Two in One: A case for Combined Points-to and MHP Analysis

RAMYA KASARANENI, Indian Institute of Technology, Madras, India
V. KRISHNA NANDIVADA, Indian Institute of Technology, Madras, India

With parallel programs becoming a mainstay, analyzing them has become an important requirement. Points-to analysis and May-Happen-in-Parallel (MHP) analysis are two of the most foundational analyses for parallel programs written in OO languages like Java. It is well known that precise MHP analysis needs points-to analysis results and precise flow-sensitive points-to analysis needs MHP analysis results. In this paper, we explore the interdependence between these analyses and propose new schemes to perform them efficiently.

We start by proving that there is a phase-ordering relation between MHP analysis and flow-sensitive points-to analysis; that is, they are interdependent such that no particular order of performing the analyses can yield the precise result. Next, we propose a novel combined MHP and points-to analysis scheme (called *ComPoMHP*) based on inclusion constraints, for real world parallel Java applications; this generates flow-sensitive, context-insensitive results; it even computes the required call graph information on the fly. We have implemented *ComPoMHP* in the Soot compiler framework and tested our analysis on thirteen benchmarks drawn from multiple sources. We show that the combined analysis leads to significant improvements in terms of precision of both MHP and points-to results, across seven different clients. We find that *ComPoMHP* leads to an excellent time-precision trade-off for points-to results, compared to the popular Doop based analyses. We also show that composing the points-to results of *ComPoMHP* with the results from existing Doop based flow-insensitive, context-sensitive analysis leads to highly precise results at affordable costs. We believe that our work paves the way for more precise and practical analysis of parallel Java programs.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering →** *Automated static analysis*; *Parallel programming languages*; *Object oriented languages.*

Additional Key Words and Phrases: program analysis, phase-ordering, inclusion-constraint based analysis

ACM Reference Format:
Ramya Kasaraneni and V. Krishna Nandivada. 2026. Two in One: A case for Combined Points-to and MHP Analysis. 1, 1 (July 2026), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

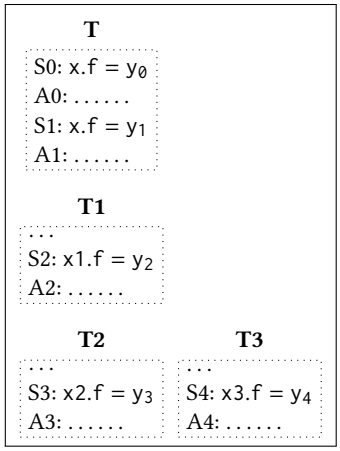
1 Introduction

As parallel hardware is becoming omnipresent, parallel programming in popular languages like Java [2], C# [21], Scala [44], Rust [26] is becoming more prevalent. Consequently, techniques for analyzing parallel programs are becoming essential and have received much attention. Some of the popular analysis techniques in this space include, May-Happen-in-Parallel (MHP) analysis [43], points-to analysis [59], thread escape analysis [12, 59], alias analysis [36], and so on. In this paper, we focus on two of these fundamental analysis techniques: MHP analysis and points-to analysis for Java programs.

MHP analysis identifies which pairs of statements run in parallel with each other, and points-to analysis identifies which objects a variable in a method or a field of an object may point to.

Authors' Contact Information: Ramya Kasaraneni, Indian Institute of Technology, Madras , Chennai, Tamil Nadu, India, raka@cse.iitm.ac.in; V. Krishna Nandivada, Indian Institute of Technology, Madras , Chennai, Tamil Nadu, India, nvk@iitm.ac.in.

ACM XXXX-XXXX/2026/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Java code snippet showing execution of multiple threads

Assume
Points-to information:
At S0, S1
$x \rightarrow \{0_x\}$, $y_0 \rightarrow \{0_0\}$,$y_1 \rightarrow \{0_1\}$
At S2,
$x1 \rightarrow \{0_x\}$, $y_2 \rightarrow \{0_2\}$
At S3,
$x2 \rightarrow \{0_x\}$, $y_3 \rightarrow \{0_3\}$
At S4,
$x3 \rightarrow \{0_x\}$,$y_4 \rightarrow \{0_4\}$

MHP information:
MHP(S0) = *MHP*(S1) = *MHP*(S2) = {};
MHP(S3) = {S4, A4};
MHP(S4) = {S3, A3}

(b) Points-to and MHP information for the code snippet in Fig. 1a

Fig. 1. Example illustrating the impact of parallelism on computing points-to information involving heap. T1, T2 and T3 are three threads starting after the statement A1. Assume that T2 and T3 start after T1 ends, which in turn starts after A1. *MHP*(*s*) returns the set of statements that may run in parallel with *s*.

Both of these are highly popular analyses with broad range of applications. MHP analysis can be used in debugging, detecting race conditions, deadlocks [4, 17, 30, 42], and in many other analyses and optimizations [41]. Points-to analysis is a prerequisite for many other analyses and optimizations. For example, in Java, points-to analysis is used for resolving virtual calls, precise call-graph construction, removing unnecessary synchronization, side effect analysis, and so on. It is well known that to perform precise flow-sensitive points-to (FSPT, in short) analysis, MHP information is required (to update the heap in a flow sensitive manner) [51]. Similarly, to compute precise MHP information, points-to information is required (to precisely identify the locking/notifying objects, and call-graph) [43]. Traditionally, the MHP analysis techniques [5, 11, 34, 43] have assumed the availability of points-to information. Similarly, conventional points-to analysis schemes have typically skipped handling of the parallel components, forcing them to assume that any statement may occur in parallel with any other statement. Consequently, these schemes inherently restrict themselves to perform only flow-insensitive update of the (shared) heap [45]. There have also been prior works that model parallel constructs to update heap flow-insensitively, and used this information to drive flow-sensitive analysis (for example, data-race detection [37], value-flow bugs [10], and so on). Considering the precision gains obtainable from FSPT analysis, a few prior works [19, 38, 47] compute FSPT results by using conservative MHP (or escaping objects) information. Though the interdependence between FSPT analysis and MHP analysis is known, we are not aware of any work that characterizes the dependencies between these analyses. We illustrate the dependencies with two motivating examples.

Consider the scenario shown in Figure 1. Assume a thread T executes two store statements, S0 and S1, and let Ai denote the statement immediately following Si (i = 0,1,2,3). After A1, three threads T1, T2, and T3 are started, which execute the store statements S2, S3, and S4, respectively as shown in Figure 1a. The flow-sensitive points-to information for variables at store statements, along with the MHP information for those statements are shown in Figure 1b. All the store operations update the field 'f' of the same abstract object (0_x). Assume that the abstract object 0_x corresponds to a

```
1 class X{ ...                                  7 class T extends Thread{
2  public void foo(T t){                        8   public void run(){ ...
3     t.start(); ...                            9     synchronized(L2){ x.f1 = ...
4     synchronized(L1){ ...                     10                       ...
5                       x.f1 = ... }            11                       x.f1 = ...
6     t.join(); ...} }                          12     } } }
```

Fig. 2. Code snippet illustrating the mutual interdependence between MHP and FSPT

single concrete object at run time (we term such objects as *non-summary* objects), and that these five are the only store operations on $O_x.f$. We now discuss computing the points-to information of heap in the presence of parallelism, focusing on $O_x.f$.

Note that the heap is shared among all the threads. Thus, flow-sensitive updates to heap require *MHP* information (for a detailed reasoning, see Section 2.3). Without it, the points-to analysis must conservatively assume that any statement may execute in parallel with any other—effectively making heap updates flow-insensitive. Even statements within the same method must be treated as potentially happening in parallel because multiple runtime threads may concurrently invoke that method. A flow-insensitive analysis of heap concludes that $O_x.f$ may point to $\{O_0, O_1, O_2, O_3, O_4\}$, since it maintains a single global points-to map. In contrast, handling of the store statement using MHP analysis facilitates flow-sensitive heap updation. Consider the store statement $x.f = y_0$ at S0. Since the *MHP* set of S0 is empty, the heap at S0 is not affected by any statement, aside from normal control flow. At S0, x points to a single non-summary object O_x, allowing a strong update [46] that sets $O_x.f$ to the points-to set of y_0 (= $\{O_0\}$) at the successor program point A0. Thus, at A0, $O_x.f$ may point to $\{O_0\}$. Similarly, at program point A1, $O_x.f$ may point to $\{O_1\}$; at A2 $O_x.f$ may point to $\{O_2\}$ and at both A3 and A4, $O_x.f$ may point to $\{O_3, O_4\}$. Since S3 and S4 may happen in parallel, their interleaving results in $O_x.f$ pointing to either O_3 or O_4. Such results are clearly more precise than the variant that does not use MHP analysis.

Consider another synthetic example code snippet shown in Fig. 2 that starts a thread at Line 3. The `start` method internally invokes the `run` method of the corresponding thread object. To compute MHP information, we need points-to information to find the possible threads that are starting at a line (for example, at Line 3). Similarly, given two synchronization blocks (for example, at Lines 4 and 9), to know if the statements within them (for example, Lines 5 and 11) may run in parallel, we need to establish two *key properties* (i) the entry point of these blocks (for example, Lines 4 and 9) *may* run in parallel, and (ii) the lock variables of the synchronization blocks (`L1` and `L2`) *must* not point to the same object. Similarly, to precisely compute the FSPT information of the fields of objects that are stored in the heap, we need to find out the statements that may happen in parallel with all the statements that write to these fields. As the heap is shared by all the threads, in the absence of precise MHP analysis, we have to conservatively assume that these fields can be updated by all the threads and perform flow-insensitive analysis.

