Predicting Null-Pointer Dereferences in Concurrent Programs [*]

Azadeh Farzan [†] azadeh@cs.toronto.edu

P. Madhusudan [‡] madhu@illinois.edu

Niloofar Razavi [†] razavi@cs.toronto.edu

Francesco Sorrentino [‡] sorrent1@illinois.edu

Abstract

We propose null-pointer dereferences as a target for finding bugs in concurrent programs using testing. A null-pointer dereference prediction engine observes an execution of a concurrent program under test and predicts alternate interleavings that are likely to cause null-pointer dereferences. Though accurate scalable prediction is intractable, we provide a carefully chosen novel set of techniques to achieve reasonably accurate and scalable prediction. We use an abstraction to the shared-communication level, take advantage of a static lock-set based pruning, and finally, employ precise and relaxed constraint solving techniques that use an SMT solver to predict schedules. We realize our techniques in a tool, ExceptioNULL, and evaluate it over 13 benchmark programs and find scores of null-pointer dereferences by using only a single test run as the prediction seed for each benchmark.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Testing, Concurrency, SMT, Null-pointers, Data-Races

1. Introduction

Errors in concurrent programs often occur under subtle interleaving patterns that the programmer had not foreseen. There are too many interleavings to explore, even on a single test input for a concurrent program, making concurrency testing a hard problem. With the rise of multicore hardware platforms, finding solutions to this problem is very important as testing is still the most effective way of finding bugs today. Current testing technologies such as stress testing have proved largely inadequate in exposing such subtle interleavings.

Prediction-based testing has emerged as a promising approach to testing concurrent programs. It involves taking one arbitrary concurrent execution of the program under test, and from that predict alternate interleavings that are more likely to contain bugs (interleavings that lead to data-races, interleavings that violate atomicity, etc.). Prediction replaces systematic search with a search for interleavings that are *close* to observed executions only, and hence is more tractable, and at the same time explores interesting interleavings that are likely to lead to errors [9, 10, 26, 27, 31].

In this paper, we explore a new target for predictive testing of concurrent programs that is fundamentally very different from data-races or atomicity errors: we propose to target executions that lead to *null-pointer dereferences*. Given an arbitrary execution of a concurrent program under test, we investigate fundamental techniques to accurately and scalably predict executions that are likely to lead to null-pointer dereferences.

Null-pointer dereferences can occur in a thread when dereferencing local variables. Consequently, an *accurate* prediction of null-pointer dereferences requires, by definition, handling local variables and the computation of threads. This is in sharp contrast to errors like data-races and atomicity violations, which depend only on *accesses to shared variables*.

The prediction algorithm aims to find some interleaving of the events in the observed run that will result in a null-pointer dereference. The naive approach to this problem is to reduce it to a constraint satisfaction problem; the set of constraints capture the semantics of local computations as well as the interaction of threads using reads and writes, concurrency control like locks, etc. A constraint solver could then solve these constraints and hence synthesize an interleaving that causes a null-pointer dereference to occur (this is similar to logic-based bounded model-checking for concurrent programs [24, 25]). However, this simply does not scale in realistic dynamic testing setting where the number of events that model reads/writes and computation can span millions of events. Consequently, accurate null-pointer dereference prediction seems intractable.

The main goal of this paper is to achieve useful and scalable prediction for finding runs that cause null-pointer dereferences. We propose a combination of four carefully chosen techniques to achieve this:

1. **Approximation:** An approximation of the prediction problem that ignores local computation entirely, and recasts the problem involving only the events that observe shared reads, writes, and concurrency control events (which we call the shared communication level) to achieve scalability. The prediction at the shared communication level can be more efficiently solved, using a combination of a lock-set based static analysis that identifies null read-write pairs and a constraint satisfaction problem to predict executions that force these null reads. Predicted runs in this model will be feasible but may not actually cause a null-pointer dereference, though they are likely to do so.
2. **Static Pruning:** An aggressive pruning of executions using static analysis based on vector-clocks that identifies a small segment of the observed run on which the prediction effort can be focused. This greatly improves the scalability of using sophisticated logic solvers. Pruning of executions does not affect feasibility of the runs, but may reduce the number of runs predicted. However, we show that in practice no additional errors were found without pruning.
3. **Relaxed prediction:** A formulation of the prediction at the shared communication level that allows some leeway so that the prediction algorithm can predict runs with mild *deviations* from the observed run which could be interesting runs; this makes the class of predicted runs larger at the expense of possibly making them *infeasible*, though in practice we found the majority of the predicted runs to be feasible.

[*] This work was funded partly by the NSERC Discovery Grant at University of Toronto and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign and by NSF Career Award #0747041.

[†] University of Toronto.

[‡] University of Illinois.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.
Copyright © 2012 ACM 978-1-4503-1614-9/12/11. . . $10.00

4. **Re-execution:** The runs predicted using the techniques may be infeasible, or may be feasible and yet not cause any null-pointer dereference. We mitigate this by a re-execution engine that executes predicted schedules accurately to check if a null-pointer dereference actually occurs. Errors reported hence are always real (i.e., they cause an uncaught exception or result in failing the test harness), and hence we incur no false positives.

We explain these techniques below, and motivate their choice and their impact.

Approximation to the shared communication level: Though null-pointer dereferences could occur on local variables and shared variables (unlike data-races and atomicity, which are defined only at the level of shared variables), a prediction that takes into account local variables and local computation simply does not scale (we have tried many experiments on such a model that validate this claim). We propose an approximation to null-pointer dereference prediction that works at the shared communication level. Consider a thread T that in some interleaving reads a shared variable x and subsequently does some computation locally using its value, and consider the task of predicting whether this could result in a null-pointer dereference. Our approximation of the prediction problem at the shared communication level asks for a run that forces the thread T to read a value of *null*. Note that this approximation is neither sound nor complete— thread T may read *null* for x but may not dereference the pointer (e.g., it could check if x is null), and there may be runs where the value read is not *null* and yet the local computation causes a null pointer dereference. However, such an approximation is absolutely necessary to scale to large runs, as it is imperative that local computation is not modeled. As our experiments demonstrate, this approximation does not inhibit us from finding interesting executions with null-pointer dereferences.

The above approximation poses the problem as a prediction problem at the shared communication level, which is a well-studied problem. In particular, there is a known *maximal causal model* that allows prediction at this level [23] and is the most precise prediction one can achieve at the shared communication level. Moreover, this prediction can be achieved using automatic *constraint solvers* that solve a constraint that demands a sequentially-consistent interleaving that respects the semantics of read/write of shared variables and the concurrency control mechanisms (locks, barriers, threads-creation, etc.). Maximal causality prediction using constraint solvers has been done in the past for data-races, atomicity, etc. (as they are properties already defined at the shared communication level) [22] and we utilize a similar technique with additional optimizations to approximately predict null-pointer dereferences.

Static pruning of executions: Prediction at the shared communication level, though more scalable than when modeling local computation, still has scalability issues in practice, as there can be in the order of hundreds of thousands of events involving shared variables. We propose pruning the execution using a simple scalable lock-set and vector-clock based analysis that determines a large prefix of the observed run that can be cut out *soundly*. The prediction algorithm hence is only applied to the remnant segment, which is smaller by an order of magnitude. If the prediction of the smaller segment succeeds, we are guaranteed that there is a way to stitch back the removed prefix to predict a full execution. Furthermore, the relaxed prediction technique (discussed below) being only applied to the smaller segment, introduces inaccuracies solely within that segment.

Relaxed prediction: The above two techniques of approximate prediction at the shared communication level and static pruning address scalability issues, and ensure that predicted runs are always feasible. However, they do not always work well in practice because sometimes *no solution* to the constraints exist. Demanding that the predicted run be absolutely guaranteed to be feasible (using the maximal causal model) seems too stringent a requirement, as it rules out runs that even mildly deviate from the requirements. For instance, if there is a read of y with value 23 in the original run, the predicted run is required to make the same read-event read the value 23, while in reality reading a different value, say 27, may not cause the program to completely deviate from the current path.

We propose a novel relaxed prediction algorithm that models the constraints to allow *bounded wiggle-room*. We propose to explore, for an increasing threshold k, whether there is a predictable run that violates at most k constraints that specify the values of shared reads. According to our experiments, even setting k to a small number (e.g., 5) is often sufficient to predict runs causing null-pointer dereferences. These predicted runs are not guaranteed to be feasible, but we show empirically that most of them actually are.

