode, one can perform a backwards analysis of the trace first to compute the dependence, and then use another forward pass to compute the VCs. This way, the algorithm would be linear in the length of the trace, but it would only work in the offline mode. However, our experiments show that backtracing is needed very rarely: the forwards VC generation algorithm behaved linearly in the length of the trace in all tested cases.

The computed relevant event set, along with the implicit partial-order relationship encoded by vector clocks and lock-atomic sets, is then passed to the trace checker to verify it against the desired property. The trace checker generates all the consistent runs in parallel, on a level-by-level basis using the algorithm in Figure 16, invoking at the same time the property monitor to verify these runs. In the worst case when there is no partial-ordering among the relevant events (corresponding to no thread-interaction), case in which our technique explores the same state-space as a model-checker, the complexity of our trace analyzer would be exponential in the number of relevant events. Yet, as shown in the next section, since the number of relevant events is usually small the complexity of the trace analyzer is quite reasonable. Dually to model-checking where the goal is to reduce the state-space, in predictive runtime analysis we want to investigate as many potential runs as possible that are consistent with the observed execution. However, if the number of such runs is too large, one can select only those runs that are most likely to appear using the idea of causality cone, as we did in [29] for the purely dynamic happen-before relation considered there. Moreover, for some
simple properties one does not even need to generate all the runs. For example, for data-race detection on a shared variable \( x \), all one needs to check is that there are two causally unrelated access events on \( x \), at least one of them is a write, and the two events have disjoint lock-sets; in this case, all what \( j\text{Predictor} \) needs to do is to compare the VCs and the lock-sets of events instead of generating the expensive permutations.

7 Evaluation and Discussion

\( j\text{Predictor} \) has been applied on several applications to evaluate its capability. Two kinds of properties are verified in the evaluated programs. We first try to detect potential dataraces in the programs, and then verify some safety properties expressed using temporal formalisms. During our evaluation, two bugs were revealed in Tomcat 5, one of which has been reported by other users of Tomcat and the other one seems unknown up to our knowledge. What follows gives the results of the evaluation, and we also discuss the application of \( j\text{Predictor} \) in practice. Note that presently we focus on strengthening the prediction ability while preserving the soundness of the approach. So we leave the comparison with other unsound dynamic analysis techniques, e.g., lock-set based approaches, to our future work, although our tool indeed detected all known bugs in most tested programs. All these experiments were done on a 2.0GHz Pentium4 machine with a 1 GB memory.

7.1 Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Slowdown</th>
<th>S. V.</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banking</td>
<td>150</td>
<td>x3</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Http-Server</td>
<td>170</td>
<td>x3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Daisy</td>
<td>1.3K</td>
<td>x10</td>
<td>312</td>
<td>3</td>
</tr>
<tr>
<td>Daisy-2</td>
<td>1.5K</td>
<td>x20</td>
<td>572</td>
<td>3</td>
</tr>
<tr>
<td>Raytracer</td>
<td>1.8k</td>
<td>x2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(Part of) Tomcat 5</td>
<td>10K</td>
<td>x10</td>
<td>20</td>
<td>3-4</td>
</tr>
</tbody>
</table>

Table 1. Benchmarks used in evaluation

Table 1 shows the benchmarks that we use, along with their size (lines of code). Slowdown column is the slowdown ratios of the program after instrumentation, S.V. column the number of shared variables found in the execution, and Threads column the number of threads created in our tests.

Banking and Http-Server are two simple examples from [32], showing some concurrent bug patterns discussed in [12]. Daisy [23] is a small file system that was developed as a concrete system for application of software verification tools. It is highly concurrent with fine-grained locking. Specifically, it uses a RandomAccessFile object to simulate the hard disk, and user-defined spin-wait locks to protect every logic block and every directory in the system. Since RandomAccessFile is a native class in Java, \( j\text{Predictor} \) cannot instrument it. This results in imprecise warnings: it only points out that there are dataraces on the disk variable, which is an object of the RandomAccessFile, but does not give more specific reasons.