These challenges of precision, along with the well known challenges of scalability in both these analyses, excite us to explore the following two questions in this paper: (i) Can a specific order between these two analyses (MHP analysis and FSPT) be specified for obtaining precise results? (ii) Can a combined analysis lead to improved precision for both the analyses? In addition, considering the known scalability issues of performing context-sensitive FSPT, we explore an additional question: (iii) How do the precision and scalability of context-insensitive FSPT aided by

MHP analysis, compare against those of popular Doop [9] based context-sensitive flow-insensitive points-to analyses?

Phase-Ordering Relation. An important aspect of MHP analysis and FSPT analysis, especially in the context of OO languages like Java, is that their interdependence is complex in nature. A common problem among the many analyses typically deployed in compilers and related tools is that finding the order among the analyses which gives the most precise result is a difficult challenge [55]. Moreover, the optimal order (if present) is often application-dependent. There have been works on finding the optimal order for a set of analyses/optimizations [14, 27, 60, 61]. They are applicable if the analyses under consideration are not mutually dependent. In cases where the analyses are mutually dependent, no single order or even repetitive alternating application can result in optimal results. Such dependent analyses are said to have a *phase-ordering* relation between them, and thus there is no best ordering between them. In this paper, we show (in Section 3) that there is a phase-ordering relation between MHP analysis and FSPT analysis.

Combined Analysis. We take inspiration from many prior works that combine passes [8, 13, 16, 18, 23, 28, 40, 58] with phase-ordering relations, and propose a combined analysis to perform context-insensitive FSPT analysis and MHP analysis. We also compute the required call-graph information in the process. To the best of our knowledge, this is the first work that performs such a combined analysis (and computes both MHP and points-to analysis results), especially in the presence of constructs like join, wait/notify, synchronized, and so on. Further, our analysis does not make highly restrictive/unscalable choices (for example, the requirement to inline all the method calls, the cloning of thread and lock objects to resolve aliasing, and so on [29, 43]) that made prior MHP analysis schemes unsuitable for real-world Java applications. We represent the individual analyses in a modular way, encode their interdependence in terms of inclusion constraints (similar to that of Andersen [1], Oxhøj et al. [45]), and solve them.

A natural question is why encoding MHP and FSPT as interdependent inclusion constraints and solving them iteratively yields a precise result that no sequential ordering (or repetition) of FSPT and MHP can achieve. A sequential schedule cannot avoid the phase-ordering problem because the analysis performed first must rely on a worst-case estimate of the other analysis, whose results are not yet available. Once this imprecision is introduced, it cannot be eliminated in later iterations (see Section 3). Our combined analysis avoids this problem by seeding neither analysis with a worst-case assumption: it grows the MHP and points-to information together, starting from the empty set, and adds information only when a constraint is actually triggered. As a result, the solve converges to the same precise result regardless of the order in which constraints are processed. One challenge here is the *must* information that is required to guard (a) strong updates to the heap, and (b) the pruning of non-parallel statement pairs. For soundness, we require that the truth value of these guards can never change from `false` to `true` as the *may* information (i.e., the MHP and points-to results) grows. We therefore pre-compute a sound, conservative estimate of just this information once, and treat it as a fixed set during the solve. This allows the *may* information to evolve jointly in an order-insensitive manner while keeping the required *must* information unchanged.

Flow-sensitive context-insensitive analysis vs. flow-insensitive context-sensitive analysis. Prior works [49, 50] have shown that context-sensitivity leads to serious issues of scalability for FSPT analysis. Thus, in this work we focus on context-insensitive, FSPT analysis – the flow-sensitivity, especially for updating the heap gets enabled due to the availability of MHP information. Interestingly, we find that our proposed approach can be an effective alternative to the popular context-sensitive, flow-insensitive approaches for points-to analysis, in terms of time-precision trade-off. However, extending our proposed ideas to context-sensitive, FSPT analysis [54] to extract further precision in a scalable manner is left as future work.

Sets	
Vars	= Set of all variables in P.
Classes	= Set of all classes in P.
Functions	= Set of all Functions in P.
Refs	= Set of all abstract references in P.
Fields	= Set of all the fields in P.
Stmts	= Set of all the statements in P.

Maps ($\mathcal{P}(X)$ *indicates the power set of the set* X)	
varPts	$: Stmts \times Vars \rightarrow \mathcal{P}(Refs)$
fieldPts	$: Stmts \times Refs \times Fields \rightarrow \mathcal{P}(Refs)$
MHP	$: Stmts \rightarrow \mathcal{P}(Stmts)$

Fig. 3. Sets and Maps pertinent to flow-sensitive points-to (FSPT) and MHP analyses for a given program P.

Our Contributions.

• We show that a phase-ordering relation exists between MHP and FSPT analyses for Java programs.

• We propose the first known solution for performing combined MHP and FSPT analysis (termed *ComPoMHP*); we use an inclusion-constraint based approach. We take advantage of the modular nature of our proposed scheme and also derive the first inclusion constraint based formulation of the standalone MHP analysis for Java (that assumes the availability of points-to results), by making some minor simplifications to the formulation of *ComPoMHP*.

• We implemented *ComPoMHP* and the standalone MHP analysis in the Soot framework [57]. We present an elaborate evaluation of our analysis on thirteen benchmarks from 9.12, 23.11 versions of DaCapo [6], and real world Java applications. We show that the combined analysis leads to significant improvements in the precision of MHP and points-to analysis, compared to schemes that perform these analyses one after the other. For example, the improvement in precision of MHP results varied from 0.0% to 99.8%; geomean = 5.5%, median = 3.9%. Similarly, the improvement in precision of points-to results varied from 2.3% to 85.0%; geomean = 37.8%, median = 41.5%. We also show that in contrast to existing schemes, the standalone MHP analysis successfully analyzes all these large benchmarks.

• As an additional evaluation, we compared *ComPoMHP* (FSPT with MHP) against the 1-object-sensitive (*D1obj*) and adaptive-2-object-sensitive+heap (*Da2objH*) (flow-insensitive and context-sensitive) points-to analyses of Doop [9] and found that *ComPoMHP* outperforms these analyses, in many cases, in terms of analysis time and memory consumption. In terms of quantitative metrics signifying precision, we found that *ComPoMHP* is comparable to those of *D1obj* and *Da2objH*. Based on a simple observation, we find that the results of *ComPoMHP* can be combined with those of an off-the-shelf flow-insensitive, context-sensitive analysis (like *D1obj*, and *Da2objH*) to obtain very highly precise (yet, conservative and sound) results. We show that the composed results significantly improve the precision.

Even though we propose our techniques in the context of Java programs, we believe that they can also be extended to other OO languages like Scala, C#, and so on.

2 Background

2.1 MHP and Points-to Sets

We represent the flow-sensitive points-to (FSPT) information via two maps *varPts* and *fieldPts* maps, and MHP information via *MHP* map (see Fig. 3). For each program point, the abstract stack *varPts* maintains the information about what a variable may point to at different program points, and the abstract heap *fieldPts* maintains the information about what the fields of different references may point to. The *MHP* map maintains the set of statements that may run in parallel with each statement. For each memory allocation statement s, we use an abstract reference O_s to represent all the references that may be created at s, during runtime. We use a unique abstract object to represent null. For the ease of explanation we use a super-graph, a graph obtained upon taking a

union of all the CFGs of the functions, along with call and return edges, to represent the program; the entry node of the main-function is the root of this super-graph.

2.2 Summary and Non-Summary Statements and Objects

We mark a statement as a non-summary statement if it is guaranteed to be executed at most once; else, it is marked as a summary statement. We can identify a statement s to be a summary (or, non-summary) statement, if there exists multiple paths (or, a unique path) from the entry-node of super-graph to s. Using allocation site abstraction, an object is considered non-summary if its allocation site corresponds to a non-summary statement; otherwise, it is marked as a summary object.

Consider a store statement S_1 of the form `x.f = y`. On processing this statement, the points-to maps of heap locations can be updated in two ways: strong or weak [46]. A strong update is used when the analysis can identify a single concrete target location (for example, `x` is guaranteed to point to a singleton set containing a non-summary object, say O_n). In such cases, the points-to information of $O_n.f$ can be replaced with $varPts(S_1, \mathtt{y})$. A weak update is used when multiple target locations are possible (for example, when `x` may point to more than one object, or when `x` points to only one object, say O_s and O_s represents a summary object). In such cases, in contrast to the idea of replacement in case of strong-update, for each object $O \in varPts(S_1, \mathtt{x})$, $varPts(S_1, \mathtt{y})$ has to be added to the points-to information of $O.f$.