This technique, as well as the approximation to the shared communication level, causes unsoundness (predicted runs may be feasible or not cause a null-pointer dereference), which is handled by using an accurate re-execution tool that checks whether the predicted runs are indeed feasible and cause a null-pointer dereference.

The main technical contributions of this paper is the mechanism of targeting null-pointer dereference prediction using the approximation to the shared-communication level, the static pruning for scalability, and the prediction by relaxing constraints so as to make the prediction useful.

Evaluation: The final contribution of this paper is a full-fledged implementation realizing the new prediction-based testing tool that targets null-pointer dereferences, called EXCEPTIONULL . EXCEPTIONULL monitors and reschedules interleavings for Java programs using Java bytecode rewriting (building over the PENELOPE [27] infrastructure), and implements the identification of *null-WR* pairs, the pruning, and the logic-based procedures for both precise and relaxed prediction, using the SMT solver Z3 from Microsoft Research [11]. We show that EXCEPTIONULL is effective in predicting a large number of feasible runs in a suite of concurrent benchmarks. Evaluated over a suite of 13 benchmarks, we show the discovery of a slew of about 40 null-pointer dereference errors just be predicting from single test executions, and with no false positives. We know of no current technique that can find anywhere close to these many null-pointer dereferences on these benchmarks. We believe that the techniques proposed here hence make a significant leap forward in testing concurrent programs.

We also study some of the effects of techniques we have introduced, and estimate the inaccuracies caused and the scalability gained. We show that the pruning technique provides significant scalability benefits, while at the same time does not prohibit the prediction from finding any of the errors. We also show experimentally that the relaxation technique allows us to predict many more runs than the maximal causal model that result in errors (14 of the 41 errors were found due to relaxation).

We also show the efficacy of the relaxation technique by adapting our relaxed prediction to find *data-races*. Note that data-races are already defined at the shared communication level, and hence the approximation technique is not relevant. However, even in this setting, the static pruning allows us to scale more and the relaxation technique allows us to predict a lot more runs that have data-races than the strict prediction on the maximal causal model can (the latter is the current state-of-the-art in predicting data-races). Our tool discovers 60 data-races over our benchmarks of which 17 are found using relaxed prediction, showing that relaxed prediction is a very effective for other types of errors as well.

Related Work: The closest work related to ours in the realm of logic-based methods are those that stem from *bounded model-*

checking for finding executions of *bounded length* in concurrent programs that have bugs. Typically, a program's loops are unrolled a few times to get a bounded program, and using a logical encoding of the runs in this bounded program, a constraint solver is used to check if there is an error. We refer the reader to the papers from the NEC Labs group [24, 25] ([24] gives a clean encoding) as well as work from Microsoft Research [2, 12, 16, 21], where the programs are converted first to a sequential program from which bounded run constraints are generated. The crucial difference in our work is that we use logic in the *testing* setting to predict alternate interleavings. Another closely related work is ConMem [35] (see also [34]), where the authors target a variety of memory errors in testing concurrent programs, including null-pointer dereferences, but the prediction algorithms are much weaker and quite inaccurate compared to our robust prediction techniques. Furthermore, there is no accurate rescheduling engine which leads the tool to have many false positives.

There are two promising approaches that have emerged in testing concurrent programs: *selective interleavings* and *prediction-based testing* (and combinations of these). The selective interleaving approach is to focus in testing a *small* but carefully chosen subset of interleavings. There are several tools and techniques that follow this philosophy: for instance, the CHESS tool from Microsoft [18] tests all interleavings that use a bounded number preemptions (unforced context-switches), pursuing the belief that most errors can be made to manifest this way. Several tools concentrate on testing *atomicity violating patterns* (for varying notions of what atomicity means), with the philosophy that they are much more likely to contain bugs [17, 19, 20, 33]. However, systematically testing even smaller classes of interleavings is often impossible in practice, as there are often too many of them.

There are several work on prediction-based testing that do not use logical methods. These algorithms may focus on predicting runs violating *atomicity* or containing *data-races*: those by Sorrentino et al. [13, 27], those by Wang and Stoller [31, 32], and [14] by Huang and Zhang. A more liberal notion of generalized dynamic analysis of a single run has also been studied in a series of papers by Chen et al. [9, 10]. JPredictor [10] offers a predictive runtime analysis that uses *sliced causality* [9] to exclude the irrelevant causal dependencies from an observed run and then exhaustively investigates all of the interleavings consistent with the sliced causality to detect potential errors. The main drawback of non-logical prediction approaches is that the predicted runs may not be feasible. In fact, they ignore data which makes them less effective in finding bugs that are data-dependent such as null-pointer dereferences.

Logic-based prediction approaches, target precise prediction. Given an execution of the program, several work [29, 30] model the whole computation (local as well as global) logically to guarantee feasibility. The research presented in the above related work has too big an overhead to scale to large executions. Maximal Causality Model (MCM) [23], on the other hand, allows prediction at the level of shared communication and is the most precise prediction one can achieve at this level. MCM has been used by Said et al. [22] for finding data-race witnesses. We also use this model to predict runs leading to null-pointer dereferences.

2. Motivating Example

Consider a code extract from the `Pool 1.2` library [5] in the `Apache Commons` collection, presented in Figure 1. The object `pool`'s state, *open* or *closed*, is tested outside the synchronized block in method `returnObject`, by checking whether the flag variable `isClosed` is true. If so, then some local computation occurs, followed by a synchronized block that dereferences the shared object `pool`. A second method `close` closes the pool and sets `isClosed` to true to signal that the pool has been closed.

An error in this code (and such errors are very typical) stems from the fact that the check of `isClosed` in the method `returnObject` is not within the synchronized block; hence, if a thread executes the check at line ℓ, and then a concurrent thread executes the method `close()` before the synchronized block begins, then the access to the pool object at line ℓ' will raise an uncaught *null-pointer dereference exception*.

In a dynamic testing setting, consider the scenario where we observe an execution σ with two threads, where T executes the method `returnObject` first, and then, T' executes the method `close` after T finishes executing `returnObject`. There is no null-pointer dereference in σ. Our goal is to predict an alternate scheduling of events of σ that causes a null-pointer dereference.

Our prediction for null-pointer dereferences works as follows. In the run σ, a read of the shared variable `pool` at ℓ' occurs in T and the read value is not null. Also, a write to `pool` occurs in T' at ℓ'' which writes the value `null`. We ask whether there exists an alternative run σ' in which, the read at ℓ' (in T) can read the value `null` written by the write at location ℓ'' (in T') (as illustrated by the arrow in Figure 1).

Our prediction algorithm observes the shared events (such as shared reads/writes) but suppresses the semantics of local computations entirely and does not even observe them; they have been replaced by "..." in the figure as they play no role in our analysis.

Prediction of runs that force the read at ℓ' to read the null value written at ℓ'' must meet several requirements. Even if the predicted run respects the synchronization semantics for locks, thread creation, etc., the run may diverge from the observed run due to reading a different set of values for shared variables which will result in a different local computation path (e.g. the condition check at ℓ will stop the computation of the function right away if the value of `isClosed` is true). Therefore, we also demand that all other shared variable reads read the same value as they did in the original observed run, in order to guarantee that unobserved local computations will unfold in the same way as they did in the original run. This ensures the feasibility of the predicted runs.

Relaxed prediction: The requirement for all shared variable reads to read the same values, however, can be too strict in some cases. For instance, in our example, the variable `modCount` is a global counter keeping track of the number of modifications made to the `pool` data structure, and does not play any role in the local control flow reaching the point of null-pointer dereference at ℓ''. In the real execution leading to this null-pointer dereference, which is the one where block b_1 (from T) is executed first, followed by b_3 (from T') and then b_2 (from T), the read of `modCount` will read a different value than the corresponding value read in σ. However, this does not affect the feasibility of the run (in contrast to the value read for `isClosed`, which plays an important role in reaching the null-pointer dereference).

Our relaxed prediction model gives a slack threshold k, allowing predicted runs to have at most k reads that do not have to read the same values as in σ. By increasing the threshold k iteratively, our technique will find an execution that violates the read condition on `modCount`, but yet finds a feasible run that causes the null-pointer dereference in this example.

