# Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis

David Devecsery
University of Michigan
ddevec@umich.edu

Peter M. Chen
University of Michigan
pmchen@umich.edu

Jason Flinn
University of Michigan
jflinn@umich.edu

Satish Narayanasamy
University of Michigan
nsatish@umich.edu

***CCS Concepts*** • **Theory of computation** → *Program analysis*; • **Software and its engineering** → *Dynamic analysis*; *Software reliability*; *Software safety*; *Software testing and debugging*;

**ACM Reference Format:**
David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3177153

## Abstract

Dynamic analysis tools, such as those that detect data-races, verify memory safety, and identify information flow, have become a vital part of testing and debugging complex software systems. While these tools are powerful, their slow speed often limits how effectively they can be deployed in practice. Hybrid analysis speeds up these tools by using static analysis to decrease the work performed during dynamic analysis.

In this paper we argue that current hybrid analysis is needlessly hampered by an incorrect assumption that preserving the soundness of dynamic analysis requires an underlying sound static analysis. We observe that, even with unsound static analysis, it is possible to achieve sound dynamic analysis for the executions which fall within the set of states statically considered. This leads us to a new approach, called *optimistic hybrid analysis*. We first profile a small set of executions and generate a set of likely invariants that hold true during most, but not necessarily all, executions. Next, we apply a much more precise, but *unsound*, static analysis that assumes these invariants hold true. Finally, we run the resulting dynamic analysis speculatively while verifying whether the assumed invariants hold true during that particular execution; if not, the program is reexecuted with a traditional hybrid analysis.

Optimistic hybrid analysis is as precise and sound as traditional dynamic analysis, but is typically much faster because (1) unsound static analysis can speed up dynamic analysis much more than sound static analysis can and (2) verifications rarely fail. We apply optimistic hybrid analysis to race detection and program slicing and achieve 1.8x over a state-of-the-art race detector (FastTrack) optimized with traditional hybrid analysis and 8.3x over a hybrid backward slicer (Giri).


*ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA*

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3177153

## 1 Introduction

Dynamic analysis tools, such as those that detect data-races [23, 46], verify memory safety [41, 42], and identify information flow [16, 20, 31], have become a vital part of testing and debugging complex software systems. However, their substantial runtime overhead (often an order of magnitude or more) currently limits their effectiveness. This runtime overhead requires that substantial compute resources be used to support such analysis, and it hampers testing and debugging by requiring developers to wait longer for analysis results.

These costs are amplified at scale. Many uses of dynamic analysis are most effective when analyzing large and diverse sets of executions. For instance, nightly regression tests should run always-on analyses, such as data-race detection and memory safety checks, over large test suites. Debugging tools, such as slicing, have been shown to be more informative when combining multiple executions, e.g. when contrasting failing and successful executions [4, 25]. Forensic analyses often analyze weeks, months, or even years of computation [16, 31]. Any substantial reduction in dynamic analysis time makes these use cases cheaper to run and quicker to finish, so performance has been a major research focus in this area.

*Hybrid analysis* is a well-known method for speeding up dynamic analysis tools. This method statically analyzes the

program source code to prove properties about its execution. It uses these properties to prune some runtime checks during dynamic analysis [13, 14, 33, 41]. Conventionally, hybrid analysis requires sound [1] (no false negatives) static analysis, so as to guarantee that any removed checks do not compromise the accuracy of the subsequent dynamic analysis. However, soundness comes at a cost: a lack of precision (i.e., false positives) that substantially reduces the number of checks that can be removed and limits the performance improvement for dynamic analysis tools such as race detectors and slicers.

The key insight in this paper is that **hybrid analysis can benefit from carefully adding unsoundness to the static analysis, *and* preserve the soundness of the final dynamic analysis by executing the final dynamic analysis speculatively.** Allowing the static analysis to be unsound can improve its precision and scalability (Figure 1), allowing it to dramatically speed up dynamic analyses such as race detection (even after accounting for the extra cost of detecting and recovering from errors introduced by unsound static analysis).

*Optimistic hybrid analysis* is a hybrid analysis based on this insight. It combines unsound static analysis and speculative execution to create a dynamic analysis that is as precise and sound as traditional hybrid analysis, but is much faster. Optimistic hybrid analysis consists of three phases. First, it profiles a set of executions to derive optimistic assumptions about program behavior; we call these assumptions *likely invariants*. Second, it performs a static analysis that assumes these likely invariants hold true, we call this *predicated static analysis*. The assumptions enable a much more precise analysis, but require the runtime system to compensate when they are violated. Finally, it speculatively runs the target dynamic analysis, verifying that all likely invariants hold during the analyzed execution. If so, both predicated static analysis and the dynamic analysis are sound. In the rare case where verification fails, optimistic hybrid analysis rolls back and re-executes the program with a traditional hybrid analysis.

We demonstrate the effectiveness of optimistic hybrid analysis by applying it to two popular analyses on two different programming languages: OptFT, an optimistic hybrid data-race detection tool built on top of a state-of-the-art dynamic race detector (FastTrack) [23] for Java, and OptSlice, a optimistic hybrid backward slicer, built on the Giri dynamic slicer [45] for C. Our results show that OptFT provides speedups of 3.5x compared to FastTrack, and 1.8x compared to a hybrid-analysis-optimized version of FastTrack. Further, OptSlice analyzes complex programs for which Giri cannot run without exhausting computational resources, and it provides speedups of 8.3x over a hybrid-analysis-optimized version of Giri. We then show how predicated static analysis can improve foundational static analyses, such as points-to analysis, indicating that optimistic hybrid analysis techniques will benefit many more dynamic analyses.

[1] Following convention, we classify an analysis as sound even if it is only "soundy" [34]. For example, most "sound" static analysis tools ignore some difficult-to-model language features.

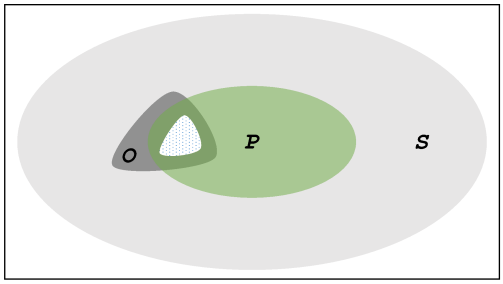


**Figure 1.** Sound static analysis not only considers all valid program states P, but due to sound over-approximation, it also considers a much larger S. Using likely invariants, predicated static analysis considers a much smaller set of program states O that are commonly reached (dotted space in P).

The primary contributions of this paper are as follows:

- We present optimistic hybrid analysis, a method of dramatically reducing runtimes of dynamic analysis without sacrificing soundness by first optimizing with a predicated static analysis and recovering from any potential unsoundness through speculative execution.
- We identify properties fundamental to selecting effective likely invariants, and we identify several effective likely invariants: unused call contexts, callee sets, unreachable code, guarding locks, singleton threads, and no custom synchronizations.
- We demonstrate the power of optimistic hybrid analysis by applying the technique to data-race detection and slicing analyses. We show optimistic hybrid analysis dramatically accelerates these analyses, without changing the results of the analysis. To the best of our knowledge, OptFT is currently the fastest dynamic happens-before data-race detector for Java that is sound.

## 2 Design

Optimistic hybrid analysis reduces the overhead of dynamic analyses by combining a new form of unsound analysis, known as predicated static analysis, with speculative execution. The use of speculative execution allows optimistic hybrid analysis to provide correct results, even when entering states not considered by predicated static analysis. A predicated static analysis assumes dynamically-gathered likely invariants hold true to reduce the state space it explores, creating a fundamentally more precise static analysis.

Figure 1 shows how the assumptions in a predicated static analysis can dramatically reduce the state space considered. A sound static analysis must make many overly-conservative

approximations that lead it to consider not just all possible executions of a program (P), but also many impossible executions (S).

Rather than paying the cost of this over-approximation, a hybrid analysis can instead construct a static analysis based only on the set of executions likely to actually be analyzed dynamically. Speculative assumptions make the state space (O) much smaller than not only S but also P, demonstrating that by using a predicated static analysis, optimistic hybrid analysis has the potential to optimize the common-case analysis more than even a perfect sound static analysis (whose results are bounded by P). The set of states in P not in O represent the set of states in which predicated static analysis is unsound. Optimistic hybrid analysis uses speculation and runtime support to handle when these states are encountered. As long as the set of states commonly experienced at runtime (denoted by the dotted area) resides in O, optimistic hybrid analysis rarely mis-speculates, resulting in an average runtime much faster than that of a traditional hybrid analysis.