A special case: Consider a scenario in which $varPts(S_1, \mathtt{x})$ contains a single object O_s created at a summary allocation statement s. Since s is marked as a summary statement, O_s is also marked as a summary object. Because O_s is a summary object, the above discussion requires that a weak update is performed at S_1. However, inspired by prior work [3, 39], we identify additional opportunities to perform strong update at S_1 by marking O_s as a non-summary object at S_1, if (i) for each instance of execution of S_1 there is a unique instance of s that is executed, and (ii) s dominates S_1 in the program super-graph. Otherwise, we mark O_s as a summary object at S_1. Consequently, while it is enough to maintain a global set of non-summary statements, we need a per-statement set of non-summary objects to perform flow-sensitive analysis.

2.3 Points-to Analysis in the Presence of Parallelism

We briefly examine the key concepts of Java points-to analysis in the presence of parallelism. Flow-insensitive points-to analysis ignores control flow, and hence the presence (or absence) of parallelism in the input program has no impact on the computed results. Flow-sensitive analysis, however, takes into consideration the control-flow in the program, and tracks points-to maps for variables and heap locations at each program point associated with every statement. At each program point, it updates the points-to maps based on the semantics of the corresponding statement and propagates this information to the control-flow successors [35] of the statement. In the context of parallel programs, updates to shared locations must also be propagated to statements that may execute concurrently, as determined by May-Happen-in-Parallel (MHP) analysis. Since variables are thread-local, only the points-to information about the heap objects and their fields needs to be propagated across potentially parallel statements. Without MHP information, updates must be propagated to all statements in the program, effectively reducing the analysis of shared heap to a flow-insensitive one. Thus, precise flow-sensitive updation of points-to maps of shared locations (like heap), for FSPT analysis, requires MHP information.

Besides precise points-to analysis results, precise MHP analysis also requires flow-sensitivity. That's because a flow-insensitive MHP analysis is overly conservative: since it does not account for control-flow ordering, it could assume that each statement may execute in parallel with every statement. This commonality of flow-sensitivity and interdependence between MHP analysis and

FSPT, forms the basis of the phase-ordering relation between them (Section 3), and our proposed combined analysis (Section 4).

2.4 Relevant Java Constructs

Call Stmt. In a call statement of the form `x.bar(..)`, the object pointed-to by `x` is termed as the receiver. Inside `bar`, the special variable `this` (0^{th} argument) points to the object pointed to by the receiver.

Parallel constructs. In Java, a programmer realizes parallelism by the use of threads. In this paper, for the ease of exposition, we mainly focus on a subset of Java constructs for realizing parallelism: thread creating, joining, waiting, notifying, and synchronizing. In Section 5, we briefly describe how we deal with other concurrency-related constructs of Java.

Thread creation and joining. In Java, a thread-type class extends the `Thread` class or implements the `Runnable` interface. A thread is an object created by instantiating a thread-type class. Such a class must implement the `run` method which is the method invoked internally when a thread is started (by invoking the `start` method on the thread object). A join function call of the form `t.join()` will wait for the termination of the thread object pointed-to by `t`.

Mutual Exclusion. Accesses to shared memory can be guarded by using the `synchronized` construct in Java. Each object in Java has an associated lock. If blocks of code (executed by two or more threads) are guarded by the same lock object, the codes are executed in a mutually-exclusive manner. The syntax to create a monitor block is as follows: `synchronized (p) S`. Internally, we replace such a block by three statements *syncStart* `p`; `S`; *syncEnd* `p`.

Thread interaction. A thread can send a notification to any one or all the waiting threads (on an object pointed-to by `x`) by calling `x.notify()` or `x.notifyAll()`, respectively. A thread can suspend its execution using a `wait` call of the form `x.wait()`, till a notification (on the same object pointed-to by `x`) is sent by another thread.

3 MHP and Points-to Analyses: Phase-Ordering

Many prior works [47, 51] have recognized that precise MHP analysis requires points-to information and precise flow-sensitive points-to (FSPT) analysis of parallel programs requires MHP analysis. However, we have not found any work that discusses if there is a phase-ordering relation between the two; if such a relation exists, then there will be no particular order in which these two passes can be run (in a compiler, one or more times) to obtain precise results. In this Section, we answer this question in the affirmative. We take inspiration from Nandivada and Palsberg [40] and prove the existence of a phase-ordering relation by using an example. The example shows that performing MHP analysis followed by FSPT, or vice-versa, any number of times, can leave scope for improving the precision of the points-to and MHP results.

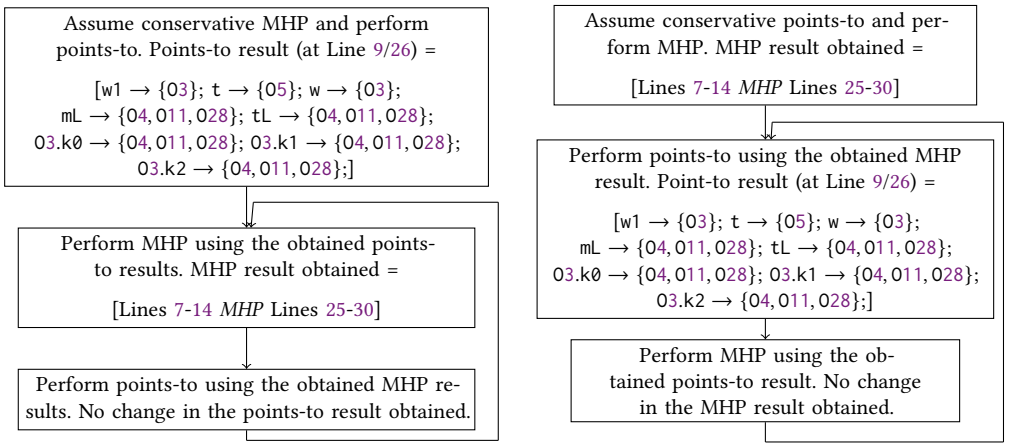
Consider the synthetic code snippet shown in Fig. 4a. The `main` method in `Example` class, starts and joins a thread of type `T`. The created thread runs in parallel with the synchronized block at Line 9, let us name the synchronized block as `B1` (Lines 10-12). The run method of `T` contains another synchronized block at Line 26, let us name the block `B2` (Lines 27-29). A class type `LockWrap` has three fields `k0, k1, k2`. The main method has a variable `w1` and `T` has a field `w` of type `LockWrap`. Observe that at Line 6, `w1.k0` and `w.k0` both point to the same object {`O4`}. The lock variables `mL` and `tL` also point to {`O4`} as they are aliases of `w1.k0` and `w.k0` respectively from Lines 8, 25. So, the two synchronized blocks `B1` and `B2`, whose lock variables are `mL` and `tL` respectively, will not run in parallel. Note that if `B1` and `B2` do not run in parallel (if they run one after the other `B1;B2` or `B2;B1`), then the points-to information of `w1.k0` and `w.k0` does not change anywhere after the initial assignment. They both continue to point to {`O4`}. We now discuss two possible orderings:

Perform FSPT analysis first (Fig. 4b). We make a conservative assumption about MHP information and assume that any statement may run in parallel with any other statement (including Line 4,

```
1 public class Example{
2   public static void main(...){
3     LockWrap w1 = new LockWrap(); //O3
4     w1.k0 = new MyLock();  //O4
5     T t = new T(w1);  //O5
6     t.start();
7     for(...){
8       MyLock mL = w1.k0;
9       synchronized(mL){ /*B1*/
10        w1.k2 = w1.k0;
11        w1.k1 = new MyLock(); //O11
12        w1.k0 = w1.k2;
13      } } /* end-for */
14    t.join();
15    } }
```

```
16 class LockWrap{
17   public MyLock k0, k1, k2;
18 }
19 class MyLock { ...
20 }
21 class T extends Thread{
22   LockWrap w;
23   public T(LockWrap w){this.w = w;}
24   public void run(){
25     MyLock tL = w.k0;
26     synchronized(tL){ /*B2*/
27       w.k1 = w.k0;
28       w.k2 = new MyLock(); //O28
29       w.k0 = w.k1;
30     } } }
```

(a) Example (abstract object created at line# n is denoted by On)



(b) Perform points-to analysis first and repeat (c) Perform MHP analysis first and repeat

Fig. 4. Illustration for interdependence of FSPT and MHP analyses.

Lines 10-12 (B1) and Lines 27-29 (B2)). With such an interleaving, upon completion of the analysis, both `w1.k0` and `w.k0` may point-to {O4, O11, O28}, at Lines 8 and 25, respectively. Note that even if we use some simple heuristic to ignore the assignment at Line 4, `w1.k0` and `w.k0` will still point-to a non-singleton set {O11, O28}. Because of the presence of the loop (at Line 7), upon reaching the fixed-point, `w1.k0` and `w.k0` point to {O4, O11, O28}. Similarly, `mL` and `tL` also point to {O4, O11, O28}. Now, performing MHP analysis with this points-to information, will again give the result that the statements at Lines 10-12 may run in parallel with statements at Lines 27-29. Note: statically, two synchronization blocks are guaranteed to not run in parallel, only if both the lock variables point to a single non-summary object (see Section 2.2).