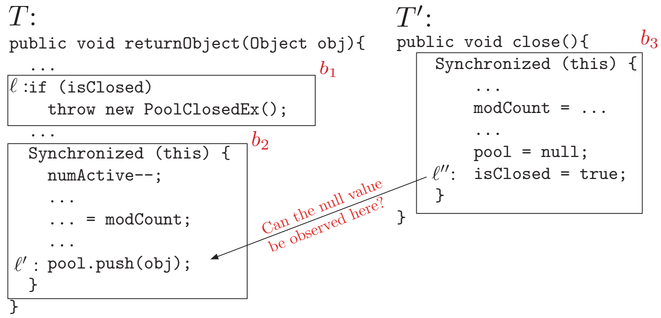
3. Preliminaries

Here, we present an overall overview of our prediction-based approach and set up a formal notation to describe the predicted runs.

3.1 Overview of proposed approach

Given a concurrent program P and an input I, we perform the following steps:

- **Monitoring:** We execute P on the input I and observe an arbitrarily interleaved run σ.



T:

```
public void returnObject(Object obj){
  ...
ℓ :if (isClosed)
    throw new PoolClosedEx();
  ...
  Synchronized (this) {
    numActive--;
    ...
    ... = modCount;
    ...
ℓ' : pool.push(obj);
  }
}
```

T':

```
public void close(){
  Synchronized (this) {
    ...
    modCount = ...
    ...
    pool = null;
ℓ'': isClosed = true;
  }
}
```

Figure 1. Code snippet of the buggy implementation of Pool.

- **Run Prediction:** We analyze the run σ to find a set of pair of events $\alpha = (e, f)$ in σ such that: (i) e is a write to shared variable x that writes a *null* value, (ii) f is a read from the same shared variable x that reads a non-null value in another thread, and (iii) static analysis on the run determines that there is a run $\widehat{\sigma}$ of P, obtained from reshuffling of events in σ, that respects locks and in which f reads the null value written by e. We call such pair of events $\alpha = (e, f)$ a *null-WR* pair. For each *null-WR* pair, we logically encode the set of runs and use SMT solvers to predict concrete runs $\widehat{\sigma}$ that force null-reads.
- **Rescheduling:** For each $\widehat{\sigma}$ generated by the *run prediction* phase, we re-execute the program, on the same input I, forcing it to follow $\widehat{\sigma}$. If it succeeds, then the null value read at f *may* later result in an error, such as a *null-pointer dereference exception*; we report all such confirmed errors.

We now set up the formal notation to describe the run prediction phase. In particular, the prediction algorithm will *ignore* computation of threads, and interleave at the level of blocks of local computations that happen between two reads/writes to global variables.

3.2 Modeling program runs, suppressing local computation

We model the runs of a concurrent program as a *word* where each letter describes the action done by a thread in the system. The word will capture the essentials of the run— shared variable accesses, synchronizations, thread-creating events, etc. However, we will *suppress* the local computation of each thread, i.e. actions a thread does by manipulating local variables, etc. that are not (yet) visible to other threads, and model the local computation as a *single* event *lc*. (In the formal treatment, we will ignore other concurrency constructs such as barriers, etc.; these can be accommodated easily into our framework.)

We fix an infinite countable set of thread identifiers $T = \{T_1, T_2, ..\}$ and define an infinite countable set of shared variable names SV that the threads manipulate. Without loss of generality, we assume that each thread T_i has a *single* local variable lv_i that reflects its entire local state. Let $V = SV \bigcup_i \{lv_i\}$ represent the set of all variables. Let $\mathit{Val}(x)$ represent the set of possible values that variable $x \in SV$ can get, and define $\mathit{Init}(x)$ as the initial value of x. We also fix a countable infinite set of locks L.

The actions that a thread T_i can perform on a set of shared variables SV and global locks L is defined as:

$$\begin{aligned}\Sigma_{T_i} = \{&T_i{:}\mathit{read}_{x,val},\ T_i{:}\mathit{write}_{x,val} \mid x \in SV, val \in \mathit{Val}(x)\} \\ &\cup \{\ T_i{:}\mathit{lc}\} \cup \{T_i{:}\mathit{acquire}(l),\ T_i{:}\mathit{release}(l) \mid l \in L\} \\ &\cup \{T_i{:}\ tc\ T_j \mid T_j \in T\}\end{aligned}$$

Actions $T_i{:}\mathit{read}_{x,val}$ and $T_i{:}\mathit{write}_{x,val}$ correspond to the thread T_i reading the value val from and writing the value val to the shared variable x, respectively. Action $T_i{:}\mathit{lc}$ corresponds to a local computation of thread T_i that accesses and changes the local state lv_i. Action $T_i{:}\ acquire(l)$ represents acquiring the lock l and the action $T_i{:}\ release(l)$ represents releasing of the lock l, by thread T_i. Finally, the action $T_i{:}\ tc\ T_j$ denotes the thread T_i creating the thread T_j.

We define $\Sigma = \bigcup_{T_i \in T} \Sigma_{T_i}$ as the set of actions of all threads. A word w in Σ^*, in order to represent a run, must satisfy several obvious syntactic restrictions, which are defined below.

Lock-validity, Data-validity, and Creation-validity: There are certain semantic restrictions that a run must follow. In particular, it should respect the semantics of locks and semantics of reads, i.e. whenever a read of a value from a variable occurs, the last write to the same variable must have written the same value, and the semantics of thread creation. These are captured by the following definitions ($\sigma|_A$ denotes the word σ projected to the letters in A).

DEFINITION 3.1 (Lock-validity). *A run $\sigma \in \Sigma^*$ is* lock-valid *if it respects the semantics of the locking mechanism. Formally, let $\Sigma_l = \{T_i{:}\ acquire(l), T_i{:}\ release(l) \mid T_i \in T\}$ denote the set of locking actions on lock l. Then σ is lock-valid if for every $l \in L$, $\sigma|_{\Sigma_l}$ is a prefix of*

$$\left[\bigcup_{T_i \in T}(T_i{:}acquire(l)\ T_i{:}release(l))\right]^* \qquad \blacksquare$$

DEFINITION 3.2 (Data-validity). *A run $\sigma \in \Sigma^*$ over a set of threads T, shared variables SV, and locks L, is* data-valid *if it respects the read-write constraints. Formally, for each n such that $\sigma[n] = T_i{:}\mathit{read}_{x,val}$, one of the following holds:*
(i) The last write action to x writes the value val. I.e. there is a $m < n$ such that $\sigma[m] = T_j{:}\mathit{write}_{x,val}$ and there is no $m < k < n$ such that $\sigma[k] = T_q{:}\mathit{write}_{x,val'}$ for any val' and any thread T_q, or (ii) there is no write action to variable x before the read, and val is the initial value of x. I.e. there is no $m < n$ such that $\sigma[m] = T_j : \mathit{write}_{x,val'}$ (for any val' and any thread T_j), and $val = \mathit{Init}(x)$. $\blacksquare$

DEFINITION 3.3 (Creation-validity). *A run $\sigma \in \Sigma^*$ over a set of threads T is* creation-valid *if every thread is created at most once and its events happen after this creation, i.e., for every $T_i \in T$, there is at most one occurrence of the form $T_j{:}\ tc\ T_i$ in w, and, if there is such an occurrence, then all occurrences of letters of Σ_{T_i} happen after this occurrence.* $\blacksquare$

Program Order: Let $\sigma = a_1...a_n$ be a run of a program P. The *occurrence* of actions in runs are referred to as *events* in this paper. Formally, the set of events of the run is $E = \{e_1, \ldots, e_n\}$, and there is a labeling function λ that maps every event to an action, given by $\lambda(e_u) = a_u$.

While the run σ defines a total order on the set of events in it $(E, \leq)$, there is an induced total order between the events of each thread. We formally define this as $\sqsubseteq_i$ for each thread T_i, as follows: for any $e_s, e_t \in E$, if a_s and a_t belong to thread T_i and $s \leq t$ then $e_s \sqsubseteq_i e_t$. The partial order that is the union of all the program orders is $\sqsubseteq = \cup_{T_i \in T} \sqsubseteq_i$.

The Maximal Causal Model for prediction: Given a run σ corresponding to an actual execution of a program P, we would like our prediction algorithms to synthesize new runs that interleave the events of σ to cause reading of null values. However, we want to predict *accurately*; in other words we want the predicted runs to be feasible in the actual program.

We now give a sufficient condition for a partial run predicted from an observed run to be *always* feasible. This model of prediction was defined by Şerbănuţă et al., and is called the *maximal causal model* [23]; it is in fact the most liberal prediction model that ensures that the predicted runs are always feasible in the program that work purely dynamically (i.e. no other information about the program is known other than the fact that it executed this set of observable events, which in turn do not observe computation).

We generalize the model slightly by taking into account thread creation.