We apply these principles using our three-phase analysis. First, we profile a set of executions of the target program and generate optimistic assumptions from these executions that might reduce the state space the static analysis needs to explore. As these dynamically gathered assumptions are not guaranteed to be true for all executions, we call them *likely invariants* of the executions.

Second, we use these likely invariants to perform a predicated static analysis on the program source. Leveraging the likely invariants allows this static analysis to be far more precise and scalable than traditional whole-program analysis, ultimately allowing it to better optimize dynamic analyses.

Finally, we construct and run the final dynamic analysis optimistically. Because predicated static analysis is not sound, we insert extra checks in this optimistic dynamic analysis to verify the likely invariants assumed hold true for each analyzed execution. If the checks determine that the likely invariants are in fact true for this execution, the execution will produce a sound, precise, and relatively efficient dynamic analysis. If the additional checks find that the invariants do not hold, the analysis needs to compensate for the unsoundness caused by predicated static analyses.

The rest of this section describes the three analysis steps, and important design considerations.

### 2.1 Likely Invariant Profiling

A predicated static analysis is more precise and scalable than traditional static analysis because it uses *likely invariants* to reduce the program states it considers. Likely invariants are learned though a dynamic profiling pass. We next discuss the desirable properties of a likely invariant, and how optimistic hybrid analysis learns the invariants by profiling executions.

**Strong:** By assuming the invariant, we should reduce the state space searched by predicated static analyses. This is the key property that enables invariants to help our static phase; if the invariant does not reduce the state space considered statically, the dynamic analyses will see no improvement.

**Cheap:** It should be inexpensive to check that a dynamic execution obeys the likely invariants. For soundness, the final dynamic analysis must check that each invariant holds during an analyzed execution. The cost of such checks increase the cost of the final dynamic analysis, so the net benefit of optimistic hybrid analysis is the time saved by eliding dynamic runtime checks minus the cost of checking the likely invariants. Note that the time spent in the profiling stage to gather likely invariants is done exactly once, and is therefore less important; only dynamically verifying the invariants needs to be inexpensive.

**Stable:** A likely invariant should hold true in most or all executions that will be analyzed dynamically. If not, the system will declare a mis-speculation, and recovering from such mis-speculations may be expensive for some analyses.

There is a trade-off between stability and strength of invariants. We find it sufficient to consider invariants that are true for all profiled executions. However, we could aggressively assume a property that is infrequently violated during profiling as a likely invariant. This stronger, but less stable invariant may result in significant reduction in dynamic checks, but increase the chance of invariant violations. If the reduced checks outweigh the costs of additional invariant violations this presents a beneficial trade-off.

### 2.2 Predicated Static Analysis

The second phase of optimistic hybrid analysis creates an unsound static analysis used to elide runtime checks and speed up the dynamic analysis. Traditional static analysis can elide some runtime checks. However, to ensure soundness, such static analysis conservatively analyzes not only all states that may be reached in an execution, but also many states that are not reachable in any legal execution. This conservative analysis harms both accuracy and scalability of static analysis.

A better approach would be for the static analysis to explore precisely the states that will be visited in dynamically analyzed executions. A predicated static analysis tries to achieve this goal by predicting these states through profiling and characterizing constraints on the states as likely invariants. By exploring only a constrained state space of the program (the states predicted reachable in future executions), predicated static analysis provides fundamentally more precise analysis.

This reduction of state space also improves the scalability of static analysis, which now need perform only a fraction of the computation a traditional static analysis would. Static analysis algorithms frequently trade-off accuracy for scalability [27, 35, 50, 53]. In some instances this improved efficiency allows the use of more sophisticated static analyses that are more precise but often fail to scale to large programs.

### 2.3 Dynamic Analysis

The final phase of optimistic hybrid analysis produces a sound, precise and relatively efficient dynamic analysis. Dynamic analysis is implemented by instrumenting a binary with additional checks that verify a property such as data-race freedom and then executing the instrumented binary to see if the verification succeeds.

In our work, the instrumentation differs from traditional dynamic analysis in two ways. First, we elide instrumentation for checks that static analysis has proven unnecessary; this is done by hybrid analysis also, but we elide more instrumentation due to our unsound static analysis. Second, we add checks that verify that all likely invariants hold true during the execution and violation-handling code that is executed when a verification fails.

To elide instrumentation, this phase consumes the set of unneeded runtime checks from the predicated static analysis phase. For instance, a data-race detector will instrument all read/write memory accesses and synchronization operations. The static analysis may prove that some of these read/write or synchronization operations cannot contribute to any races, allowing the instrumentation to be elided. Since the overhead of dynamic analysis is roughly proportional to the amount of instrumentation, eliding checks leads to a commensurate improvement in dynamic analysis runtime.

The instrumentation also inserts the likely invariant checks. By design, these invariants are cheap to check, so this code is generally low-overhead and simple. For example, checking likely unused code requires adding an invariant violation call at the beginning of each assumed-unused basic block. This call initiates rollback and re-execution if the check fails.

Roll-back is necessary as predicated static analysis may optimize away prior metadata updates needed for sound execution once an invariant is violated. Figure 2 shows how a metadata update for variable *a* on line 2 is elided by optimistic hybrid analysis because of the likely-unused code (LUC) invariant on line 4. If the invariant fails, then the metadata required to do the race check on line 5 is missing, and will be recovered by rolling-back and executing line 2 with conservative analysis.

We currently handle invariant violations with a catch-all approach: roll-back the entire execution and re-analyze it with traditional (non-optimistic) hybrid analysis. As we target retroactive analysis, this approach is practical for several reasons. First, with sufficient profiling invariant violations will be rare enough that even this simple approach has minimal impact on overall analysis time. Second, restarting a deterministic replay, and guaranteeing equivalent execution is trivial with record/replay systems, which are commonly used in retroactive analyses. If the cost of rollback became an issue or full record/replay systems were impractical, we could reduce the costs of rollbacks through more profiling or explore cheaper rollback mechanisms, such as partial roll-back or partial re-analysis.

One appealing approach to reducing the cost of invariant mis-speculation is to recover by rolling back to a predicated static analysis analysis that doesn't assume the invariant just violated. However, doing so generally would require an analysis for each possible set of invariant violations ($O(2^n)$ where n is number of invariants), far too many static analyses to reasonably run. It may be possible to reduce this number by grouping invariants, but since we do not experience significant slowdowns with our sound analysis recovery method, we do not explore this approach further.

## 3 Static Analysis Background

OptFT and OptSlice are built using several *data-flow* analyses, such as backward slicing, points-to, and may-happen-in-parallel. Data-flow analysis approximate how some property propagates though a program. To construct this approximation, a data-flow analysis builds a conservative model of information flow through the program, usually using a definition-use graph (DUG). The DUG is a directed graph that creates a node per definition (def) analyzed. For example, a slicing DUG would have a node per instruction, while a points-to analysis would have nodes for pointer definitions. Edges represent information flow in the program between defs and the defs defined by uses. For example, an assignment operation in a points-to analysis creates an edge from the node representing the assignment's source operand to the node representing its destination. Once the DUG is constructed, the analysis propagates information through the graph until a closure is reached. To create optimistic versions of these data-flow analyses, we leverage likely invariants to reduce the number of paths through which information flows in the DUG.

There are many modeling decisions that an analysis makes when constructing the DUG. One critical choice is that of context-sensitivity. A call-site context-sensitive analysis logically distinguishes different call-stacks, allowing more precise analysis. A context-insensitive analysis tracks information flow between function calls, but does not distinguish between different invocations of the same function.

Logically, a context-insensitive analysis simplifies and approximates a program by assuming a function will always behave the same way, irrespective of calling context. To create this abstraction, context-insensitive analyses construct what we call local DUGs for each function by analyzing the function independently and creating a single set of nodes in the graph per function. The analysis DUG is then constructed by connecting the nodes of the local DUGs at inter-function communication points (e.g. calls and returns).