To better evaluate the tool, we implemented a revised version of Daisy, named Daisy-2, which replaces RandomAccessFile by PseudoFile class that is based on a byte array. For this version, the tool successfully reports fine-grained race conditions. Both Daisy and Daisy-2 involve a large number of shared variables because every block of the disk holds a shared variable as the mutex lock. This imposes a heavy load on the analysis tool. Daisy-2 has more shared variables because of the shared byte array used to simulate the disk.

Raytracer is a program from the Java Grande benchmark suite [16], which implements a multi-threaded ray tracing algorithm. Tomcat [31] is a popular open source Java application server. The version used in the experiment is 5.0.28, the latest version of Tomcat 5.0.x. Instead of instrumenting and running the whole system, only some test programs are tried to specifically test a few components of Tomcat, e.g. the class loaders and logging handlers, because of the limitation of time and also because of the consideration of performance.

From Table 1, we can see that the runtime overhead caused by the instrumentation is acceptable for most applications. Moreover, most bugs are found in a single run of the programs.
### 7.2 Detecting Dataraces

We applied our approach first on datarace detection since the property is well defined and also crucial for multi-threaded programs. Note that the tool needs to repeat VC generation and property checking for every shared variable. The time cost shown in Table 2 is the analysis time for just one shared variable. $T_{pre}$ is the time for pre-processing, $T_{vc}$ is the time of VC generation, and $T_\phi$ is the time used to detect the dataraces. It is worth noting that, according to these experiments, even though the possibly worst cost of the VC generation is $O(|\xi|^2)$, it usually takes time linear to the length of the trace. This shows that the backtracing case in the algorithm is rare in practice. The performance of property verification is also reasonable, which may be credited to the specific algorithm for detecting dataraces. Sometimes it takes less time to detect the races than to generate VCs because the tool returns right after it finds the bug.

We can see that our tool does not produce false alarms in these experiments, and found almost all the previously known dataraces in most program except Tomcat. For Tomcat, four potential dataraces are found but further analysis shows that two of them are benign in the sense that do not cause real errors in the system. The other two seem to be real bugs in our understanding. As a matter of fact, they have been submitted to the bug database of Tomcat by some users of Tomcat. Both bugs are hard to reproduce and only rarely occur under very heavy workloads, but our tool was able to catch them using a few working threads. Moreover, one bug was claimed to be fixed but when we try the patched version, the bug still exists in our point of view. Let us take a more detailed view on these two bugs next.

The first one is found in `startCapture` method of the `org.apache.tomcat.util.log.SystemLogHandler` class. The buggy code is shown in Figure 20. `reuse` is a static member of the class, shared by different threads. There is an obvious datarace between `reuse.isEmpty()` and `reuse.pop()`, and it would cause an `EmptyStackException`. The difficulty of detecting this bug resides in the complicated interaction among threads in the implementation of Tomcat. We do not apply the lock-set algorithm and the traditional happen-before technique in the evaluation for comparison. But our inspection shows that there are many unprotected shared variables used in the Tomcat system without causing dataraces. The lock-set algorithm would very likely produce a lot of false alarms in such case and overwhelm the true bug. On the other hand, the traditional happen-before technique needs good luck, especially in such complex system, to obtain some execution, in which there are no other unrelated inter-thread interactions between those conflicting memory accesses; otherwise, the race will not be revealed. But more experiments are needed to really compare our tool with other datarace detection tools, e.g. [21].

The other bug is more subtle, residing in `findClassInternal` of `org.apache.catalina.loader.WebappClassLoader`. Due to the limited space, we leave out the detailed explanation and only gives the simplified description here. This bug was originally reported as dataraces on shared variables `entry.binaryContent` and `entry.loadedClass` at the first conditional statement in Figure 21. The race on `entry.loadedClass` does not lead to any really errors. And the one on `entry.binaryContent` does no harm by itself, but would cause some arguments of a later call on function `definePackage(packageName, entry.manifest, entry.codeBase)` to be null, which is illegal for calling the function. It is worth noting that we actually locate this error by verifying a safety property about using the method (Section 7.3) when we tried to evaluate the impact of detected races. The error scene is not straightforward and would take quite some time to

![public static void startCapture() {...](image)

Table 2. Race detection results.