DEFINITION 3.4 (Maximal causal model of prediction [23]). *Let σ be a run over a set of threads T, shared variables SV, and locks L. A run σ' is* precisely predictable *from σ if (i) for each $T_i \in T$, $\sigma'|_{T_i}$ is a prefix of $\sigma|_{T_i}$, (ii) σ' is lock-valid, (iii) data-valid, and (iv) creation-valid. Let $PrPred(\sigma)$ denote the set of all runs that are precisely predictable from the run σ.* ∎

The first condition above ensures that the events of T_i executed in σ' is a prefix of the events of T_i executed in σ. This property is crucial as it ensures that the local state of T_i can evolve correctly. Note that we are forcing the thread T_i to read the same values of global variables as it did in the original run. Along with data-validity, this ensures that the thread T_i reads precisely the same global variable values and updates the local state in the same way as in the original run. Lock-validity and creation-validity are, of course, required for feasibility. We will refer to runs predicted according to the maximal causal model (i.e. runs in $PrPred(\sigma)$) as the precisely predicted runs from σ.

The following soundness of the prediction that assures all predicted runs are feasible, follows:

THEOREM 3.5 ([23]). *Let P be a program and σ be a run corresponding to an execution of P. Then every precisely predictable run $\sigma' \in PrPred(\sigma)$ is feasible in P.* ∎

The above theorem is independent from the class of programs. We will assume however that the program is locally deterministic (non-determinism caused by threads interleaving is, of course, allowed). The above theorem, in fact, even holds when local computations of P are *non-deterministic*; i.e. the predicted runs will still be feasible in the program P. However, in order to be able to execute the predicted runs, we need to assume determinism of local actions. In this case, we can schedule the run σ' precisely and examine the outcomes of the tests on these runs.

3.3 The prediction problem for null-reads

We are now ready to formally define the precise prediction problem for forcing null-reads.

DEFINITION 3.6 (Precisely predictable null-reads). *Let σ be a run of a program P. We say that σ' is a* precisely predictable run that forces null-reads *if there is a thread T_i and a variable x such that the following are satisfied: (i) $\sigma' = \sigma''.f$ where f is of the form T_i: $read_{x,null}$, (ii) σ'' is a precisely predictable run from σ using the maximal causal model, and (iii) there is some val $\neq$ null such that $(\sigma''|_{\Sigma_i})$. T_i: $read_{x,val}$ is a prefix of $\sigma|_{\Sigma_i}$.* ∎

Intuitively, the above says that the run σ' must be a precisely predictable run from σ followed by a read of null by a thread T_i on variable x, and further, in the observed run σ, thread T_i must be executing a non-null read of variable x after executing its events in σ''. The above captures the fact that we want a precisely predictable run followed by a single null-read that corresponded to a non-null read in the original observed run. Note that σ' itself is not in $PrPred(\sigma)$, but is always feasible in the program P, and results in a null-read by thread T_i on variable x that had not happened in the original run.

The precisely predictable runs that force null-reads are hence excellent candidates to re-execute and test; if the local computation after the read does not check the null-ness of x before dereferencing a field of x, then this will result in an exception or error.

4. Identifying *null-WR* pairs using lock-sets

The first phase of our prediction is to identify *null-WR* pairs $\alpha = (e, f)$ where e is a write of null to a variable and f is a read of the same variable, but where the read in the original run reads a non-null value. Moreover, we would like to identify pairs that are feasible at least according to the hard constraints of thread-creation and locking in the program. For instance, if a thread writes to a shared variable x and reads from it in the same lock-protected region of code, then clearly the read cannot match a write protected by the same lock in another thread. Similarly, if a thread initializes a variable x to a non-null and then creates another thread that reads x, clearly the read cannot read an uninitialized x. We use a lock-set based static analysis of the run (without using a constraint solver) to filter out such impossible read-write pairs. The ones that remain are then subject to a more intensive analysis using a constraint solver.

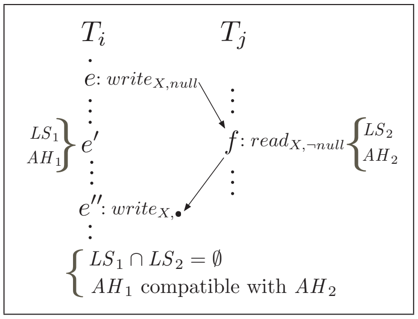
Using a static analysis on the observed run σ, we first collect all *null-WR* pairs $\alpha = (e, f)$. Then, we prune away *null-WR* pairs for which there is no lock-valid run in which f is reading the null value written by e. Then, for each *null-WR* pair $\alpha = (e, f)$ left, we use our precise logical prediction algorithm to obtain a lock-valid, data-valid and creation-valid run in which f is reading the null value written by e. However, instead of using run σ for the purposes of the prediction, we slice a *relevant* segment of it, and use the segment instead. The reason for this is twofold: (1) these run segments are often orders of magnitude smaller than the complete run, and this increases the scalability of our technique and (2) when a precisely predictable run does not exist, we use a more relaxed version of the constraints to generate a new run, limiting the improvisation to a smaller part of run increases our chances of obtaining a feasible execution. We formally define this *relevant run segment* and how it is computed in Section 7.

In this static analysis, the idea is to check if the *null-WR* pair $\alpha = (e, f)$ can be realized in a run that respects lock-validity and creation-validity constraints only (and not data-validity). Creation validity is captured by computing a vector clock associated with each event, where the vector clock captures only the hard causality constraints of thread creation. If f occurs before e according to this relation, then clearly it cannot occur after e and the pair is infeasible. Lock-validity is captured by reducing the problem of realizing the pair (e, f) to *pairwise reachability* under nested locking [15], which is then solved by computing lock-sets and acquisition histories for each event. We describe only the lock-validity checking below. Similar techniques have been exploited for finding *atomicity* violations in the tool PENELOPE [27].

Checking lock-valid reachability Consider a *null-WR* pair $\alpha = (e, f)$ and a run σ in which f (a read in thread T_j) occurs first, and later the write event e is performed by T_i.

Let us assume that e'' is the next write event (to the same variable accessed in e and f) in T_i after e. If there exists a lock-valid run σ' (obtained by permuting the events in σ) in which f reads the null value provided by e, then in σ', f should be scheduled after e, but before e''; if f is scheduled also after e'', then the write in e'' overwrites the null value written by e before it reaches f. This means that there should exist an event e' of thread T_i, occurring between events e and e'', that is executed right before (or after f) in σ'; in other words, e' and f are co-reachable.

We use a simple technique [15] to check if there is an event e' in T_i between e and e'' such that e' and f are co-reachable in a lock-valid run. The co-reachability check is done by examining the lock-sets and ac-


T_i
T_j
e: $write_{X,null}$
LS_1
AH_1
e'
f: $read_{X,\neg null}$
LS_2
AH_2
e'': $write_{X,\bullet}$
$LS_1 \cap LS_2 = \emptyset$
AH_1 compatible with AH_2

$$\psi \equiv PO \wedge CV \wedge DV \wedge LV$$

$$PO = (\bigwedge_{i=1}^{n} PO_i) \wedge C_{init}$$

$$C_{init} = \bigwedge_{i=1}^{n} (ts_{e_{init}} < ts_{e_{i,1}})$$

$$PO_i = \bigwedge_{j=1}^{m_i - 1} (ts_{e_{i,j}} < ts_{e_{i,(j+1)}})$$

$$DV = \bigwedge_{x} \quad \bigwedge_{val \in Val(x)} \quad \bigwedge_{r \in R_{x,val}} \left(\bigvee_{w' \in W_{x,val}} Coupled_{r,w'} \right)$$

$$Coupled_{r,w} = (ts_w < ts_r) \wedge \bigwedge_{e'' \in W_x - \{w\}} ((ts_{e''} < ts_w) \vee (ts_r < ts_{e''}))$$

$$LV = LV_1 \wedge LV_2$$

$$LV_1 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{[e_{ac}, e_{rel}] \in L_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e_{rel}} < ts_{e'_{ac}} \vee \ ts_{e'_{rel}} < ts_{e_{ac}} \right)$$

$$LV_2 = \bigwedge_{i \neq j \in \{1,..,n\}} \bigwedge_{lock\ l} \bigwedge_{\substack{e_{ac} \in NoRel_{i,l} \\ [e'_{ac}, e'_{rel}] \in L_{j,l}}} \left(ts_{e'_{rel}} < ts_{e_{ac}} \right)$$

Figure 2. Constraints capturing the maximal causal model.

quisition histories at e' and f: the lock-sets at e' and f must be disjoint and the acquisition histories at e' and f must be compatible.