A context-sensitive analysis differs from a context-insensitive analysis by distinguishing all possible calling contexts of all functions, even those which will never likely occur in

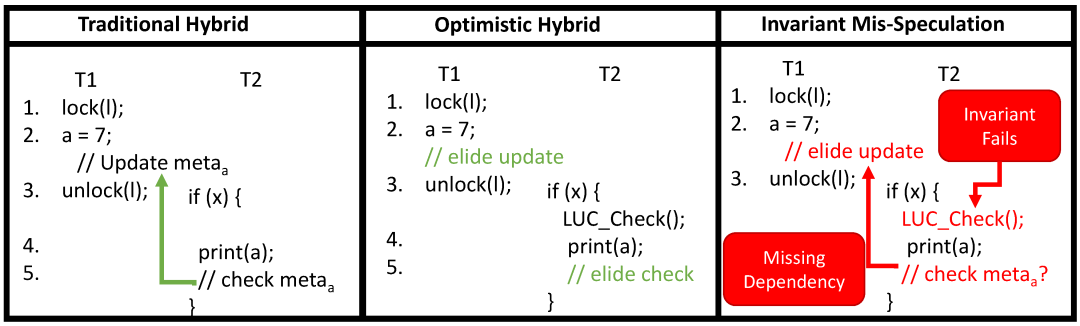


**Figure 2.** Example of how OptFT can require rollback on invariant violation. When the likely-unused code (LUC) invariant is violated on rollback, the execution must rollback and re-execute line 2 to gather the metadata required for the check on line 5.

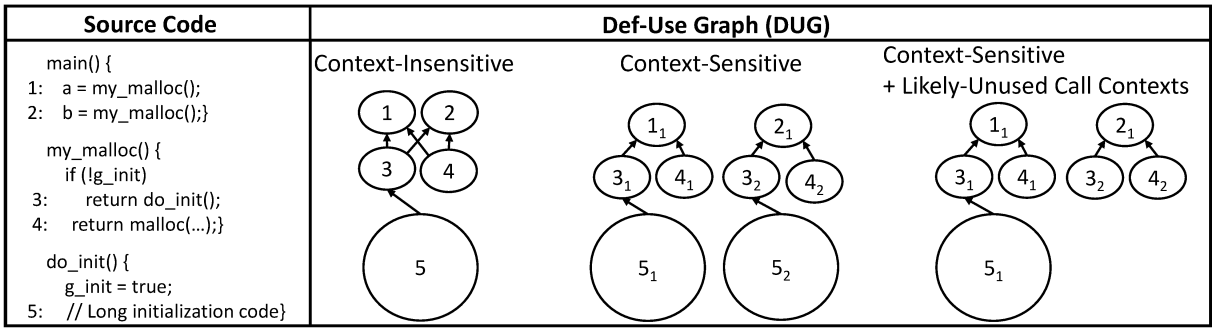


**Figure 3.** Demonstration of how context-sensitive and context-insensitive analysis parse a code segment to construct a DUG, as well as the reductions from likely-unused call contexts

practice. To create this abstraction, the DUG of the analysis replicates the nodes defined by a function each time a new calling-context is discovered during the DUG construction. One simple method of creating such a DUG is through what is known as a bottom-up construction phase, in which the analysis begins at main, and for each call in main it creates a clone of the nodes and edges of the local DUG for the callee function. It then connects the arguments and return values to the call-site being processed. If that callee function has any call-sites, the callee is then processed in the same bottom-up manner. This recurses until all callees have been processed, resulting in a context-sensitive DUG representing the program. The context-sensitive expression of the DUG is much larger than that of a context-insensitive analysis, but it also allows for more precise analysis.

Figure 3 illustrates the differences between DUGs constructed by a context-sensitive and insensitive analysis. Nodes 3, 4, and 5 are replicated for each call to *my_malloc*(), allowing the analysis to distinguish between the different call-contexts, but replicating the large *do_init*() function.

Context-sensitive analyses tend to be precise, but not fully scalable, while context-insensitive analyses are more scalable at the cost of accuracy. We build both context-sensitive and insensitive variants of several predicated static analyses.

# 4 OptFT

To show the effectiveness of optimistic hybrid analysis, we design and implement two sample analyses: OptFT, an optimistic variant of the FastTrack race detector for Java, and OptSlice, an optimistic dynamic slicer for C programs. This section describes OptFT and Section 5 describes OptSlice.

OptFT is a dynamic data-race detection tool that provides results equivalent to the FastTrack race detector [23]. FastTrack instruments load, store, and synchronization operations to keep vector clocks tracking the ordering among memory operations. These vector clocks are used to identify unordered read and write operations, or data-races.

OptFT uses the Chord analysis framework for static analysis and profiling, building on Chord's default context-insensitive static data-race detector [40]. For dynamic analysis we use the RoadRunner [24] analysis framework, optimizing their default FastTrack implementation [23].

## 4.1 Analysis Overview

The Chord static data-race detector is a context-insensitive, lockset-based detector. The analysis depends on two fundamental data-flow analyses, a may-happen-in-parallel (MHP) analysis, which determines if memory accesses may happen in parallel, and a points-to analysis, which identifies the

memory locations to which each pointer in the program may point.

The analysis first runs its static MHP analysis to determine which sets of loads and stores could dynamically happen in parallel. Once those sets are known, the analysis combines this information with a points-to analysis to construct pairs of potentially racy memory accesses which may alias and happen in parallel. Finally, the analysis uses its points-to analysis to identify the lockset guarding each memory access, and it uses these to exclude pairs of loads and stores guarded by the same lock from its set of potentially racing accesses.

To optimize the dynamic analysis, OptFT elides instrumentation around any loads or stores that predicated static analysis identifies as not racing. The analysis also elides instrumentation around some lock/unlock operations, as we discuss in Section 4.2.4.

### 4.2 Invariants

OptFT is optimized with four likely invariants. OptFT first gathers the invaraints with a set of per-invariant profiling passes, and stores the invariant set for each profiling execution in a text file. This text file maps invariant sites to sets of invariant data (e.g. a basic block to how many times its visited, or an indirect callsite to the functions it may call). Then, after all profiles are run, the individual run's invariant sets are merged, (by intersecting the sets of invariants, to find invariants that hold true for all runs) to gather the invariant set for all of the profiling experiments. The individual invariants gathered and used by OptFT are:

#### 4.2.1 Likely Unreachable Code

The first, and simplest, invariant OptFT assumes is likely-unreachable code. We define a basic block within the program that is unlikely to be visited in an execution as a *likely unreachable code (LUC)* block. To profile LUC, OptFT profiles the inverse, that is used basic blocks. OptFT runs a basic block counting profiling pass, which instruments each basic block to create a count of times it was visited. OptFT uses this information to create a mapping of basic blocks to execution counts. The inverse of profiled blocks (set of basic blocks not in our visited basic block set) is our likely unvisited set.

This invariant easily satisfies the three criteria of good likely invariants. First, it is strong; the invariant reduces the state space our data-flow analyses considers by pruning nodes defined by likely unused code and any edges incident upon them from our analysis DUGs. This reduction in connectivity within the DUG can greatly reduce the amount information that propagates within the analysis. Second, the invariant is virtually free to check at runtime, requiring only a mis-speculation call at the beginning of the likely-unused code. Finally, we observe that unused code blocks are typically stable across executions.

#### 4.2.2 Likely Guarding Locks

Chord's race detector's final phase prunes potentially racy accesses by identifying aliasing locksets. Unfortunately, this optimization is unsound. To soundly identify if two locksites guard a load or store, Chord needs to prove that the two sites *must* hold the same lock when executing. However, the alias analysis Chord uses only identifies *may* alias relations. To get sound results from Chord we must either forego this lock-based pruning or use a (typically unscalable and inaccurate) *must* analysis. In the past, hybrid analyses that use Chord have opted to remove this pruning phase for soundness [47].

Likely guarding locks attempt to overcome Chord's may-alias lockset issue by dynamically identifying must-alias lock pairs. The profiling pass instruments each lock site and tracks the objects locked, creating a set of dynamic objects locked at each lock site. If it identifies that two sites always only lock the same dynamic object, it assumes a must-alias invariant for the lock pairs. The output of this profiling execution is a set of these "must-alias" relations, these pairs can then be directly consumed by chord's lockset pruning pass.

The invariant is strong. By assuming the invariant, the Chord race detection algorithm can add in some of the lockset-based pruning discarded due to its weaker *may* alias analysis. Additionally, the invariant is cheap to check at runtime. The dynamic analysis need only instrument the assumed aliasing lock-sites and verify the sites are locking the same object, a check far less expensive than the lock operation itself. Finally, executions do not vary the objects locked frequently, so this invariant remains stable across executions.