<table>
<thead>
<tr>
<th>Program</th>
<th>$T_{pre}$</th>
<th>$T_{vc}$</th>
<th>$T_\phi$</th>
<th>Races</th>
<th>Bugs</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banking</td>
<td>1s</td>
<td>2s</td>
<td>5s</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Http-Server</td>
<td>0.2s</td>
<td>0.3s</td>
<td>0.3s</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Daisy</td>
<td>5s</td>
<td>30s</td>
<td>30s</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Daisy-2</td>
<td>29s</td>
<td>30s</td>
<td>30s</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Raytracer</td>
<td>1s</td>
<td>2s</td>
<td>2s</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Tomcat</td>
<td>10s</td>
<td>20s</td>
<td>10s</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 20. Buggy code in SystemLogHandler

---

3 There is another `definePackage` function with eight arguments that allows null arguments.
infer it directly from the datarace. It seems that the Tomcat developer tried to fix this bug by put a lock around the conditional statement, as shown in Figure 22. However, our tool shows that the violation still exists in the patched code, which is now a part of the latest version of Tomcat 5, indicating that the problem has not been really solved. We have submitted this observation to the Tomcat bug database for further confirmation.

7.3 Verifying Safety Properties

Although the datarace has been the focus in debugging concurrent programs for a long time, it has been shown that the datarace itself may not be the bug that people really care about. In fact, many dataraces do not cause errors in real system, such as our observation on Tomcat, while some concurrent errors occur even when there are no dataraces, e.g. the atomicity [14]. Therefore, it is better for one to directly detect the violation of desired properties than to guess possible errors caused by dataraces. The strength of our approach allows JPredictor to verify another kind of important properties, namely safety properties that are usually expressed in the term of the event trace. For example, for Http-Server, one can detect a datarace on the client object, but the essential error is the violation of some interface specification of Thread class, stating that the calls to suspend and resume should alternate. It can be written as a regular pattern: (suspend resume)* (please see [6] for more details about this example).

<table>
<thead>
<tr>
<th>Program</th>
<th>$T_{pr}$</th>
<th>$T_{pt}$</th>
<th>$T_{tc}$</th>
<th>Properties</th>
<th>Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Http-Server</td>
<td>0.2s</td>
<td>0.3s</td>
<td>0.3s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Daisy</td>
<td>3s</td>
<td>20s</td>
<td>10s</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Tomcat</td>
<td>10s</td>
<td>20s</td>
<td>13s</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. Safety property checking results.

We tried a few temporal properties on some evaluated programs and the result is shown in Table 3. Those properties are written as Java MOP specification, from which monitors are automatically generated. For example, Figure 23 gives the specification that we checked for Http-Server, and Figure 24 shows the corresponding generated monitoring code.

The property checked for Daisy is about the atomicity of dumpDisk method: there are no writes on the disk during the disk dumping. The detected violation is not really an error w.r.t. the implementation of Daisy, but similar properties can be valuable in other file systems. For Tomcat, as we discussed above, the property is devised to verify our suspicion
on the detected dataraces. It is based on the semantics of some functions, stating that all the arguments to the function cannot be set to null until the function has been called. Our tool quickly reports a violation on calling the `definePackage` function in both the un-patched and patched version of WebappClassLoader.

8 Conclusion and Future Work

This paper presents a runtime analysis approach for predicting potential concurrency errors in multi-threaded programs. The presented approach is based on the concept of *sliced causality*, a dependence-based causal partial order on property-relevant events. By relaxing the strict “happen-before” causal partial order and abstracting events according to the property of interest, our approach provides a powerful predictive capability. A runtime prediction tool for Java, called *jPredictor* has been implemented and evaluated. The experiments illustrate that *jPredictor* is able to detect concurrent bugs at a reasonable cost. A possibly unknown bug of Tomcat 5 was found during our evaluation.

There are several aspects of the presented technique that may lead to interesting future developments. First, the current VC generation algorithm is stronger than necessary: the write-after-read dependence can be avoided. It is also possible to improve the performance of implementations by making more intensive use static analysis techniques. For example, one can use static program slicing to avoid unnecessary instrumentation. This way, the performance of both the instrumented program and that of the analysis can be improved. More development efforts and evaluation work are needed to make *jPredictor* a practical tool.

References