Perform MHP analysis first (Fig. 4c). We make a conservative assumption about points-to information and assume that every variable may point-to every possible object (of matching type). With such an assumption, the MHP analysis will conservatively conclude that the two synchronized

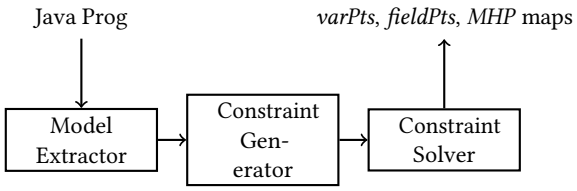


Fig. 5. Block diagram showing the working of our proposed approach.

blocks may run in parallel, as the lock variables `tL` and `mL` point-to multiple objects. Now, performing points-to analysis with this MHP information, will lead to the same imprecise points-to information as before: upon reaching the fixed-point, `w1.k0` and `w.k0` point to {`O4`, `O11`, `O28`}.

We can continue the process further and find that in both the orderings, even repeating the sequence any number of times does not lead to the precise conclusion: that at Line 7, `w1.k0` and `w.k0` may only point to {`O4`}, and Lines 10-12 and Lines 27-29 may not run in parallel. Despite repeated application of these analyses, we do not obtain a precise conclusion because each analysis needs the precise results of the other analysis to improve its own precision. Since we start with a conservative (and therefore imprecise) assumption about one of the analyses, neither of the analyses gives precise results. Hence, we say that there is a phase-ordering relation between MHP and FSPT analyses.

Note: there is no such relation between MHP and flow-insensitive points-to analyses, as flow-insensitive analysis does not require MHP information, since it does not consider the control flow and does only weak updates. Here, the optimal ordering is trivial: flow-insensitive points-to analysis, followed by MHP analysis, as done by many prior researchers [22, 31, 32].

4 The Combined Points-to and MHP Analysis

Considering the phase-ordering relation between FSPT and MHP analyses, we propose a combined points-to (flow-sensitive, context-insensitive) and MHP Analysis. We refer to it as *ComPoMHP*.

The *ComPoMHP* algorithm takes as input a Java application and the entry point (main method) and generates FSPT and MHP analysis results in three maps, as shown in Fig. 3. Fig. 5 shows the simplified block diagram of our proposed approach. We now describe the details of model extraction (Section 4.1), constraint generation (Section 4.2) and constraint solving (Section 4.3).

4.1 Model Extraction

From each function in the input program, we extract the sets shown in the left side of Fig. 3. In addition to these sets, we also maintain five auxiliary maps to model the object-type information, control-flow-graph (CFG) related information, and the concurrency-related aspects of the program.

(1) $typeOf : Refs \rightarrow Classes$. Given any abstract reference r, $typeOf(r)$ returns its type.

(2) $succ : Stmts \rightarrow \mathcal{P}(Stmts)$. Given any statement s, $succ(s)$ returns its set of CFG successors.

(3) $monPair : Stmts \rightarrow Stmts$. Given a *syncEnd* stmt s, $monPair(s)$ returns its *syncStart* stmt.

(4) $funcStmts : Functions \rightarrow \mathcal{P}(Stmts)$. For any function f, $funcStmts(f)$ returns the set of statements (intraprocedural) inside f.

(5) $func : Stmts \rightarrow Functions$. For any statement s, $func(s)$ gives the function to which it belongs.

4.2 Constraint Generation

After the model extraction, we generate inclusion constraints from the model. In addition to the variables (maps) listed in Fig. 3, the generated constraints use many additional maps to handle call-graph and concurrency-related aspects. We have listed them in Fig. 6, and the reader may refer to this figure, as and when required. As the analysis is flow-sensitive, we maintain these maps at

Additional Maps ($\mathcal{P}(X)$ *indicates the power set of the set* X)		
retVal	: *Functions* $\rightarrow \mathcal{P}$(*Refs*)	For any function f, *retVal*(f) returns the set of references that f may return.
outMHP	: *Functions* $\rightarrow \mathcal{P}$(*Stmts*)	For a function f, *outMHP*(f) returns the set of statements that may run in parallel with the statements that execute immediately after the call to f.
callSites	: *Functions* $\rightarrow \mathcal{P}$(*Stmts*)	For any function f, *callSites*(f) returns the set of call-sites of f.
funcsAtCallSite	: *Stmts* $\rightarrow \mathcal{P}$(*Functions*)	This is an inverse map of *callSites* map. For a statement s which is a call site, *funcsAtCallSite* gives the functions that may be invoked at s.
reachableFuncs	: *Functions* $\rightarrow \mathcal{P}$(*Functions*)	For a function f, *reachableFuncs*(f) gives the set of functions that are reachable from the function f.
sync	: *Refs* $\rightarrow \mathcal{P}$(*Stmts*)	For any reference r, *sync*(r) returns the set of stmts synchronized on r.
thStmts	: *Classes* $\rightarrow \mathcal{P}$(*Stmts*)	Given a thread-type class c, *thStmts*(c) returns the set of statements of the `run` method of c(including statements from all reachable methods).
monStmts	: *Stmts* $\rightarrow \mathcal{P}$(*Stmts*)	Each monitor block is represented by the starting statement of the block; we call that statement the leader of the block. *monStmts*(s) returns the set of statements of the monitor block with leader s
mustRemoved	: *Stmts* $\rightarrow \mathcal{P}$(*Stmts*)	Given a statement s, *mustRemoved*(s) returns the set of statements in *MHP*(s) that are guaranteed to not run in parallel with the successors of s (identified using the *must* information) and therefore not added to their *MHP* sets, when s is a monitor entry.
notifyStmts	: *Refs* $\rightarrow \mathcal{P}$(*Stmts*)	*notifyStmts*(r) returns the set of *notify*, *notifyAll* statements in monitor blocks controlled by r.

Fig. 6. Additional maps used in the constraint formulation.

each program point (before each statement). We now illustrate some novelties of our constraint generation scheme. For simplicity, we assume that the program is in three-address-code [35] form.

Types of Constraints. For each statement, in each method, we generate constraints indicating how the statement affects the points-to and MHP information. A statement may or may not affect (or use) either points-to or MHP information. Similar to Oxhøj et al. [45], we generate three types of constraints: Member, Propagation, and Conditional, as shown in Fig. 7. For the ease of presentation, we use parametric conditional-constraints: our conditional-constraints have the form '$cond_r$, C_r , $elseC_r$', where $cond_r$ is a condition, and C_r and $elseC_r$ are constraints. Such a conditional-constraint indicates that the constraint C_r holds when the condition $cond_r$ is true; otherwise, $elseC_r$ holds. The subscript r is used in the term (condition or constraint) to highlight that it may be using a temporary variable r. We next define the possible syntax of $cond_r$.

Let us assume that A is a set of objects (or statements). The condition $cond_r$ in the conditional-constraint can be an elementary condition (membership condition of the form $r \in A$, a cardinality condition of the form $|A| == 1$, an existence condition of the form $A.\mathtt{has}(r)$), or a conjunction of elementary conditions. A conditional constraint of the form $(r \in A, C_r)$ is a short form for the series of equations: $\forall r, (r \in A) \Rightarrow C_r$. In the conditional constraint (1) the *else* part may only be used with cardinality condition or existence condition, and (2) C_r and $elseC_r$ can be any constraint (can even be a nested conditional-constraint), that may use the temporary r. We use ϕ in place of C_r or $elseC_r$ to indicate that this component is absent. (3) a cardinality constraint for a set X is used only as a condition nested inside a membership condition on X. Thus, the cardinality needs to be checked only when a new member is added to X.

Similar to conditional constraints, for brevity in presentation, we use an additional type of constraint called FunctionCall-constraint. A FunctionCall-constraint is of the form $[\![cond_r; S_1]\!]$, where the $cond_r$ can only be a membership condition on the receiver variable. For example, consider a statement S_c of the form $x.bar()$ and a FunctionCall constraint $[\![oVar \in varPts(S_c, x); S_c]\!]$. This constraint indicates that the call-site S_c should be processed by considering each object (*oVar*)

that the receiver x may point to. This processing involves generating conditional constraints for (i) passing the values of the actual arguments to the formal ones, (ii) copying the MHP maps to the callee, and (iii) copying the return values and MHP maps back to the caller.

Insight Behind the Combined Approach. In Section 3, we showed that analyses with a phase-ordering dependency cannot be run sequentially without losing precision because whichever analysis runs first must use the imprecise results from the other, and that imprecision may not be always removable. To combine two such analyses A_1 and A_2 precisely, refinement of the results for A_1 (or, A_2) must occur during the combined analysis, whenever results of A_2 (or, A_1) get updated, so that such imprecision is avoided. For example, as discussed in Section 3, for the code snippet shown in Fig. 4a, FSPT can conclude that `mL` and `tL` both point to the same object, if and only if MHP analysis concludes that Lines 10-12 do not run in parallel with Lines 27-29. Recall that running either of the analyses ahead of the other, may lead to overall imprecise results. In contrast, if we can state the analyses as interdependent constraints and repeatedly process them, every time one of the results changes — we can get the most precise results.