Note that the above condition is *necessary* for the existence of the lock-valid run σ', but not *sufficient*; hence filtering out pairs that do not meet this condition is sound.

5. Precise prediction by logical constraint solving

We now describe how we solve the problem of precisely predicting a run that realizes a *null-WR* pair $\alpha = (e, f)$. This problem is to predict whether there is an alternate schedule in the maximal causal model that forces the read at f to read the null value written by e. We solve this using a logic constraint solver (an SMT solver); the logic of the constraints is in a fragment that is efficiently decidable.

Prediction according to the maximal causal model is basically an encoding of the creation-validity, data-validity, and lock-validity constraints using logic, where quantification is removed by expanding over the finite set of events under consideration. Modeling this using constraint solvers has been done before ([22]) in the context of finding data races. We reformulate this encoding briefly here for several reasons. First, this makes the exposition self-contained, and there are a few adaptations to the null-read problem that need explanation. Second, we perform a wide set of carefully chosen optimizations on this encoding, whose description needs this exposition. And finally, the relaxation technique, which is one of the main contributions of this paper, is best explained by referring directly to the constraints.

Capturing the maximal causal model using logic

Given a run σ, we first encode the constraints on all runs predicted from it using the maximal causal model, independent of the specification that we want runs that match a given *null-WR* pair . A predicted run can be seen as a total ordering of the set of events E of the run σ. We use an integer variable ts_e to encode the *timestamp* of event $e \in E$ when e occurs in the predicted run. Using these timestamps, we logically model the constraints required for precisely predictable runs (see Definition 3.4), namely that the run respect the program order of σ, that it be lock-valid, data-valid, and creation-valid.

Figure 2 illustrates the various constraints. The constraints are a conjunction of program order constraints (PO), creation-validity constraints (CV), data-validity constraints (DV), and lock-validity constraints (LV).

The program order constraint (PO) captures the condition that the predicted run respect the program order of the original observed run. Suppose that the given run σ_α consists of n threads, and let $\sigma_\alpha|_{T_i} = e_{i,1}, e_{i,2}, ..., e_{i,m_i}$ be the sequence of events in σ_α that relates to thread T_i. Then the constraint PO_i demands that the time-stamps of the predicted run obey the order of events in thread T_i, and PO demands that all threads meet their program order. We also consider an initial event e_{init} which corresponds to the initialization of variables. This event should happen before any thread starts the execution in any feasible permutation, and is encoded as the constraint C_{init}.

Turning to creation-validity, suppose that $e_{tc(i)}$ is the event that creates thread T_i. Then the constraint CV demands that the first event of T_i can only happen after $e_{tc(i)}$. Combined with program order constraint, this means that all events before the creation of T_i in the thread that created T_i must also occur before the first event of T_i.

The data-validity constraints DV (see Definition 3.3) capture the fact that reads must be coupled with appropriate writes; more precisely, that every read of a value from a variable must have a write before it writing that value to that variable, and moreover, there is no other intermediate write to that variable. Let $R_{x,val}$ represent the set of all read events that read value val from variable x in σ_α, W_x represent the set of all write events to variable x, and $W_{x,val}$ represent the set of all write events that specifically write value val to variable x. For each read event $r = read_{x,val}$ and write event $w \in W_{x,val}$, the formula $Coupled_{r,w}$ represents the requirement that w is the most recent write to variable x before r and hence r is coupled with w. The constraint DV demands that all reads be coupled with writes that write the same value as the read reads.

Lock-validity is captured by the formula LV. We assume that each lock acquire event ac of lock l in the run is matched by precisely one lock release event rel of lock l in the same thread, unless the lock is not released by the thread in the run. We call the set of events in thread T_i between ac and rel a lock block corresponding to lock l represented by $[ac, rel]$. Let $L_{i,l}$ be the set of lock blocks in thread T_i regarding lock l. Then LV_1 asserts that no two threads can be simultaneously inside a pair of lock blocks $[e_{ac}, e_{rel}]$ and $[e'_{ac}, e'_{rel}]$ corresponding to the same lock l. Turning to locks that never get released, the constraint LV_2 handles asserts that the acquire of lock l by a thread that never releases it must always occur after the releases of lock l in every other thread. In this formula, $NoRel_{i,l}$ stands for lock acquire events in T_i with no corresponding later lock release event.

Optimizations

The constraints, when written out as above, can be large. We do several optimization to control the formula bloat (while preserving the same logical constraint).

The data-validity constraint above is expensive to express, as it is, in the worst case, cubic in the maximum number of accesses to any variable. There are several optimizations that reduce the number of constraints in the encoding. Suppose that $r = read_{x,val}$ is performed by thread T_i.

- Each write event w' to x that occurs after r in T_i, i.e. $r \sqsubseteq_i w'$, can be excluded in the constraints related to coupling r with a write in constraint DV above.
- Suppose that w is the most recent write to x before r in T_i. Then, each write event w' before w in T_i, (i.e. $w' \sqsubseteq_i w$), can be excluded in the constraints related to coupling r with a write in constraint DV above.
- When r is being coupled with $w \in W_{x,val}$ in thread T_j, each write event w' before w in T_j, i.e. $w' \sqsubseteq_j w$, can be excluded as candidates for e'' in the formula $Coupled_{r,w}$.

- Suppose that r is being coupled with $w \in W_{x,val}$ in thread T_j and w' is the next write event to x after w in thread T_j. Then each write event w'' after w' in T_j, i.e. $w' \sqsubseteq_j w''$, can be excluded as candidates for e'' in the formula $Coupled_{r,w}$.
- Event r can be coupled with e_{init} only when there is no other write event to x before r in T_i, i.e. $\nexists w.\ (w \sqsubseteq_i r \wedge w \in W_x)$. Furthermore, it is enough to check that the first write event to x in each thread (if it exists) is performed after r.

The lock-validity formula above, which is quadratic in the number of lock blocks, is quite expensive in practice. We can optimize the constraints. If a read event r in thread T_i can be coupled with only *one* write event w which is in thread T_j then in all precisely predictable runs, w should happen before r. Therefore, the lock blocks according to lock l that are in T_j before w and the lock blocks according to lock l that are in T_i after r are already ordered. Hence, there is no need to consider constraints preventing T_i and T_j to be simultaneously in such lock blocks. In practice, this greatly reduces the number of constraints. Furthermore, when considering lock acquire events with no corresponding release events in LV_2 above, it is sufficient to only consider the *last* corresponding lock blocks in each thread and exclude the earlier ones from the constraint.

Predicting runs for a *null-WR* pair

We adapt the above constraints for predicting in the maximal causal model to predict whether a *null-WR* pair $\alpha = (e, f)$ is realizable. Suppose that σ and $\alpha = (e, f)$ are a run and a *null-WR* pair passed to the prediction phase, respectively. Notice that in the original run f reads a non-null value while we will force it to read null in the predicted run by coupling it with write event e. Indeed, this is the whole point of predicting runs— we would like to diverge from the original run at f by forcing f to read a null value. Note that once f reads a different value, we no longer have any predictive power on what the program will do (as we do not examine the code of the program but only its runs). Consequently, we cannot predict any events causally later than f.

The prediction problem is hence formulated as follows:

Given a run σ, and a null-WR pair $\alpha = (e, f)$ in σ, algorithmically find a precisely predictable run from σ that forces null-reads according to α; i.e. f is the last event and reads the null value written by e.

The prediction problem is to find precisely predicted runs that execute e followed by f, while avoiding any other write to the corresponding variable between e and f. The constraints that force the read f be coupled with the write e is $NC = Coupled_{f,e}$.

Furthermore, recall that the feasibility of the run that we are predicting needs to be ensured only *up to* the read f. Consequently, we drop from the data-validity formula that the value read at f (in the original run) match the last write (it should instead match e as above).

A further complication is scheduling events that happen after e in the same thread. Note that some of these events may need to occur in order to satisfy the requirements of events before f (for instance a read before f may require a write after e to occur). However, we may not want to predict some events after e, as we are really only concerned with f occurring after e. Our strategy here is to let the solver figure out the precise set of events to schedule after e (and before the next write to the same variable as e is writing to) in the same thread.