#### 4.2.3 Likely Singleton Thread

Likely singleton thread invariants aid Chord's MHP analysis. If a thread start location creates only a single instance of a thread, all memory accesses within that thread are ordered. If the start location spawns multiple threads (e.g. its executed within a loop), then the memory accesses in different threads associated with that start location may race. We call this single-thread start call a *singleton-thread* instance.

The knowledge of singleton-thread instances is easy to gather dynamically by monitoring thread start sites. On the other hand, statically reasoning about this information is hard, requiring understanding of complex program properties such as loop bounds, reflection, and even possible user inputs. The likely singleton thread invariant eliminates the need for this static reasoning by dynamically identifying singleton-thread sites. When profiling for this invariant, OptFT instruments each thread creation site, identifying and outputting the set of threads created exactly once. This set of singleton-threads allows the static MHP analysis to prune many memory access pairs for singleton thread instances that it would otherwise miss.

The invariant easily meets the properties of a good likely invariant. First, the invariant can greatly aid the MHP analy-

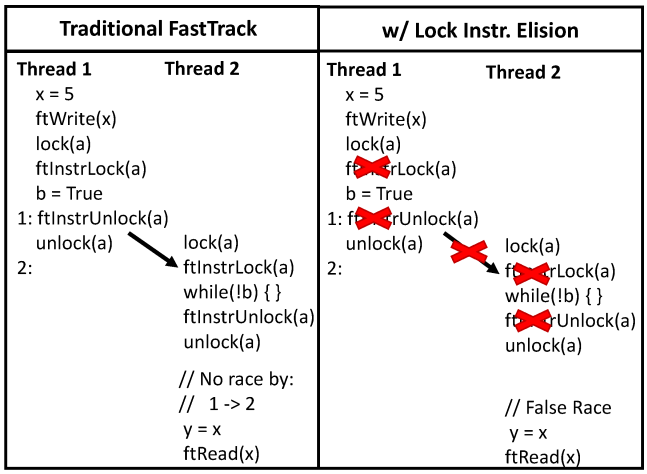


**Figure 4.** An example of how lock instrumentation elision may cause missed happens-before relations in the presence of custom synchronizations. The left hand side catches custom synchronizations, but with the elision of locking instrumentation, the necessary happens before relation (represented by an arrow) may be lost.

sis, which is foundational to our race detector. Second, the invariant is inexpensive to check, only requiring monitoring of predicted singleton thread start locations. Finally, the invariant is generally stable across runs.

#### 4.2.4 No Custom Synchronizations

Ideally, static analysis would enable OptFT to elide instrumentation for lock and unlock operations. However, the possibility of custom synchronizations stops a sound analysis from enabling this optimization. Figure 4 shows how eliding lock/unlock instrumentation, even when there are no racy accesses within the critical section, can cause a false race report. This problem is caused by custom synchronization (e.g. waiting on $b$ in Figure 4).

To enable elision of lock and unlock instrumentation, we propose the no custom synchronization invariant. Using this invariant, OptFT optimistically elides instrumentation around lock/unlock operations whose critical sections do not contain any read or write checks. To profile this invariant, we run the dynamic detector with lock/unlock operations not guarding any dynamic read/write checks elided. If this elision causes the dynamic race detector to report false races (false races are detected by comparing the output with that of a sound race detector), we know that an elided lock is guarding a custom synchronization. If so, we return the lock/unlock instrumentation to the offending locks until the false races are removed.

The drawback to this approach is that race reports must be considered as potential mis-speculations. This could be an undue burden if analysis frequently reports data-races; however, if a program has frequent data-races, there is little need for a highly optimized race detector.

This invariant is highly useful. First, it helps the static analysis eliminate work by reducing the instrumentation around locks. Second, it is easy to check, our race detector already detects races. Finally, custom synchronizations rarely change between executions, so it is stable.

## 5 OptSlice

OptSlice is our optimistic dynamic backward slicing tool. A backward slice is the set of program statements that may affect a target (or slice point) Program slices are important debugging tools, as they simplify complex programs and help developers locate the sources (i.e., root causes) of errors more easily. Backward slicing is particularly powerful when analyzing multiple executions to find differences between failing and non-failing executions [4, 25].

OptSlice optimizes the Giri dynamic slicer [45] with an optimistic variant of Weiser's classic static slicing algorithm [52]. OptSlice collects data-flow slices. Data-flow slices do not consider control dependencies and are often used when control dependencies cause a slicer to output so much information the slice is no longer useful.

### 5.1 Static Analysis

OptSlice uses a backward slicing analysis that builds on the results of a separate points-to analysis; we next describe these two analyses.

#### 5.1.1 Backward Slicing

The static slicer used by OptSlice first constructs a DUG of the program. We have implemented two versions of this algorithm: a context-sensitive and a context-insensitive variant. The DUG contains a node for every instruction in the program and edges representing the reverse information flow through the program (i.e., from any defs which use instructions to the defs providing those uses). The slicing analysis resolves indirect def-use edges (e.g., loads and stores) by using a points-to analysis to determine aliases. As slicing is a flow-sensitive analysis, when resolving these indirect edges, the slicer only considers stores in basic blocks that may precede the load being analyzed according to the program's control-flow graph.

Once the DUG is constructed, our static analysis computes the conservative slice by calculating the closure of the graph, starting from any user-defined slice endpoints. The final slice is composed of any instructions whose defs are represented by the nodes within this closure.

Our optimistic backward slicer implements several optimizations. First, it lazily constructs the DUG, creating nodes only when required. Second, it uses binary decision diagrams (BDDs) [9] to keep track of the visited node set. This is similar to how BDDs are used to track points-to sets [6].

#### 5.1.2 Points-To

Our Andersen's-based points-to analysis [5] constructs a DUG with a node for each statement in the program that defines a pointer. Edges represent the information flow by pointer uses. Unlike the slicing DUG, not all nodes and edges can be resolved at graph creation time. These nodes and edges are dynamically added as points-to sets are discovered during the next analysis phase.

After constructing the DUG, the analysis associates an empty points-to set with each node and initializes the points-to sets of any nodes which define a pointer location (e.g. malloc calls). The algorithm then propagates points-to information along the edges defined by the graph. Additionally, the algorithm may add edges to the graph as indirect def-use pairs are discovered; e.g., if a load-source and store-destination are found to alias, the analysis makes an edge between the two nodes. After all information has finished propagating through the DUG, each node has its conservative set of potential pointers.

Indirect function calls are handled in a special manner. For context-insensitive analyses, a pointer used in an indirect function call is resolved, the arguments and return values are connected to the existing nodes for the resolved function(s) in the graph. Context-sensitive analyses, however, require distinct information pathways for different static call stacks. In a context-sensitive analysis, nodes may have to be added to the graph. When an indirect function call is resolved in a context-sensitive analysis, the analysis scans the call stack to check for recursive calls. If the new callee creates a recursive call, the call is connected to the prior nodes in the DUG representing that callee. If the function call is not recursive, a new set of nodes must be added for the call in the same manner as for the bottom-up DUG construction phase.

The analysis is complete once the graph reaches transitive closure. Each def's points-to set is the union of the points-to sets of all nodes representing that def.

Our algorithm uses heap cloning and is structure-field sensitive. We also use several well-known optimizations, including offline graph optimizations (HVN/HRU) [30], and cycle detection (LCD and HCD) [29], and BDDs to track points-to sets [6]. These optimizations contribute to the scalability and accuracy of the analysis, but they do not impact how we apply likely invariants, so we do not discuss them further.

### 5.2 Invariants

OptSlice uses several general invariants, aimed at increasing overall analysis accuracy. After the invariants are profiled, we use them to reduce the set of states our static analysis considers. As with OptFT, OptSlice first gathers the invariants by profiling individual executions, storing them in a text file, and then intersects or unions the sets (depending on the invariant) of invariants together to gather its final set. Below, we discuss how each invariant affects DUG construction.

#### 5.2.1 Likely Unreachable Code

OptSlice uses likely unreachable code identically to OptFT.