In our combined analysis, to break the phase-ordering dependency between these interdependent analyses, we encode dependencies between them using conditional constraints (of the form $cond_r, C_r, elseC_r$), such that an analysis result is propagated (only) when the requisite information from the other analysis is available. Such conditional constraints with non-empty $elseC_r$ present a unique challenge: for soundness we require a property that the value of the predicate $cond_r$ can change from `true` to `false`, but never the other way round. However, if $cond_r$ requires the checking of membership for a set (say X) being computed by the combined analysis, this property may be violated. Since X may grow monotonically, a particular element e which was initially absent ($e \in X$ gets evaluated to `false`) may be added later ($e \in X$ gets evaluated to `true`). To avoid such cases, we pre-compute a conservative estimate for X (using a helper analysis), and use that estimate using an *existence* condition (of the form $X.\texttt{has}(x)$).

We now make this key insight precise. The combined analysis is order-insensitive because its maps grow additively and are never seeded using a worst-case estimate of either analysis (Sections 3, 4.3). This additive growth applies only to the *may* information computed by the two analyses; however, the combined analysis also relies on certain *must* information to guard some operations. For example, when computing MHP information for statements inside a monitor block, statements in other monitor blocks can be excluded only if the lock variables of both monitors refer to the same non-summary object (an abstract object that may have at most one runtime instance; see Section 2.2), which is a *must* property. Without this must property, other monitor statements cannot be conservatively excluded, and the result will be less precise. A straightforward solution would be to pre-compute the entire guard, but that would incur significant imprecision. Instead, we encode such guards using a mix of variables computed by the ongoing analysis and some pre-computed *must* information: the sets of statements and objects have at most one runtime instance, together with the conservative thread-statement and synchronization maps (*nonSummaryStmts*, *nonSummaryRefs*, *thStmtsHA*, *syncHA* of Section 4.2.2). As discussed above, if a guard depends on information that is still being computed, its value may change from `false` to `true` during the analysis. Such changes would violate the monotonicity on which order-insensitivity rests, whereas a pre-computed, frozen estimate keeps it intact. This pre-computation is therefore essential. Without it, the proposed analysis could not soundly perform its precision-inducing steps — flow-sensitive (strong) heap updates, and the precise modeling of join, wait, and synchronization. Note that even though we use the pre-computed information only where the monotonicity discipline forces us to, its precision may still impact the overall precision of the analysis (see Section 8).

Generation of Constraints. For every pair of statements (S_1, S_2), where S_2 is one of the control-flow successors [35] of S_1, we now show the constraints that are generated based on the

Constraint type	Form	Example
Member	$M \in A$	$[\![O_1 \in varPts(S, x)]\!]$
Propagation	$A \subseteq B$	$[\![varPts(S, x) \subseteq varPts(S, y)]\!]$
Conditional	$\begin{cases}\text{If } (cond_r):\\ C_r\\ [elseC_r]\end{cases}$	$\begin{cases}\text{If } (oVar \in varPts(S, x)):\\ [\![fieldPts(S, oVar, f) \subseteq varPts(S', y))]\!]\\ \phi\end{cases}$
FunctionCall	$cond_r; S_c$	$[\![oVar \in varPts(S_c, x); S_c]\!]$

Fig. 7. Constraint types. Notation used: A and B indicate flow-sets, C and $elseC$ indicate constraints; M indicates any element in the flow-set; r, and $oVar$ indicate temporary variables used by our scheme (not application variables); $cond_r$, C_r, and $elseC_r$, indicate conditions and constraints that may be referring to a temporary r; S and S' indicate any arbitrary statement; and S_c indicates a call statement.

statement-type of S_1. The constraints indicate how the different maps at S_1 are transferred to S_2, based on the effects of S_1. Next, we show the constraint generation for different types of statements.

For the ease of explanation, we divide the statements into two categories: Non Concurrency-related statements and Concurrency-related statements, and give the constraint generation rules in Sections 4.2.1 and 4.2.3, respectively. In these sections, we only show the constraints for S_2, where its maps differ from that of S_1. The remaining maps from S_1 are copied to S_2; not shown for brevity. Although all constraints appear additive, flow-sensitivity is ensured by indexing maps with program statements (rather than a single global map), and selectively propagating the maps from S_1 to S_2. For example, consider the piece of code:

```
S0: x = new A(); // O1
    y = new B(); // O2

S1: x = y;
S2:
```

Assume that before processing these statements *varPts*(S0, x)=*varPts*(S0, y) = {}. Before processing S1, *varPts*(S1, x) = {O1}, and *varPts*(S1, y) = {O2}. Our constraints will (i) compute the value *varPts*(S2, x) using the constraint *varPts*(S1, y) ⊆ *varPts*(S2, x); (ii) will not copy the *varPts*(S1, x) to *varPts*(S2, x); and (iii) copy *varPts*(S1, y) to *varPts*(S2, y), because y is not being updated. In contrast, a flow-insensitive analysis will compute a global map where x → {O1, O2}.

4.2.1 Non Concurrency-related Statements. This section shows the constraints that will be generated if S_1 is a non concurrency-related statement. Such a statement does not directly lead to an update of the MHP related maps. However, the points-to related information may get impacted due to the MHP information. The constraints, along with brief explanations, are shown in Figure 8. To clearly identify the constraints that depict the dependence of points-to information on MHP information, we use a dashed circle to number such constraints; for example, ⟨2.1⟩. Note that the information flow for the points-to maps is standard, except that the flow now depends on the MHP maps as well.

We would like to highlight the handling of the store statement. A strong update [46] can be performed for a store-statement if the analysis can infer that the points-to set of x is a set containing a single abstract object, and that abstract object is a not a summary object (see Section 2.2). We evaluate this property, for a points-to set A at S_1, using the predicate $must(A)$ defined below:

$$must(A) = (|A| == 1 \wedge nonSummaryRefs(S_1).\texttt{has}(x)), \text{ where } x \text{ is the element in } A$$

We use a simple pre-analysis *HA* to conservatively compute the set *nonSummaryRefs* of such objects at each statement.

We handle arrays similar to Spark in Soot [57], in a field-insensitive manner: Arrays are treated like objects with a single virtual field. Assignment to any element of an array assigns to this field. The rules for arrays are skipped for brevity.

4.2.2 Helper Analysis HA. In our formulation of the combined analysis, we more or less compute all information that our analysis depends on (like, call graph information, statements of a thread, and so on). However, as described earlier in this section (see the discussion on 'Insight'), for soundness, we have to pre-compute and use the conservative estimates of a small number of sets. The sets *nonSummaryStmts* and *nonSummaryRefs* discussed above are two such examples.

The version of *HA* we employ is a constraint-based, flow-sensitive points-to analysis, with heap updates handled flow-insensitively, and it uses standard points-to analysis constraints. Given an input program, *HA* computes *nonSummaryRefs* and *nonSummaryStmts* maps, *thStmtsHA*, and *syncHA* maps; we use these maps in *must* condition checks, as well as existence conditions. Computing these maps requires points-to information and call-graph which is derived from points-to information. Points-to information is computed using the constraints shown in Fig. 8, with a key modification to the store and load constraints: the *fieldPts* map is made flow-insensitive. Specifically, *fieldPts* is maintained as a global structure without statement indices, and the store constraint omits the *must* condition, resulting in only weak updates.

Now, we will see how we compute the non-summary statements (*nonSummaryStmts*) and non-summary objects (*nonSummaryRefs*), discussed in Section 2.2 from the points-to information computed by the helper analysis *HA*. The computation of *thStmtsHA* and *syncHA* is straightforward, and is described in Section 4.2.3. Given a statement s, we ensure that $s \notin nonSummaryStmts$, if s can be executed multiple times at run time. We find it by checking if s is inside a loop, or a function that can be invoked multiple times; the latter can be found using the call graph obtained from *HA*.

Next, *nonSummaryRefs* map is computed for each statement using the following rules. Given a statement s and an object o which represents the abstract object created at an allocation site s_a,

(1) If $s_a \in nonSummaryStmts$, then $o \in nonSummaryRefs(s)$ because, we can guarantee that there can be at most one runtime instance of o created.
(2) If $s_a \notin nonSummaryStmts$, there can be multiple runtime instances of o. But if we can determine that s will always receive a fresh (most recent) instance of object o created at s_a, then we can conclude that $o \in nonSummaryRefs(s)$ [3, 39]. This is possible if s always runs together with s_a and only after s_a in any execution sequence. This ensures a one-to-one correspondence between executions of s_a and s. To find this, we use an approach based on call graph and dominance.
   (a) For any such statement s_a, we try to identify a unique statement called *loopOrigin* which is present inside a loop and *loopOrigin* must execute for s_a to execute. To compute *loopOrigin*, we define the function $loopOrigin = findLoopOrigin(s_a)$ as below:
      (i) If s_a is inside a loop: return s_a
      (ii) Else /* s_a is inside a method that is called multiple times */
         - if $func(s_a)$ has multiple call sites (>1): No unique *loopOrigin* exists (multiple independent invocation points exist, so objects from different invocations can reach s). return *null*
         - Else: Let c = only call site of $func(s_a)$
           return $findLoopOrigin(c)$
   (b) We then verify whether *loopOrigin* dominates s, and whether s resides within the same immediately enclosing loop as *loopOrigin*. If such a *loopOrigin* exists, then o is added to $nonSummaryRefs(s)$.