For events after e in T_i, we enforce lock-validity and data-validity constraints only if they are scheduled *before* f. More precisely, we replace $\vee_{w'} Coupled_{r_i,w'}$ in the formula DV to $(ts_r < ts_f \Rightarrow \vee_{w'} Coupled_{r_i,w'})$. Similarly, we drop the lock constraints on events occurring after f (this relaxation is more involved but straightforward).

In summary, we have reduced the problem of predicting a run according to the maximal causal model that causes the null write-read pair to be realizable to a satisfiability of a formula ψ in logic. The constraints generated fall within the class of quantifier-free difference logic constraints which SMT solvers efficiently solve in practice.

6. Relaxed prediction

The encoding proposed in the previous section is sound, in the sense that it guarantees feasibility of the predicted runs. However, as demonstrated by the example in Section 2, sound prediction under the maximal causal model can be too restrictive and result in predicting no runs. Slightly diverging from the original can sometimes lead to prediction of runs that are feasible in the original program.

We hence have a tension between two choices— we would like to maintain the same values read for as many shared variable reads as possible to increase the probability of getting a feasible run, but at the same time allow a few reads to read different values to make it possible to predict some runs. Our proposal, which is one of the main contributions of this paper, is an iterative algorithm for finding the *minimum* number of reads that can be exempt from data-validity constraints that will allow the prediction algorithm to find at least one run. We define a suitable *relaxed* logical constraint system to predict such a run. Our experiments show that exempting a few reads from data-validity constraints greatly improves the flexibility of the constraints and increases the possibility of predicting a run, and at the same time, the predicted runs are often feasible.

The iterative algorithm works as follows. Let's assume there are n shared variable reads that are required to be coupled with specific write by the full set of data-validity constraints. The data-validity constraints are expressed so that we specifically ask for n shared reads to be coupled correctly. If we fail to find a solution satisfying constraints for all n reads, then we repeatedly decrement n, and attempt to find a solution that couples $n - 1$ reads in the next round, and so on. The procedure stops whenever a run (solution) is found. The change required in the encoding to make this possible is described below.

For every read event $r_i \in R$, we introduce a new Boolean variable, b_i, that is true if the data-validity constraint for r_i is satisfied, and false otherwise. In addition, we consider an integer variable $bInt_i$ which is initially 0, and set to 1 only when b_i is true. This is done through a set of constraints, one for each $r_i \in R$: $[(b_i \rightarrow bInt_i = 1) \wedge (\neg b_i \rightarrow bInt_i = 0)]$. Also, for each $r_i \in R$, we change the sub-term $\vee_{w'} Coupled_{r_i,w'}$ to $(ts_r < ts_f) \Rightarrow (b_i \Rightarrow \vee_{w'} Coupled_{r_i,w'})$ in DV, forcing the data-validity constraint for read r_i to hold when b_i is true. Note that with these changes, we require a different theory, that is *Linear Arithmetic* in the SMT solver to solve the constraints, compared to the *Difference Logic* which was used for our original set of constraints.

Initially, we set a threshold η to be $|R|$, the number of all read events. In each iteration, we assert the constraint $\sum_{1 \leq i \leq |R|} bInt_i = \eta$, which specifies the number (η) of data-validity constraints that should hold in that iteration. If no run can be predicted with the current threshold η (i.e. the constraint solver reports unsatisfiability), then η is decremented in each iteration, until the formula is satisfiable. This way, when a satisfying assignment is found, it is guaranteed to have the maximum number of reads that respect data-validity possible for predictable run.

Note that once $\eta < |R|$, the predicted run is not theoretically guaranteed to be a feasible run. However, in practice, when η is close to $|R|$ and a run is predicted, this run is usually feasible in the program.

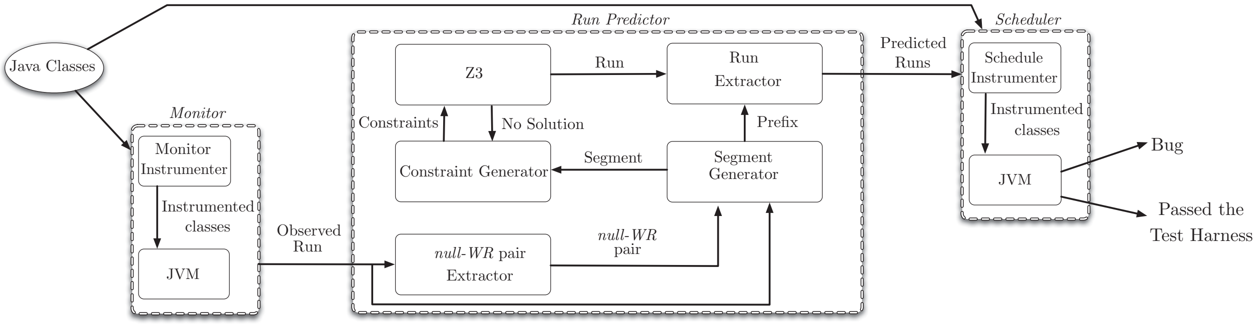


Figure 3. EXCEPTIONULL.

7. Pruning executions for scalability

Identifying *null-WR* pairs using the lock-set based analysis and then subjecting them to constraint checking is a precise method to force null reads. However, in our experiments, we discovered that the constraint solving approach does not scale well when runs get larger. In this section, we propose a pruning technique for the runs that removes a large prefix of them while maintaining the property that any run predicted from the suffix will still be feasible. While this limits the number of predictable runs in theory, we show that in practice, it does not prevent us from finding errors (in particular, *no error* was missed due to pruning in our experiments). Furthermore, we show that in practice pruning improves the scalability of our technique, in some cases by an order of magnitude.

Consider an execution σ and a *null-WR* pair $\alpha = (e, f)$. The idea behind pruning is to first construct the causal partial order of events of σ, and then remove two sets of events from it. The first set consists of events that are *causally after* e and f (except for some events, as described in detail below). The second set is a causally prefix-closed set of events (a configuration) that are *causally before* e and f, and where all the locks are free at the end of execution of this configuration. The intuition behind this is that such a configuration can be replayed in the newly predicted execution precisely in the same way as it occurred in the original run, and then stitched to a run predicted from the suffix, since the suffix will start executing in a state where no locks are held.

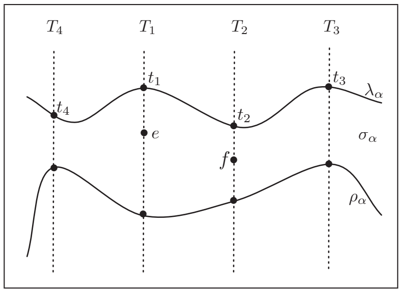
In order to precisely define this run segment, we define a notion of partial order on the set of events E that captures the causal order. Let D denote the dependency relation between actions that relates two actions of the same thread, reads and writes on the same variable by different threads, and lock acquisition and release actions of the same lock in different threads. We define the partial order $\preceq \subseteq E \times E$ on the set of program events as the *least* partial order relation that satisfies the condition that $(e_i, e_j) \in \preceq$ whenever $a_i = \sigma[i]$, $a_j = \sigma[j]$, $i \leq j$, and $(a, a') \in D$ where a_i and a_j are actions performed by events e_i and e_j, respectively.

Let us define ρ_α as the *smallest* subset of events of σ that satisfies the following properties: (1) ρ_α contains events e and f, (2) for any event e' in ρ_α, all events $e'' \preceq e'$ are in ρ_α, and (3) for every event corresponding to a lock $acquire$ in ρ_α, its corresponding *release* event is also in ρ_α.

The intuition is that events that are not in ρ_α are not relevant for the scheduling of the *null-WR* pair ; they are either far enough in the future, or are not dependent on any of the events in ρ_α. The figure below presents a run of a program with 4 threads that is projected into individual threads. Here, e belongs to thread T_1 and f belongs to thread T_2. The cut labeled ρ_α marks the boundary after which all events are not *causally before* e and f, and hence, need not be considered for the generation of the new run.

Next, we identify a causally prefix-closed set of events before e and f to remove. For the *null-WR* pair α, define λ_α as the *largest* subset of events of ρ_α that has the following properties: (1) it does not contain e or f, (2) for any event e' in λ_α, all events $e'' \preceq e'$ are in λ_α, and (3) for any event e' in T_i such that e' is the last event of T_i in λ_α (with respect to $\sqsubseteq_i$), the lockset associated to e' in T_i is empty. In the above figure, the curve labeled λ_α marks the boundary of λ_α, and events $T_1, \ldots, T_4$ have empty lock-sets.