#### 5.2.2 Likely Callee Sets

Our points-to analysis's indirect function call resolution process can lead to considerable slowdowns, increased memory consumption, and analysis inaccuracies. If the analysis is unable to resolve the destination pointer of an indirect function call, it may have to conservatively assume that the callee may be *any* function in the program, connecting the call-site arguments of this function to all functions. On its own, this is a major source of inaccuracy. It also can lead to propagating inaccuracies if a function argument is used later as an indirect call. This issue is compounded in context-sensitive analyses, where the nodes in the local DUGs for all functions are replicated, dramatically increasing the analysis time. This problem is particularly impactful in programs like Perl (Perl is an interpreter that has a generic variable structure type that holds all types of variables, including ints, floats, structures, and function pointers).

Likely callee sets are the dynamically-gathered likely destinations of indirect function calls. The profiling pass instruments each indirect callsite, and identifies and maintains the mapping from callsite to dynamically observed callee destination. This invariant helps resolve many of the inaccuracies and inefficiencies that unknown indirect calls can add to our points-to analysis. Because likely callee-sets is a reachable invariant (as opposed to unreachable invariants unreachable code, and unused call contexts), individual profile run's results are unioned together instead of intersected.

This invariant converts all indirect calls in the DUG to direct calls to the assumed callee functions. The invariant is relatively inexpensive to check at runtime, requiring only a relatively small (usually singleton) set inclusion check on a function pointer update (a relatively rare operation). Most indirect function calls have very small sets of destinations, and they don't vary from execution to execution, making this invariant stable across executions.

#### 5.2.3 Likely Unused Call Contexts

Context-sensitive analyses can suffer significant scalability problems due to excessive local DUG cloning, as discussed in Section 3. Likely callee set invariants minimize local DUG cloning by stopping the context-sensitive analysis from cloning DUGs for call contexts, or call stacks, that are unlikely to occur. This invariant is profiled by logically constructing the call stack for each thread. The profiling pass instruments each callsite and appends the destination to a thread-local "call-stack". If the newly created callstack is unique, it is added to a set of observed callstacks. Once the profile has completed, the set of all observed callstacks is

written out. This caller stack is then used by the context sensitive analysis to limit local DUG cloning around unrealized call chains. This effect is demonstrated in Figure 3, removing the likely-unrealized second call to *do_init*().

Likely unused call contexts meet two of the criteria for good likely invariants. First, they are strong, as they can dramatically reduce the size of the DUG and the amount of space the data-flow analysis explores. Second, the invariant tends to be stable across executions.

Cheap checking of likely unused call contexts is a slightly more complex matter. Logically, the check needs to ensure that unused call contexts are never reached, requiring a call-stack set inclusion check at many call-sites. We found that a naive implementation of this functionality was too inefficient for some programs. To accelerate it, we use a Bloom filter to elide the majority of our relatively expensive set inclusion tests. We find that this methodology makes the dynamic cost of the invariant check acceptable.

# 6 Evaluation

In this section, we show that optimistic hybrid analysis can dramatically accelerate dynamic analyses by evaluating our two sample analyses, OptFT and OptSlice, over a wide range of applications.

## 6.1 Experimental Setup

### 6.1.1 OptFT

We evaluate the effectiveness of OptFT on the Dacapo [7] and JavaGrande [48] benchmark suites. Our test suite is composed of all multi-threaded benchmarks from these suites which are compatible with our underlying Chord [40], and RoadRunner [24] frameworks.

Optimistic hybrid analysis requires considerable profiling, more than the single profile set provided by these benchmark suites. To test these applications we construct large profiling and testing sets for each benchmark. For several benchmarks we use large, readily-available input sets:

- **lusearch** – Search novels from Project Gutenberg [2].
- **pmd** – Run the pmd source code analysis tool across source files in our benchmarks.
- **montecarlo** – Analyze 10 years of S&P 500 stock data.
- **batik** – Render svg files from svgcuts.com [3].
- **xalan** – Convert xhtml versions of pydoc 2.7 Web pages to XSL-FO files.
- **luindex** – Index novels from Project Gutenberg [2].

The remainder of our benchmarks (sunflow, raytracer, sor, moldyn, lufact, crypt, series, and sparse) benchmarks do not have large, freely available input sets, so we generated large sets by sweeping parameters across the input sets (e.g. input size, number of threads, pseudo-random seed).

To profile OptFT, we generate two sets of 64 inputs for each test. One set is our candidate profiling runs; the other is our testing corpus.

We run OptFT as a programmer would on a large set of regression tests. We first profile increasing numbers of profiling executions, until the number of learned dynamic invariants stabilize. Then, we run OptFT over all tests in our testing set. We run all data race detection experiments using 8 cores of an Intel Xeon E5-2687W v3 3.1 GHz processor.

### 6.1.2 OptSlice

OptSlice is implemented in the LLVM-3.1 compiler infrastructure. We accelerate the Giri dynamic backward slicer [45]. We evaluate the effectiveness of our analysis over a suite of common desktop and server applications.

Our test suite workloads consist of:

- **nginx** – Serve a clone of the pydoc 2.7 documentation, and load random pages.
- **redis** – Redis-benchmark application with varying data-size, client, and number of requests.
- **perl** – The SPEC2006 diffmail application with different inbox configurations.
- **vim** – Problem solutions from vimgolf.com [1].
- **sphinx** – Process a large database of voice samples.
- **go** – Predict the next best move from random points in an archive of professional go games.
- **zlib** – Compress files with a gzip-like utility. Input files are sampled from our sphinx input files.

For perl we analyze the SPEC2006 diffmail application. This represents the scenario of optimizing the interpreter to run a repeated analysis over a single script (e.g. running a single web content-generation script repeatedly on different requests), not the scenario of a single interpreter being run over many different perl programs.

Much as we did for OptFT, we generate profile and testing sets (512 files each for redis, zlib, sphinx, perl, and nginx; 2048 for go and vim). We profile each program, growing the input set until the number of dynamic invariants stabilizes. We then test on our testing set of inputs. This methodology is consistent with how we imagine OptSlice may be used for debugging, such as when comparing good and failing executions of a program.

We select static slices from several random locations within our benchmarks, using the most accurate static analysis that will complete on that benchmark without exhausting available computational resources.

Once we have gathered our set of slices, we generate dynamic slicing instrumentation. To determine statistical significance between good and bad runs of a program, a developer would start at a suspect instruction and calculate the backward slice over many executions (both failing and successful). We therefore select non-trivial endpoints for calculating such slices and calculate the slice from each endpoint for each

execution in the testing set. We define a non-trivial endpoint to be an instruction with a sound static slice containing at least 500 instructions. We use non-trivial endpoints because they tend to be far more time consuming to compute slices (there is little use optimizing a trivial analysis), and they are common; on average, 55% of the endpoints from our sound static slicer are non-trivial.

We slice each endpoint with the most accurate predicated static slicer that will run on that program. Once we have our predicated static slices, we optimize our dynamic instrumentation, and dynamically slice all tests in our testing set with our dynamic slicer. We repeat this process until we analyze five different program endpoints; this provides a sufficient set of endpoints to gain confidence in OptSlice's ability to optimize slicing for a general program point.

All experiments are run using a single core of our Xeon processor, and each is allocated 16 GB of RAM. Table 2 gives an overview of our benchmarks, their relative sizes, static analysis times, and which static analysis we use.

### 6.2 Dynamic Overhead Reduction

Figure 5 shows how optimistic hybrid analysis improves performance for race detection. Although we show all benchmarks for completeness, 5 benchmarks (those to the right of the vertical line in Figure 5) are quite simple and can be statically proven to be race-free. Thus, there is no need for any dynamic race analysis in these cases. For the remaining 9 benchmarks, OptFT shows average speedups of 3.5x versus traditional FastTrack, and 1.8x versus hybrid FastTrack. Impressively, for many of the benchmarks analyzed, the costs of OptFT approach those of the underlying RoadRunner instrumentation framework; this presents a near-optimal reduction in work due to the OptFT algorithm.

There are two remaining benchmarks for which OptFT sees limited speedups: sunflow and montecarlo. These benchmarks both make considerable use of fork-join and barrier based parallelism. Consequently, the lockset based Chord detector is algorithmically unequipped to optimize their memory operations, even with optimistic invariants. A static analysis algorithm better equipped to deal with barrier based parallelism would likely see more profound speedups from optimistic hybrid analysis.