Type of S_1	**Generated Constraints** ⌢ points-to depends on MHP ◯ independent	**Explanation**
1. Copy: $x = y$	(1.1) $[\![varPts(S_1, y) \subseteq varPts(S_2, x)]\!]$	The points-to set of y at S_1 flows into the points-to set of x at S_2. This statement does not affect/use *MHP* information as x and y are local variables (allocated on the stack, local to a thread).
2. Store: $x.f = y$	(2.1) ⌢ If $(s \in \{\{S_2\} \cup MHP(S_1)\})$: If $(r \in varPts(S_1, x))$: If $(must(varPts(S_1, x)))$: $[\![varPts(S_1, y) \subseteq fieldPts(s, r, f)]\!]$ $[\![(varPts(S_1, y) \cup fieldPts(S_1, r, f)) \subseteq fieldPts(s, r, f)]\!]$ ϕ ϕ	For each reference r in the points-to set of x at S_1, the points-to set of y flows into the points-to set of $r.f$ at all statements in $\{\{S_2\} \cup MHP(S_1)\}$ Since heap (shared among threads) is being accessed, *MHP* information is required for FSPT analysis (see Section 2.3). A strong update (see Section 2.2) is performed when the points-to set of x is singleton, and the referenced object is a non-summary object (checked using *must*). When doing a strong update, only $varPts(S_1, y)$ is propagated. In contrast, under a weak update (as specified in the corresponding else constraint), $varPts(S_1, y)$ and the existing points-to information of the field are both propagated.
3. Load: $x = y.f$	(3.1) ⌢ If $(s \in \{\{S_1\} \cup MHP(S_1)\})$: If $(r \in varPts(S_1, y))$: $[\![fieldPts(s, r, f) \subseteq varPts(S_2, x)]\!]$ ϕ ϕ	For each reference r in the points-to set of y at S_1, the points-to set of $r.f$ (field f of r) at all statements in $\{\{S_1\} \cup MHP(S_1)\}$ flows into points-to set of x at S_2.
4. Object creation: $x = new\ A()$; *// Say, o_1 is the abstract obj created at this site*	(4.1) $[\![o_1 \in varPts(S_2, x)]\!]$	o_1 is added into points-to set of x at S_2. Like the copy statement, it doesn't affect/use MHP information, as x is a local variable.
5. Method Call: $k = x.bar(args)$	(5.1) $[\![r \in varPts(S_1, x); S_1]\!]$	For each reference r in the points-to set of the receiver x at S_1, we process the call in S_1 based on the receiver type. The above treatment for virtual calls is simplified for static calls; receiver type is known always.
6. Return: *return c; // Say, cFunc is the current function.*	(6.1) $[\![varPts(S_1, c) \subseteq retVal(cFunc)]\!]$ (6.2) $[\![MHP(S_1) \subseteq outMHP(cFunc)]\!]$	(6.1) The points-to set of c at S_1 flows into the $retVal(cFunc)$; defined in Fig. 6. (6.2) The set of statements running in parallel with S_1, may run in parallel with the statements that may be executed immediately after the invocation of the current function, and hence, the set is added to $outMHP(cFunc)$; defined in Fig. 6.

Fig. 8. Constraints for non concurrency-related statements

4.2.3 Concurrency-related Statements. This section shows the constraints that will be generated if S_1 is a concurrency-related statement. Such a statement does not directly lead to an update of

the points-to related maps. However, the MHP information may get impacted due to the points-to information. The constraints are shown in Figure 9, along with a brief explanation. To clearly identify the constraints that depict the dependence of MHP information on points-to information, we use a dashed rectangle to number such constraints; for example, ⌈7.2⌉.

We give additional explanation for the handling of `join`, `syncStart` and `wait` statements. Given a statement s, the predicate $must(s)$ is defined as follows:

$$must(s) = (|callSites(func(s))| == 1 \wedge nonSummaryStmts.\mathsf{has}(x)),$$
where x is the element in $callSites(func(s))$

The map $func(s)$ returns the function containing s, and the *callSites* map is computed by our current analysis. We use the *must* condition checks on points-to sets and statements for computing the MHP information of the successors of `join`, `syncStart`, and `wait` statements:
(i) S_1 is of the form $t.join()$ (Constraint ⌈8.1⌉, Fig. 9): If the statements of the thread object pointed-to by t can be uniquely identified ($must(varPts(t))$ is true), and for any of those statements s, $must(s)$ holds then s will not run in parallel with S_2.
(ii) S_1 is a `syncStart` (Fig. 9), or `wait` (Fig. 10): Constraints ⌈9.1⌉, and ⌈11.3⌉ also use the must-information like ⌈8.1⌉. In addition, Constraint ⌈9.2⌉ computes the *mustRemoved* set in order to use it at the monitor exit. In case of `join` and `wait` statements, we do not compute the *mustRemoved* set as that information is not required later.

Note (I): The computation of the maps *thStmtsHA* and *syncHA* happens similar to *thStmts* and *sync*, respectively, except that they are computed using the points-to information and call-graph of *HA*.

Note (II): Unlike the other predicates (in the conditional constraints), which remain true once they become true, these cardinality based predicates may become false, after they were evaluated to true. For example, a set S (initially empty) may become a singleton set and later may hold more than one element. We follow a disciplined approach to ensure soundness: consider a constraint of the form $cond, C, elseC$. (i) As mentioned before, these cardinality constraints (of the form $|S| == 1$) are always nested inside membership constraints, thus the first time the constraint will be solved is when at least one member is added to the set S, and later more members may be added. Thus, C may be processed before $elseC$, but never the other way round. (ii) Further, C and $elseC$ are chosen such that (a) when executed, each of them will add zero or more elements to the same set (say X), and (b) the set of elements added by $elseC$ are never fewer than those of C. Thus the set X will grow monotonically. See Section 4.5 for a formal correctness argument.

Standalone MHP Analyzer. A subset of our constraints to compute the combined points-to and MHP analyses, can also be used to compute only the MHP analysis, by assuming the availability of points-to analysis (computed via some off-the-shelf analyzer).

Maintaining the Inverse maps. For each statement S_i, the following conditional constraints are added irrespective of the type of S_i. The constraint 1 maintains the symmetry of MHP maps. The remaining inverse map constraints help us to accumulate information across methods.

$$1\ \begin{cases}\text{If } (S_j \in MHP(S_i)):\\ [\![S_i \in MHP(S_j)]\!]\\ \phi\end{cases}$$

$$2\ \begin{cases}\text{If } (S_i \in callSites(f)):\\ [\![f \in funcsAtCallSite(S_i)]\!]\\ \phi\end{cases}$$

Type of S_1	**Generated Constraints** ⌜ ⌝ MHP depends on points-to ☐ independent	**Explanation**
7. Thread start: $t.start()$	7.1 $[\![MHP(S_1) \subseteq MHP(S_2)]\!]$ 7.2 If $(r \in varPts(S_1, t))$: $[\![thStmts(r_c) \subseteq MHP(S_2)]\!]$ ϕ	7.1 The *MHP* info flows from S_1 to S_2. 7.2 When a thread starts, the statements in the *run* method of the started thread may happen in parallel with statements in the current method after the *start*() call. r_c is $typeOf(r)$. The constraints for populating *thStmts* map are explained later in this section (see the note on auxiliary maps).
8. Thread join: $t.join()$	8.1 If $(r \in varPts(S_1, t))$: If $(must(varPts(S_1, t)))$: If $(s \in MHP(S_1))$: If $(thStmtsHA(r_c).\mathsf{has}(s) \wedge must(s))$: ϕ $[\![s \in MHP(S_2)]\!]$ ϕ $[\![MHP(S_1) \subseteq MHP(S_2)]\!]$ ϕ	When a thread is joined, the statements in the *run* method of the started thread will not run in parallel with statements after the $join()$ call. Thus, instead of copying all the statements from $MHP(S_1)$ to $MHP(S_2)$, statements of the *run* method of the joined thread are omitted. To ensure soundness, we exclude a statement $s \in MHP(S_1)$ from $MHP(S_2)$ only if s is guaranteed to be executed solely by the joined thread; we do so by checking that the both the predicates $must(varPts(S_1, t))$ and $must(s)$ hold. Otherwise, all statements in $MHP(S_1)$ are retained in $MHP(S_2)$. Note (i) *thStmtsHA* is the corresponding map of *thStmts* from the helper-analysis *HA*, and (ii) r_c is $typeOf(r)$.
9. Monitor entry: *syncStart p*	9.1 If $(r \in varPts(S_1, p))$: If $(must(varPts(S_1, p)))$: If $(s \in MHP(S_1))$: If $(syncHA(r).\mathsf{has}(s) \wedge must(s))$: ϕ $[\![s \in MHP(S_2)]\!]$ ϕ $[\![MHP(S_1) \subseteq MHP(S_2)]\!]$ ϕ 9.2 If $(r \in varPts(S_1, p))$: If $(must(varPts(S_1, p)))$: If $(s \in MHP(S_1))$: If $(syncHA(r).\mathsf{has}(s) \wedge must(s))$: $[\![s \in mustRemoved(S_1)]\!]$ ϕ ϕ ϕ 9.3 If $(r \in varPts(S_1, p))$: $[\![monStmts(S_1) \subseteq sync(r)]\!]$ ϕ	9.1 When a monitor is entered, the statements which are in the other monitor-blocks synchronized on the same reference as that of the current monitor, will not happen in parallel with any statement in the current monitor-block. To maintain soundness, we exclude a statement $s \in MHP(S_1)$ from to $MHP(S_2)$ by checking that both the predicates $must(varPts(S_1, p))$ and $must(s)$ hold. Otherwise, all statements in $MHP(S_1)$ are retained in $MHP(S_2)$. Note that *syncHA* is the corresponding map of *sync* from the helper-analysis *HA*. 9.2 Any such statement s (that is not added to $MHP(S_2)$) is added to $mustRemoved(S_1)$ to be used later at monitor exit. 9.3 Populate the *sync* map using *monStmts*. The constraints for populating *monStmts* are explained towards the end of this section.
10. Monitor exit: *syncEnd p*	10.1 $[\![(mustRemoved(monPair(S_1)) \cup MHP(S_1)) \subseteq MHP(S_2)]\!]$.	10.1 We add $MHP(S_1)$ to $MHP(S_2)$, along with the statements that were removed at the monitor-entry statement.