The run segment relevant to a *null-WR* pair α is then defined as the set of events in $\sigma_\alpha = \rho_\alpha \setminus \lambda_\alpha$ scheduled according to the total order in σ ($\leq$). One can use a simple worklist algorithm to compute both ρ_α and λ_α, and consequently σ_α. This run segment is passed to the run prediction phase, in the place of the whole run σ.



8. Implementation

We have implemented our approach in a tool named EXCEPTIONULL . Figure 3 demonstrates the architecture of EXCEPTIONULL . It consists of three main components: a monitor, a run predictor, and a scheduler. The monitor and scheduler are built on top of the PENELOPE tool framework, with considerable enhancements and optimizations, including the extension of the monitoring to observe values of shared variables at reads and writes. In the following, we will explain each of these components in more details.

Monitor: The monitor component has an instrumenter which uses the Bytecode Engineering Library (BCEL) [4] to (automatically) instrument every class file in bytecode so that a *call* to an event recorder is made after each *relevant* action is performed. These relevant actions include field and array accesses, acquisition and releases of locks, thread creations and thread joins, etc., but exclude accesses to local variables. The instrumented classes are then used in the Java Virtual Machine (JVM) to execute the program and get an observed run. For the purpose of generating the data-validity constraints, the values read/written by shared variable accesses are also recorded. For variables with primitive types (e.g. Boolean, integer, double, etc), we just use the values read/written. Objects and arrays are treated differently; the object *hash code* (by `System.identityHashCode()`) is used as the value every time an object or an array is accessed.

Run Predictor: The run predictor consists of several components: *null-WR* pair extractor, segment generator, constraint generator, Z3 SMT solver, and run extractor. The *null-WR* pair extractor generates a set of *null-WR* pairs from the observed run by the static lock analysis described in Section 4. The segment generator component, for each *null-WR* pair $\alpha = (e, f)$, isolates a part of

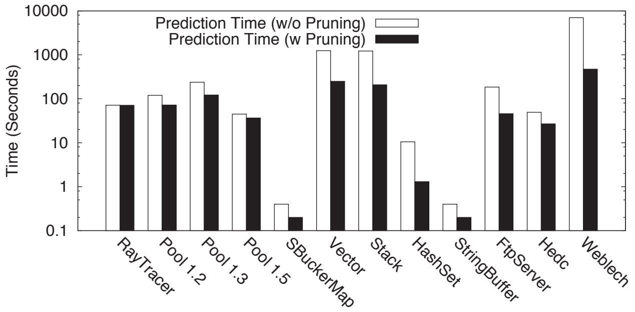


Figure 4. Prediction times with/without pruning in log scale.

run ρ that is *relevant* to α as described in Section 7 and passes it to the constraint generator. Given a *null-WR* pair and the relevant segment, the constraint generator produces a set of constraints, based on the algorithm presented in Section 5, and passes it to the Z3. Any model found by Z3 corresponds to a concurrent schedule. The run extractor component generates a run based on the model returned by Z3. When Z3 cannot find a solution, the constraint generator iteratively weakens the constraints (see Section 6) and calls Z3 until a solution is found.

Scheduler: The scheduler is implemented using BCEL [4] as well; we instrument the scheduling algorithm into the Java classes using bytecode transformations, so that the program interacts with the scheduler when it is executing the same set of events that were monitored. The scheduler, at each point, looks at the predicted run, and directs the appropriate thread to perform a sequence of n steps. The threads wait for a signal from the scheduler to proceed, and only then do they execute the number of (observable) events they are instructed to execute. Afterwards, the threads communicate back to the scheduler, relinquishing the processor, and await further instructions. The communication between the scheduler and the threads is implemented using wait-notify synchronization which allows us to have a finely orchestrated scheduling process.

Data-Race Prediction: Our proposed monitoring, logic-based precise and relaxed prediction on statically pruned runs, and the rescheduling is a general framework and can be adapted to errors other than null-pointer dereferences as well. In order to study the effects of relaxed prediction and static pruning, we implemented a data-race prediction unit as well to our tool, as data-races are a more well studied class of errors for which precise prediction has been studied. Due to lack of space, we do not discuss the details of the data race detection unit here.

9. Evaluation

We subjected EXCEPTIONULL to a benchmark suite of 13 concurrent programs, against several test cases and input parameters. Experiments were performed on an Apple MacBook with 2 Ghz Intel Core 2 Duo processors and 2GB of memory, running OS X 10.4.11 and Sun's Java HotSpot 32-bit Client VM 1.5.0.

Benchmarks. The benchmarks are all concurrent Java programs that use `synchronized` blocks and methods as means of synchronization. They include `RayTracer` from the Java Grande multi-threaded benchmarks [3], `elevator` from ETH [28], `Vector`, `Stack`, `HashSet` and `StringBuffer` from Java libraries, `Pool` (three different releases) and `StaticBucketMap` from the Apache Commons Project [5], `Apache FtpServer` from [7], `Hedc` from [6], and `Weblech` from [8]. `elevator` simulates multiple lifts in a building; `RayTracer` renders a frame of an arrangement of spheres from a given view point; `Pool` is an object pooling API in the Apache Commons suite; `StaticBucketMap` is a thread-safe implementation of the Java Map Interface; `Apache FtpServer` is a FTP server by Apache; and `Vector`, `Stack`, `HashSet` and `StringBuffer` are Java libraries that respectively implement the concurrent vector, the concurrent stack, the HashSet and the StringBuffer data structures. `Hedc` is a Web crawler application and `Weblech` is a websites download tool.

Experimental Results. Table 1 illustrates the experimental results for *null-pointer dereference prediction*; information is provided about all the three phases of monitoring, run prediction, and scheduling.

In the monitoring phase, the number of threads, shared variables, locks, the number of potential interleaving points (i.e. number of global events), and the time taken for monitoring are reported. For the prediction phase, we report the number of *null-WR* pairs in the observed run, the number of precisely predicted runs, and the additional number of predicted runs after relaxing the data-validity constraints (when there is no precisely predicted run for a null read-write pair). In the scheduling phase, we report the total number of schedulable predictions among the predicted ones. Finally, we report the average time for prediction and rescheduling of each run, the total time taken to complete the tests (for on all phases on all predicted executions), and also the number of errors found using the precise and relaxed predicted runs.

Errors Found. In almost all the cases, the errors manifested in the form of raised exceptions during the execution. In `Weblech`, in addition to a null-pointer dereference, an unwanted behavior occurred (the user is asked to push a stop button even after the website is downloaded completely, resulting in non-termination!). `RayTracer` has a built-in validation test which was failed in some of the predicted runs. For some of the test cases of `Vector` and `Stack` the output produced was not the one expected. We report the errors found in two categories; those that were found through the precise prediction algorithm, and those that were found after weakening data-validity constraints (relaxation).

The effect of pruning: Figure 4 illustrates the substantial impact of our pruning algorithm in reducing prediction time. We present prediction times with and without using the pruning algorithm. Note that the histogram is on a logarithmic scale. For example, in the case of `Weblech`, the prediction algorithm is about 16 times faster with pruning. Furthermore, all errors found without the pruning were found on the pruned runs, showing that the pruning did not affect the quality of error-finding on our benchmarks.

Data Race Detection. Table 2 presents the results of data-race prediction on our benchmarks using the same observed runs as in the null-reads prediction. For each benchmark we report the total number of data-races found; these are all distinct races identified by the code location of the racy access. We also report the number of distinct variables involved in data-races. For brevity, information about different test cases is aggregated for each benchmark (see [1] for more details).

Observations: EXCEPTIONULL performs remarkably well, predicting a large number of feasible program runs on which there are null-pointer dereferences and data-races. In total, it finds about 40 executions with null-pointer dereferences and 60 races, which to our knowledge, is the most successful attempt at finding errors on

Application	Elevator	RayTracer	Pool 1.2	Pool 1.3	Pool 1.5	SBucketMap	Vector	Stack	HashSet	StringBuffer	FtpServer	Hedc	Weblech	**Total**
Num. of Data-Races (by precise prediction)	-	3	5	-	8	1	1	1	6	-	8	5	5	**43**
Additional Data-Races (by relaxation)	-	3	0	-	2	0	1	1	3	-	3	0	4	**17**
Number of Distinct Involved Variables	-	1	3	-	4	1	2	2	3	-	7	4	3	**31**

Table 2. Experimental Results for data race prediction.