Figure 5 additionally shows that the invariant checking and mis-speculation overheads associated with OptFT are negligible for nearly all benchmarks. Overall, invariant checking overheads have little effect on the runtime of our race detector, averaging 4.3% relative to a baseline execution. Additionally roll-backs are infrequent and cause little overhead, ranging from 0.0% to 21.9% and averaging 5.7%.

Figure 6 shows the online overheads for OptSlice versus a traditional hybrid slicer. We do not compare to purely dynamic Giri, as it exhausts system resources even on modest executions. OptSlice dramatically reduces the runtime of dynamic slicing, with speedups ranging from 1.2x to 78.5x, with an average speedup of 8.3x. Our worst absolute speedups are from perl and nginx. Perl's state is divided largely into two subsets, the interpreter state and the script state. Without knowledge of the script running in the interpreter, static analysis cannot precisely determine how information flows through the script state. Perl scripts would be better analyzed at the script level. Nginx is largely I/O bound, but OptSlice decreases its overhead from 20% to a statistically insignificant overhead. This reduction is relatively significant, even though it is not absolutely large.

We also look at the invariant-checking and mis-speculation costs of OptSlice. The overheads of ensuring executions do not violate likely-invariants are generally inconsequential, showing no measurable overhead for zlib, go, nginx, and vim. Perl and sphinx have overheads of 26% and 127% respectively, largely due to likely-unrealized call-context checking. These overheads are low enough for optimistic hybrid analysis to improve slicing performance, but could be optimized further if lower overheads are needed [8]. Overall mis-speculation rates are low for all benchmarks, with go and vim being the only benchmarks to see even modest overheads.

So far, we have looked at speedups of using optimistic hybrid analysis when the profiling and static analysis costs are inconsequential. This is typical when static analysis can be done offline (e.g., after code is released but before the first bug is reported), or for very large analysis bases, such as analyzing months or years of prior executions in forensic queries. We next look at how much execution time must be analyzed for the dynamic savings of an optimistic hybrid analysis to overcome its analysis and profiling startup costs for smaller execution bases, which might occur when running nightly regression tests or when using delta debugging for moderate sets of inputs immediately after recompilation.

Table 1 shows break-even times for benchmarks not statically proven race-free. OptFT begins to out-perform both traditional and hybrid FastTrack within a few minutes of test time for most benchmarks. There are exceptions, such as montecarlo, sunflow, batik, and xalan, for which OptFT does not speed up dynamic analysis and therefore should not be used.

OptSlice shows a similar breakdown in Table 2, which compares OptSlice to a traditional hybrid slicer. This chart shows similar static analysis and profiling times as OptFT; however, due to the both the larger dynamic speedup of OptSlice and the reduction in static analysis state from the likely invariants, the break-even times are generally much lower. In three cases (vim, redis, and nginx), it is on average better to run a hybrid slicer when analyzing *any* execution length. In all cases, with under 3 minutes of execution time analyzed, OptSlice saves work versus traditional hybrid analysis.

We now analyze how profiling effects the trade-off between accuracy and correctness of predicated static analysis for a given execution. Figures 7 and 8 show the trade-offs between profiling, mis-speculation rate and static slice size

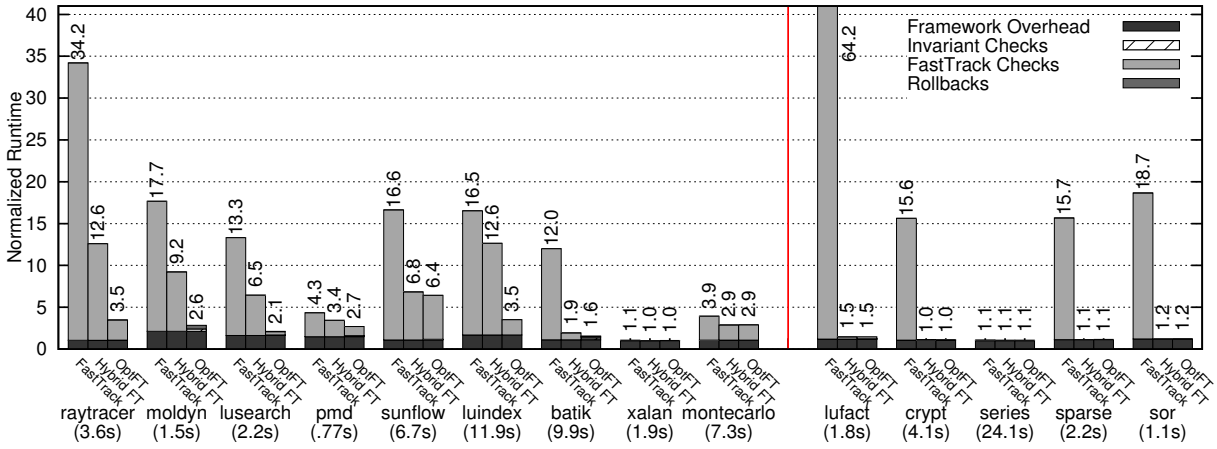


**Figure 5.** Normalized runtimes for OptFT. Baseline runtimes for each benchmark are shown in parentheses. Tests right of the red line are proven race-free by sound static race detection, but included here for completeness.

| Testname | Trad. Hybrid | Opt. Hybrid | | Break-even w/ respect to | | Opt. Speedup w/ respect to | |
|---|---|---|---|---|---|---|---|
| | Static Time | Profile | Static Time | Hybrid FT | Trad. FT | Hybrid FT | Trad. FT |
| lusearch | 1m 15s | 1m 12s | 1m 47s | 24s | 16s | 3.0x | 6.3x |
| pmd | 1m 6s | 20s | 2m 17s | 2m 0s | 1m 34s | 1.3x | 1.6x |
| raytracer | 31s | 14m 52s | 49s | 1m 39s | 30s | 3.6x | 9.8x |
| moldyn | 29s | 51m 46s | 49s | 7m 53s | 3m 29s | 3.5x | 6.7x |
| sunflow | 3m 0s | 22m 40s | 4m 10s | 58m 55s | 2m 37s | 1.1x | 2.6x |
| montecarlo | 59s | 1m 36s | 51s | – | 2m 25s | 0.99x | 1.3x |
| batik | 3m 25s | 15m 15s | 10m 9s | 60m 57s | 2m 26s | 1.2x | 7.6x |
| xalan | 55s | 51s | 1m 26s | 363m 44s | 60m 23s | 1.0x | 1.0x |
| luindex | 1m 7s | 17m 22s | 1m 57s | 1m 59s | 1m 28s | 3.6x | 4.8x |

**Table 1.** Comparing FastTrack benchmark end-to-end analysis times for pure dynamic as well as traditional and optimistic hybrid analyses. Break-even Time is the amount of baseline execution time at which optimistic analysis begins to use less computational resources (profiling + static + dynamic) than a traditional analysis. Optimistic Speedup is the ratio of runtimes for OptFT versus a traditional or hybrid FastTrack implementation.

| Testname (LOC) | Traditional | | | | Optimistic | | | | | Break-even Time | Dynamic Speedup |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Points-to | | Slice | | Profiling Time | Points-to | | Slice | | | |
| | AT | Time | AT | Time | | AT | Time | AT | Time | | |
| nginx (119K) | CI | 17s | CI | 24m 33s | 1m 4s | CS | 8s | CS | 3s | 0s | 1.2x |
| redis (80K) | CI | 1m 46s | CI | 170m 46s | 1m 4s | CI | 6s | CS | 48s | 0s | 13.1x |
| perl (128K) | CI | 24s | CS | 55m 0s | 10m 29s | CS | 160m 33s | CS | 9m 11s | 2m 29s | 1.4x |
| vim (306K) | CI | 27s | CI | 77m 55s | 11m 8s | CS | 1m 20s | CS | 21s | 0s | 9.9x |
| sphinx (13K) | CS | 7s | CS | 1s | 11m 24s | CS | 6s | CS | 0.2s | 1m 44s | 3.9x |
| go (158K) | CI | 6s | CI | 59s | 133m 54s | CI | 8s | CI | 9s | 1m 41s | 6.5x |
| zlib (21K) | CS | 14s | CS | 33s | 1m 59s | CS | 5s | CS | 0.4s | 1s | 81.2x |

**Table 2.** Comparing slicing benchmark end-to-end analysis times for traditional hybrid and optimistic hybrid analyses. Shown are a breakdown of offline analysis costs for static points-to and slicing analyses and the most accurate Analysis Type (AT), either Context-Sensitive (CS) or Context-Insensitive (CI) that will run on a given benchmark. Break-even Time is the minimum amount of baseline execution time where an optimistic analysis uses less total computational resources (profiling + static + dynamic) than a traditional hybrid analysis. Dynamic Speedup is the ratio of run-times for OptSlice versus traditional hybrid.