Fig. 9. Constraints for concurrency-related statements (start, join, monitor-entry, monitor-exit)

Type of S_1	**Generated Constraints** ⌜ ⌝ MHP depends on points-to ▭ independent	**Explanation**
11. Thread wait: $t.wait()$	⌜1̄,1⌝ If $(r \in varPts(S_1,t))$: { If $(s \in sync(r))$: { $[\![s \in MHP(S_1)]\!]$; ϕ } ; ϕ } ⌜1̄,2⌝ If $(r \in varPts(S_1,t))$: { If $(s \in notifyStmts(r) \cap MHP(S_1))$: { $[\![succ(s) \subseteq MHP(S_2)]\!]$; ϕ } ; ϕ } ⌜1̄,3⌝ If $(r \in varPts(S_1,t))$: { If $(must(varPts(S_1,t)))$: { If $(s \in MHP(S_1))$: { If $(syncHA(r).\mathsf{has}(s) \wedge must(s))$: { ϕ ; $[\![s \in MHP(S_2)]\!]$ } ; ϕ } ; $[\![MHP(S_1) \subseteq MHP(S_2)]\!]$ } ; ϕ }	⌜1̄,1⌝ A call to $wait()$ on a reference r, will release the currently held lock on r. So, statements in all other monitors with the same lock object as that of $wait()$ can run in parallel with S_1. Hence, such statements are added to $MHP(S_1)$. ⌜1̄,2⌝ After receiving a notification from a concurrently executing *notify* or *notifyAll* on r, again the lock is acquired and the execution continues. So, the successor of $wait()$, S_2 executes only after a notification and the corresponding *notify* or *notifyAll* node becomes a logical predecessor of S_2 [43]. Because S_2 resumes execution after the notification finishes, the *succ* of all the concurrently executing (checked using *MHP*.has) *notify* or *notifyAll* (given by *notifyStmts* map) nodes are transferred to $MHP(S_2)$, to account for potential parallel execution. ⌜1̄,3⌝ As the lock is re-acquired before proceeding, S_2 will run in parallel with the statements in $MHP(S_1)$, except the ones executed by taking a lock on r. Again, we need must-information here.
12. Thread notify: $t.notify()$ or $t.notifyAll()$	⌜2̄,1⌝ If $(r \in varPts(S_1,t))$: { $[\![S_1 \in notifyStmts(r)]\!]$; ϕ }	The relevant *notify* and *notifyAll* statements are stored in $notifyStmts(r)$.

Fig. 10. Constraints for concurrency-related Statements (wait, notify)

Building the *monStmts* map. Below are the constraints for populating *monStmts* map in an inter-procedural setting. For every statement s within a monitor entry $mEntry$ and its corresponding exit statement $mExit$,

1 $[\![s \in monStmts(mEntry)]\!]$, if s is a non-invoke statement

2 if s is a static invoke statement, and m is the function being invoked

{ If $(f \in reachableFuncs(m))$: { $[\![funcStmts(f) \subseteq monStmts(mEntry)]\!]$; ϕ }

3 if s is a non-static invoke statement

{ If $(m \in funcsAtCallSite(s))$: { If $(f \in reachableFuncs(m))$: { $[\![funcStmts(f) \subseteq monStmts(mEntry)]\!]$; ϕ } ; ϕ }

Maintaining Auxiliary Maps. In addition to the above discussed constraints, we generate additional conditional constraints at each function call statement S_c to populate some auxiliary

maps. Say, S_c is of the form $k = x.bar(...)$, and let $func(S_c)$ be *foo* (the caller method). We generate the following constraints, all predicated on the condition $r \in varPts(S_c, x)$. Assume that f denotes the function *bar* in the class of r (say, M).

1 Remember the call statement: $[\![S_c \in callSites(f)]\!]$.

2 Populate *thStmts*: If M is a thread-type class and *bar* is its *run* method.

$$\begin{cases} \text{If } (f' \in reachableFuncs(f)): \\ [\![funcStmts(f') \subseteq thStmts(M)]\!] \\ \phi \end{cases}$$

3 Propagate *reachableFuncs* map: $[\![reachableFuncs(f) \subseteq reachableFuncs(foo)]\!]$, all functions reachable from f will now be reachable from *foo*, as well.

4.3 Constraint Solving

The constraints generated in the previous section are solved using a solver similar to that of Oxhøj et al. [45]. The main difference is that we handle additional conditions (cardinality and existence).

We maintain a worklist of constraints, which are ready to be processed. The member constraints are added to the worklist unconditionally. In contrast, the non-member constraints (conditional, FunctionCall, and propagation) are added to the worklist only in response to changes in the contents of some maps, irrespective of the order in which they are initially generated. For example, dependent constraints arising from conditional or FunctionCall constructs are added to the worklist exclusively due to the changes in the truth value of their associated conditions, where such conditions depend on the contents of sets. Similarly, propagation constraints of the form $LHS \subseteq RHS$ are added to the worklist only when new elements are added to the *LHS* set. Consequently, constraints may be added to the worklist (and processed) in an arbitrary order. However, the additive nature of the constraints ensures that the final resulting maps are the same; the interested reader may refer to Oxhøj et al. [45] for details.

4.4 Application of the Analysis on the Example

We now illustrate how *ComPoMHP* runs on the example shown in Fig. 4a. In Fig. 11, we highlight some of the main steps of the combined analysis and the resultant maps in/after each step.

Pre-Analysis HA: Initially, pre-analysis *HA* is invoked to obtain information that is used to perform *must* and existence condition checks (Section 4.2.2). *HA* concludes that `mL` and `tL` (at Lines 9 and 26, respectively), may both point to {`04`, `011`, `028`}. The maps/sets *syncHA*, *thStmtsHA*, *nonSummaryRefs* and *nonSummaryStmts* computed by *HA* using the above points-to information (computed by the rules discussed in Section 4.2.2) are shown in Fig. 11(a). Next, we generate the constraints for the input program and start solving them.

Initially, all the result maps *varPts*, *fieldPts*, *MHP* as well as the auxiliary maps (shown in Fig. 6) are empty. We process each instruction and generate all the required constraints as discussed in Section 4.2. Fig. 11(b) shows an illustrative constraint (for Line 9). Now, we will see how the maps are populated on solving the constraints. Our goal is to demonstrate how the problematic dependency of phase-ordering is resolved using our proposed combined approach.

We illustrate the constraint processing using two of the possible orders as shown in Fig. 11(c): (i) start by processing the non concurrency-related constraints in the worklist till the worklist has only concurrency-related constraints (Fig. 11(d)). (ii) start by processing the concurrency-related constraints in the worklist till the worklist has only non concurrency-related constraints (Fig. 11(e)).

Initially, all the points-to and MHP maps are initialized to empty sets. At the beginning, the member constraints for statements at Lines 3, 5 are the only constraints in the worklist. Processing these constraints results in $\texttt{t} \rightarrow \{\texttt{05}\}; \texttt{w1} \rightarrow \{\texttt{03}\}$ at Line 9. This in turn adds to the worklist, the non-member constraints for statements (that depend on the points-to information of `w1` and `t`). These constraints are then processed sequentially. For example, processing the conditional constraint for

the store operation of Line 4 results in the application of constraint ⟨2.1⟩ and gives O3.k0 ⟶ {O4}. This in turn adds/processes the constraint for statement Line 8 leading to *varPts*(9, mL) ⟶ {O4}. Similarly, adding/processing the constraints for the store operation of Lines 10, 11, 12 does not modify what O3.k0 points-to. Note that these statements have empty MHP at this time. Thus, *varPts*(9, mL) remains unchanged in the for-loop. Similarly, the non-concurrency statements of the methods of T class are also processed. Some of the illustrative maps are shown in Fig. 11(d).