			Monitoring					Prediction			Scheduling				
Application (LOC)	Input	Base	Num. of Threads	Num. of Shared Variables	Num. of Locks	Num. of Potential Interleaving Points	Time to Monitor	Num. of *null-WR* Pairs	Num. of Precisely Predicted Runs	Additional Predicted Runs by Relaxation	Num. of Schedulable Predictions	Average Time per Predicted Run	Total Time	Null Pointer Deref. by Precise Prediction	Additional Null-Pointer Deref. by Relaxation
`Elevator` (566)	Data	7.3s	3	116	8	14K	7.4s	0	-	-	-	-	7.9s	**0**	**0**
	Data2	7.3s	5	168	8	30K	7.4s	0	-	-	-	-	8.9s	**0**	**0**
	Data3	19.2s	5	723	50	150K	19.0s	0	-	-	-	-	58.5s	**0**	**0**
`RayTracer` (1.5K)	A-10	5.0s	10	106	10	648	5.0s	9	9	-	9	5.6s	50.5s	**1***	**0**
	A-20	3.6s	20	196	20	1.7K	4.4s	19	19	-	19	6.7s	2m15s	**1***	**0**
	B-10	42.4s	10	106	10	648	42.5s	9	9	-	9	42.7s	6m24s	**1***	**0**
`Pool 1.2` (5.8K)	PT1	<1s	4	28	1	98	<1s	3	2	1	3	<1s	1.6s	**2**	**0**
	PT2	<1s	4	29	1	267	<1s	3	0	0	-	-	8.8s	**0**	**0**
	PT3	<1s	4	20	3	180	<1s	26	0	23	16	1.2s	27.0s	**0**	**3**
	PT4	<1s	4	24	3	360	<1s	32	2	21	15	2.5s	57.8s	**0**	**1**
`Pool 1.3` (7K)	PT1	<1s	4	30	1	100	<1s	3	0	3	3	<1s	2.6s	**0**	**0**
	PT2	<1s	4	31	1	271	<1s	3	0	0	-	-	9.8s	**0**	**0**
	PT3	<1s	4	20	3	204	<1s	35	0	30	19	1.4s	42.9s	**0**	**0**
	PT4	<1s	4	23	3	422	<1s	62	1	48	29	2.2s	1m49s	**0**	**1**
`Pool 1.5` (7.2K)	PT1	<1s	4	33	2	124	<1s	2	0	1	1	1.5s	1.5s	**0**	**0**
	PT2	<1s	4	34	2	306	<1s	5	0	1	0	10.5s	10.5s	**0**	**0**
	PT3	<1s	4	15	2	108	<1s	3	0	0	-	-	4.1s	**0**	**0**
	PT4	<1s	4	18	2	242	<1s	18	1	7	8	3.4s	27.4s	**0**	**1**
`SBucketMap` (750)	SMT	<1s	4	123	19	892	<1s	2	2	-	2	<1s	1.3s	**1**	**0**
`Vector` (1.3K)	VT1	<1s	4	44	2	370	<1s	21	11	10	21	<1s	14.3s	**2**	**0**
	VT2	<1s	4	34	2	536	<1s	31	21	10	31	1.1s	33.0s	**1**	**0**
	VT3	<1s	4	34	2	443	<1s	32	22	10	32	<1s	22.1s	**1**	**0**
	VT4	<1s	4	29	2	517	<1s	30	0	30	30	2s	59.4s	**0**	**1***
	VT5	<1s	4	29	2	505	<1s	85	1	84	82	2s	2m57s	**0**	**1***
`Stack` (1.4K)	ST1	<1s	4	29	2	205	<1s	11	6	5	11	<1s	5.5s	**2**	**0**
	ST2	<1s	4	24	2	251	<1s	16	11	5	15	<1s	10.9s	**1**	**0**
	ST3	<1s	4	24	2	248	<1s	17	12	5	17	<1s	10.3s	**1**	**0**
	ST4	<1s	4	29	2	515	<1s	30	0	30	30	1.8s	53.2s	**0**	**1***
	ST5	<1s	4	29	2	509	<1s	85	1	84	83	2.0s	2m51s	**0**	**1***
`HashSet` (1.3K)	HT1	<1s	4	76	1	432	<1s	7	7	-	7	<1s	3.2s	**1**	**0**
	HT2	<1s	4	54	1	295	<1s	0	-	-	-	-	<1s	**0**	**0**
`StringBuffer` (1.4K)	SBT	<1s	3	16	3	80	<1s	2	2	-	2	<1s	1.3s	**1**$^{+}$	**0**
`Apache FtpServer` (22K)	LGN	1m2s	4	112	4	582	60s	116	78	32	65	1m13s	2h14m46s	**9**	**3**
`Hedc` (30K)	Std	1.7s	7	110	6	602	1.74s	18	9	1	10	11.7s	1m57s	**1**	**0**
`Weblech v.0.0.3` (35K)	Std	4.9s	3	153	3	1.6K	4.92s	55	10	29	30	16.26s	10m34s	**1**	**1**$^{@}$
											Total Number of Errors			**27**	**14**

Table 1. Experimental results for predicting null-reads. Errors tagged with * represent test harness failures. Errors tagged with $^{+}$ represent array-out-of-bound exceptions. Errors tagged with $^{@}$ represent unexpected behaviors. All other errors are null-pointer dereference exceptions.

these benchmarks in the literature. Furthermore, all the errors are completely reproducible deterministically using the scheduler.

We count exceptions raised in different parts of the code as separate errors. More precisely, each error reported in Table 1 consists of a unique read-write pair in the code that were forced to perform a null-read and that resulted in an error. For example, the 12 exceptions in `FtpServer` are raised in 7 different functions and at different locations inside the functions, and involve null-pointer dereferences on 5 different variables.

The prediction algorithm works extremely well— while there were several runs that were predicted in the precise model, the relaxed prediction gives a lot more predictions, and a large fraction of these were schedulable. The time taken for prediction and scheduling are very reasonable for the kind of targeted analysis that we perform, despite the use of fairly sophisticated static analysis and logic-solvers.

The number of data-races found using relaxed prediction further shows the efficacy of relaxed prediction. A further 17 data-races were found using relaxed prediction, showing that predicting beyond the maximal causal model can be effective even in finding errors other than null-pointer dereferences.

References

[1] http://www.cs.uiuc.edu/∼sorrent1/penelope/exceptionull.

[2] http://research.microsoft.com/en-us/projects/poirot.

[3] http://www.javagrande.org/.

[4] http://jakarta.apache.org/bcel.

[5] http://commons.apache.org.

[6] http://www.hedc.ethz.ch.

[7] http://mina.apache.org/ftpserver.

[8] http://weblech.sourceforge.net.

[9] F. Chen and G. Roşu. Parametric and sliced causality. In *CAV*, pages 240–253, 2007.

[10] F. Chen, T.F. Serbanuta, and G. Roşu. JPredictor: a predictive runtime analysis tool for java. In *ICSE*, pages 221–230, 2008.

[11] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[12] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

[13] A. Farzan, P. Madhusudan, and F. Sorrentino. Meta-analysis for atomicity violations under nested locking. In *CAV*, pages 248–262, 2009.

[14] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *ISSTA*, pages 144–154, 2011.

[15] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV*, pages 505–518, 2005.

[16] S.K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV*, pages 509–524, 2009.

[17] Zhifeng Lai, S.C. Cheung, and W.K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, pages 235–244, 2010.

[18] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[19] C-S Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, pages 135–145, 2008.

[20] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[21] Z. Rakamarić. STORM: static unit checking of concurrent programs. In *ICSE*, pages 519–520, 2010.

[22] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an smt-based analysis. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] T.F. Şerbănuţă, F. Chen, and G. Roşu. Maximal causal models for sequentially consistent systems. Technical report, University of Illinois at Urbana-Champaign, October 2011.

[24] N. Sinha and C. Wang. Staged concurrent program analysis. In *FSE*, pages 47–56, 2010.

[25] N. Sinha and C. Wang. On interference abstractions. In *POPL*, pages 423–434, 2011.

[26] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *POPL*, pages 387–400, 2012.

[27] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *FSE*, pages 37–46, 2010.

[28] C. von Praun and T. R. Gross. Object race detection. *SIGPLAN Not.*, 36(11):70–82, 2001.

[29] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *TACAS*, pages 328–342, 2010.

[31] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146, 2006.

[32] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, 2006.

[33] J. Yi, C. Sadowski, and C. Flanagan. SideTrack: generalizing dynamic atomicity analysis. In *PADTAD*, pages 1–10, 2009.

[34] W. Zhang, J.Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.

[35] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.