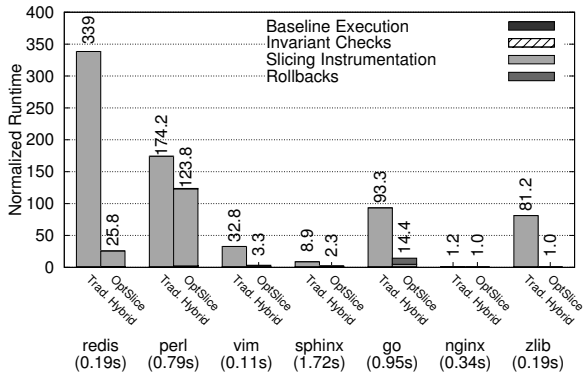


**Figure 6.** Normalized runtimes for OptSlice. Baseline runtimes for each benchmark are shown in parentheses.

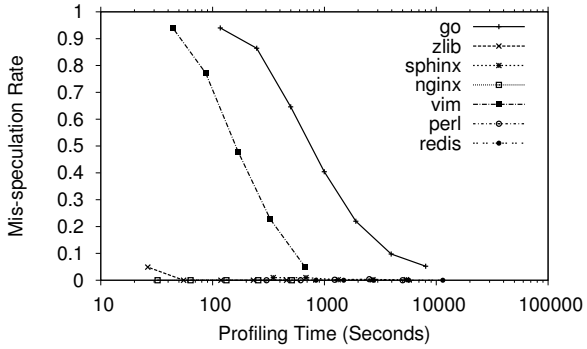


**Figure 7.** The effect of profiling time has on mis-speculation rates for OptSlice benchmarks.

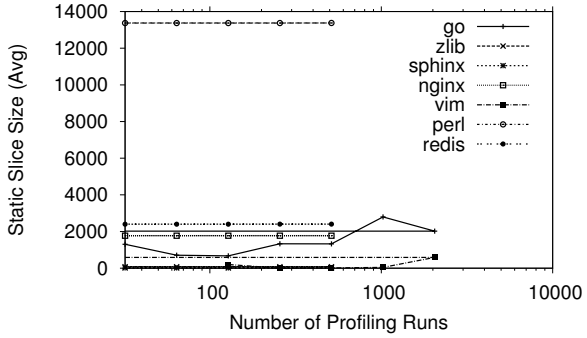


**Figure 8.** The effect of profiling on static slice sizes.

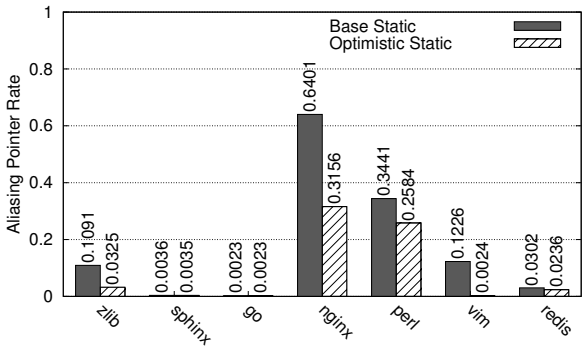


**Figure 9.** Alias rates for points-to analyses, reported as a chance that a store may alias with a load.

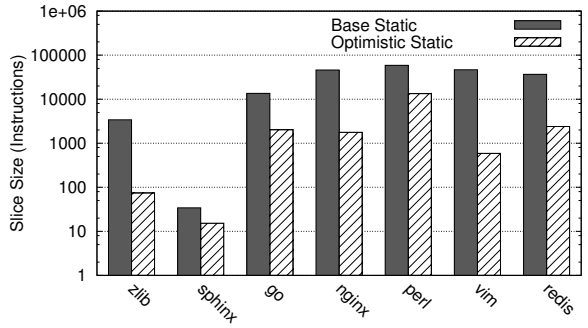


**Figure 10.** Static slice sizes, in number of instructions, as reported by a sound and a predicated static slicer.

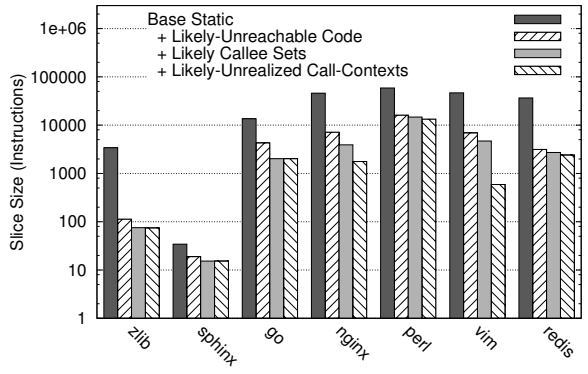


**Figure 11.** The effect of different likely invariants on slice size. Vim and nginx begin using context-sensitive analyses when adding likely-unrealized call-contexts.

for OptSlice. Figure 7 shows that most benchmarks converge to a nearly 0% mis-speculation rate very quickly, with the exceptions being vim and go, which explore very large states and consequently require more profiling. Figure 8 shows that for most applications slice size remains consistent, even as more profiling samples are added. The major exception to this is go, which explores a very large state-space resulting in different slice sizes as more profiling input is added. Not all experiments show monotonically increasing slice sizes. This is caused by the variations between profiling runs of the experiments. Go is particularly notable for this, as it uses timeouts to bound its execution time, resulting in inconsistent code paths and profiled invariant sets.

### 6.3 Predicated Static Analysis

We next evaluate the effects of predicated static analysis on our C-based points-to and slicing analyses. Both are general-purpose analyses with many applications. In fact, alias anal-

ysis is foundational to most complex static analyses; any improvement to it will have wide ranging effect on the many analyses that depend on it.

Figure 9 shows how a predicated static analysis significantly increases the accuracy of an alias analysis. Alias rates are measured as the probability that any given load can alias with any given store. For fairness, both baseline and optimistic analyses consider only the set of loads and stores present in the optimistic analysis (this is a subset of the baseline set due to state reduction caused by likely invariants). Figure 10 shows the reduction in overall slice sizes, with optimistic analysis providing one to two orders of magnitude in slice reduction.

We next break down how the likely invariants individually benefit static analyses. Figure 11 measures static slice size when running a sound static analysis and incrementally adds each likely invariant for three tests: vim, nginx, and zlib. The introduction of the likely-unrealized call-context invariant allows vim and nginx to scale to context-sensitive slicing and points-to analysis, causing a large reduction in slice sizes.

## 7 Related Work

Optimistic hybrid analysis deliberately inserts unsoundness in its static analysis without sacrificing the accuracy of dynamic analysis. Our work builds on the considerable prior work done to combine static and dynamic analysis [21]. We classify this prior work according to the order of these analyses and their soundness properties.

**Sound static then dynamic.** Traditional hybrid analyses use sound static analysis to accelerate some form of dynamic analysis [13, 14, 19, 38, 39, 41, 44]. The requirement of soundness limits the precision and scalability of static analysis, ultimately resulting in a suboptimal dynamic analysis. Optimistic hybrid analysis uses unsound static analysis to optimize dynamic analysis, resulting in considerably faster analyses (Figures 5 and 6).

**Dynamic then (unsound) static.** Some static analyses use dynamic tools, such as Daikon [22], to gather information about a program's behaviors, then use this information to guide static analysis [17, 18, 26, 32, 36, 43, 45, 51]. Like predicated static analysis, these systems sacrifice the soundness of their static analysis by including dynamic invariants, but unlike optimistic hybrid analysis, they do not compensate for the resulting unsoundness in a later stage. In addition, because these invariants are never checked at runtime, they are chosen without regard for the cost of invariant checking. We propose and use invariants, such as likely unused call contexts, that reduce the state space of the analysis but still meet the criteria of cheapness and stability (2.1).

**Dynamic then unsound static then unsound dynamic.** A few systems learn likely invariants, then use these invariants in an unsound static analysis to produce a faster final dynamic analysis [15, 28]. However, these systems do not compensate for the unsoundness introduced by their unsound static analysis, so the final dynamic analysis is unsound. Conversely, optimistic hybrid analysis solves the unsoundness introduced in the static phase with speculative execution, and a carefully designed predicated static analysis.

In other related works, Lee et al. propose a deterministic replay system which leverages profiling, static analysis, and dynamic analysis[33]. Their system, however, uses profiling to aid in applying a sound static analysis. They could likely apply optimistic hybrid analysis techniques to leverage unsound static analysis for greater performance improvements.

Data-race detection is an important problem that many researchers have tried to accelerate, including with hybrid analysis [14, 19, 33, 44]. These techniques all rely on sound static analysis to retain dynamic accuracy, and could be further improved with the application of optimistic hybrid analysis.

Work on profile-guided optimizations [11, 37, 49], including those used in just-in-time compilers [10, 12], learn likely invariants though profiling and use them for optimizing a given program. Our work on optimistic hybrid analysis differs in two ways. First, profile-guided optimizations have focused on local analyses and optimizations (e.g., loop-invariant code motion). In contrast, we use likely invariants in whole-program analyses (e.g., pointer aliasing), allowing wide-ranging effects to the code. We identify likely invariants that enable us to perform several scalable and precise *whole program* context-sensitive static analyses, which are effective in reducing dynamic analysis overhead. Second, optimistic hybrid analysis is aimed at speculatively optimizing analysis code, whereas profile-guided optimizations is aimed at optimizing the original executable.

## 8 Conclusion

We argue that the traditional application of a sound static analysis to accelerate dynamic analysis is suboptimal. To this end, we introduce the concept of optimistic hybrid analysis, an analysis methodology that combines unsound static analysis and speculative execution to dramatically accelerate dynamic analysis without the loss of soundness. We show that optimistic hybrid analysis dramatically accelerates two dynamic analyses: program slicing and data-race detection.

## Acknowledgments

We thank the anonymous reviewers and our shepherd, Santosh Pande, for their thoughtful comments. This work was supported by the National Science Foundation under grants CNS-1513718, SHF-1703931, and CAREER-1149773. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

## References


[1] VimGolf. http://vimgolf.com, 2016. Accessed: 2016-07-31.

[2] Project Gutenbveg. (n.d.). http://www.gutenberg.org, 2017. Accessed: 2017-04-12.

[3] SvgCuts. http://svgcuts.com, 2017. Accessed: 2017-07-28.

[4] Agrawal, H., Horgan, J. R., London, S., and Wong, W. E. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on* (1995), IEEE, pp. 143–151.

[5] Andersen, L. O. Program analysis and specialization for the c programming language. In *PhD thesis, DIKU, University of Copenhagen* (1994).

[6] Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. Points-to analysis using bdds. In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 103–114.

[7] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (New York, NY, USA, Oct. 2006), ACM Press, pp. 169–190.

[8] Bond, M. D., and McKinley, K. S. Probabilistic calling context. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 97–112.

[9] Brace, K. S., Rudell, R. L., and Bryant, R. E. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (New York, NY, USA, 1990), DAC '90, ACM, pp. 40–45.

[10] Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande* (1999), ACM, pp. 129–141.

[11] Calder, B., Feller, P., and Eustace, A. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 1997), MICRO 30, IEEE Computer Society, pp. 259–269.

[12] Chambers, C., and Ungar, D. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not. 24*, 7 (June 1989), 146–160.

[13] Chang, W., Streiff, B., and Lin, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 39–50.

[14] Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany, June 2002).

[15] Csallner, C., Smaragdakis, Y., and Xie, T. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol. 17*, 2 (May 2008), 8:1–8:37.

[16] Devecsery, D., Chow, M., Dou, X., Flinn, J., and Chen, P. M. Eidetic systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation* (Broomfield, CO, October 2014).

[17] Dufour, B., Ryder, B. G., and Sevitsky, G. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 118–128.

[18] Dufour, B., Ryder, B. G., and Sevitsky, G. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2008), SIGSOFT '08/FSE-16, ACM, pp. 59–70.

[19] Elmas, T., Qadeer, S., and Tasiran, S. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (2007), pp. 245–255.

[20] Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).

[21] Ernst, M. D. Static and dynamic analysis: Synergy and duality. In *IN WODA 2003: ICSE WORKSHOP ON DYNAMIC ANALYSIS* (2003), pp. 24–27.

[22] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering* (New York, NY, USA, 1999), ICSE '99, ACM, pp. 213–224.

[23] Flanagan, C., and Freund, S. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 121–133.

[24] Flanagan, C., and Freund, S. N. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (New York, NY, USA, 2010), PASTE '10, ACM, pp. 1–8.

[25] Gupta, N., He, H., Zhang, X., and Gupta, R. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (2005), ACM, pp. 263–272.

[26] Gupta, R., Soffa, M. L., and Howard, J. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol. 6*, 4 (Oct. 1997), 370–397.

[27] Guyer, S. Z., and Lin, C. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis* (Berlin, Heidelberg, 2003), SAS'03, Springer-Verlag, pp. 214–236.

[28] Hangal, S., and Lam, M. S. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering* (May 2002), pp. 291–301.

[29] Hardekopf, B., and Lin, C. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 290–299.

[30] Hardekopf, B., and Lin, C. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium* (2007), Springer, pp. 265–280.

[31] Kim, T., Chandra, R., and Zeldovich, N. Efficient patch-based auditing for Web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation* (Hollywood, CA, October 2012).

[32] Kinder, J., and Kravchenko, D. Alternating control flow reconstruction. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation* (Berlin, Heidelberg, 2012), VMCAI'12, Springer-Verlag, pp. 267–282.

[33] Lee, D., Chen, P. M., Flinn, J., and Narayanasamy, S. Chimera: Hybrid program analysis for determinism. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation* (Beijing, China, June 2012).

[34] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: A manifesto. *Commun. ACM 58*, 2 (Jan. 2015), 44–46.

[35] Mangal, R., Zhang, X., Nori, A. V., and Naik, M. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 462–473.

[36] Mock, M., Atkinson, D. C., Chambers, C., and Eggers, S. J. Improving

program slicing with dynamic points-to data. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, USA, 2002), SIGSOFT '02/FSE-10, ACM, pp. 71–80.

[37] Mock, M., Das, M., Chambers, C., and Eggers, S. J. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2001), ACM, pp. 66–72.

[38] Myers, A. C. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (San Antonio, TX, January 1999), pp. 228–241.

[39] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not. 45*, 8 (June 2010), 31–40.

[40] Naik, M., Aiken, A., and Whaley, J. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 308–319.

[41] Necula, G. C., McPeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy code. *SIGPLAN Not. 37*, 1 (Jan. 2002), 128–139.

[42] Nethercote, N., and Seward, J. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation* (San Diego, CA, June 2007).

[43] Nimmer, J. W., and Ernst, M. D. Invariant inference for static checking:. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (New York, NY, USA, 2002), SIGSOFT '02/FSE-10, ACM, pp. 11–20.

[44] Rhodes, D., Flanagan, C., and Freund, S. N. Bigfoot: Static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 141–156.

[45] Sahoo, S. K., Criswell, J., Geigle, C., and Adve, V. Using likely invariants for automated software fault localization. *ACM SIGPLAN Notices 48*, 4 (2013), 139–152.

[46] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15*, 4 (November 1997), 391–411.

[47] Sengupta, A., Biswas, S., Zhang, M., Bond, M. D., and Kulkarni, M. Hybrid static–dynamic analysis for statically bounded region serializability. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 561–575.

[48] Smith, L. A., Bull, J. M., and Obdrzálek, J. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), SC '01, ACM, pp. 8–8.

[49] Steffan, J. G., and Mowry, T. C. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 25th International Symposium on Computer Architecture* (February 1998), pp. 2–13.

[50] Voung, J. W., Jhala, R., and Lerner, S. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (Dubrovnik, Croatia, 2007), pp. 205–214.

[51] Wei, S., and Ryder, B. G. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2013), ISSTA 2013, ACM, pp. 336–346.

[52] Weiser, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449.

[53] Zhu, J. Towards scalable flow and context sensitive pointer analysis. In *Design Automation Conference, 2005. Proceedings. 42nd* (June 2005), pp. 831–836.