Now, we will process the concurrency-related constraints using the FSPT information computed so far. Processing the constraint for t.start() at Line 6 results in (i) the addition of all statements of T:run method (see the constraints for auxiliary maps in Section 4.2) to *thStmts*(*T*); and (ii) the processing of the constraint ⌊7.2⌉ that results in *MHP*(7) = {25, 26, 27, 28, 29}. This in turn gets propagated to successor *MHP*(8) and then to *MHP*(9). Now, consider the processing of the constraints ⌊9.1⌉, ⌊9.2⌉ for the monitor entry at Line 9 that computes *MHP*(10). The expression *must*(*varPts*(9, mL)) returns true because *varPts*(9, mL) = {O4} is a singleton set, and O4 ∈ *nonSummaryRefs*(9) (from *HA*). The elements of *MHP*(9), which are also present in *syncHA*(O4), are {27, 28, 29}. The expressions *must*(27), *must*(28) and *must*(29) return *true* because their enclosing function T:run is called from only one call-site (Line 6) and Line 6 is a non-summary statement (from *HA*). By solving constraint ⌊9.1⌉ for Line 9, the statements 27, 28, 29 get added to *mustRemoved*(9), and the remaining elements of *MHP*(9) are added to *MHP*(10). After processing constraint ⌊9.1⌉ for Line 9, we obtain the updated maps for *mustRemoved* and *MHP*, as shown in Fig. 11(d).

From symmetry property of MHP maps, we have *MHP*(25) = *MHP*(26) = {7, 8, 9, 10, 11, 12, 14}; *MHP*(27) = *MHP*(28) = *MHP*(29) = {7, 8, 9, 14}. The MHP computation of monitor entry at Line 26 results in processing of constraints ⌊9.1⌉, ⌊9.2⌉. Similar to the computation of *MHP*(9) for the monitor entry in main, *must*(*varPts*(26, tL)) is true and *mustRemoved*(26) = {10, 11, 12}. Hence, *MHP*(27) remains {7, 8, 9, 14}. This is propagated to the MHP maps of successors: *MHP*(28) and *MHP*(29).

Because of the change in MHP maps, the constraints for statements performing heap operations in both main and run methods will be added to the worklist and processed. Thus the effect of MHP computation is reflected in the FSPT computation. Since the statements at lines 10-12 do not run in parallel (~~*MHP*~~) with statements at lines 27-29, the points-to information for none of the variables/fields (including the values of w1.k0, w1.k1, w1.k2, w.k0, w.k1, and w.k2) change, and we reach a fixed point.

Say, we process the concurrency-related statements first as shown in Fig. 11(e). Since all points-to maps are empty initially, the constraints of t.start() statement at Line 6 (which is the origin of concurrency in the program) cannot be added to the worklist, as the points-to set for t is empty. Hence, MHP maps are also empty. Now, performing FSPT using this MHP is the same as starting with FSPT as in Fig. 11(d) and so results in the same FSPT and MHP information.

Final result: Fig. 11(f) shows the final result based on our proposed approach. *ComPoMHP* successfully finds that (i) Lines 10-12 (B1) and 27-29 (B2) *may not* run in parallel and (ii) *varPts*(9, mL) = *varPts*(26, tL) = {O4}. This is in contrast to the non-combined approach discussed in Section 3 that leads to an imprecise result: (i) Lines 10-12 (B1) and 27-29 (B2) *may* run in parallel and (ii) *varPts*(9, mL) = *varPts*(26, tL) = {O4, O11, O28}.

4.5 Correctness

We have proved the correctness argument for the constraint-generation part, and the constraint-solving part separately. We also show that the points-to information and MHP information obtained by *ComPoMHP* are sound. The details can be found in Appendix A.

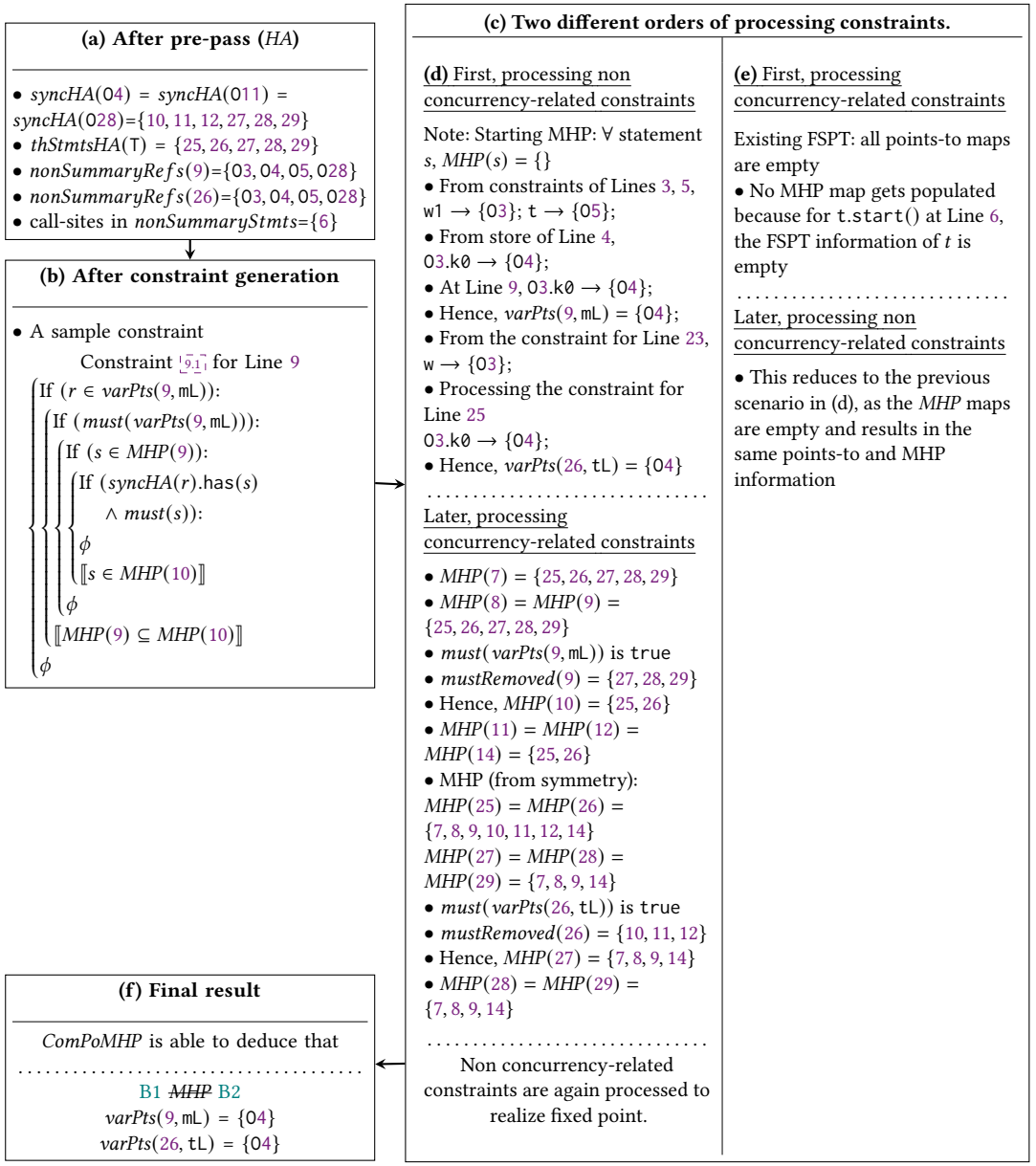


Fig. 11. Detailed run of *ComPoMHP* on the example shown in Fig 4a.

5 Discussion

5.1 Comparability of Context-sensitive Analysis in Terms of Speed and Precision

As is well known [50], many context-sensitive analyses do not scale well with the added complexity of flow-sensitivity, and therefore, researchers (especially those using Doop [9] based framework) have focused mainly on flow-insensitive variants of them. Note that even though they use the SSA

intermediate-representation, which gives flow-sensitivity while updating local variables, the heap update (for store statements) still happens flow-insensitively.

Flow-sensitivity brings in a different dimension of precision, which is incomparable to the precision due to context-sensitivity; the precision depends on the target application. A natural question to ask is if the precision and speed of the Doop based analysis are clearly better to that of a context-insensitive, flow-sensitive analysis like *ComPoMHP*. In Section 7.2, we answer the question in the negative and show that *ComPoMHP* provides an interesting alternative to the context-sensitive flow-insensitive variants.

5.2 Composing Results from Different Conservative Analyses

Points-to analysis is typically applied to clients that fall into two categories:

(i) superset-results-clients (e.g., set of call-graph edges, possibly failing casts), where the client result computed from points-to information is a superset (over-approximate) of the actual run-time behavior. For instance, the call graph derived from points-to data is a superset of the true run-time call graph.

(ii) subset-results-clients (e.g., monomorphic calls), where the result computed from points-to information is a subset (under-approximate) of the actual behavior. For example, the set of monomorphic calls identified from points-to data is a subset of the monomorphic calls that occur at run time.

Consider a client K and two conservative (sound) points-to analyses `X` and `Y`, with corresponding client results K_X and K_Y. These results can be composed depending on the category of K:

(a) superset-results-clients: For such clients, the intersection $K_X \cap K_Y$ remains conservative. Example: Consider the client K to be the "set of call-graph edges": the results (say, K_X and K_Y) derived from any two different variants of points-to analyses for this client, over-approximate the actual call-graph edges, and differ only in false positives. Intersecting these sets may remove some of the spurious edges (the ones that are present only in one set, but not in the other), thereby improving the precision while preserving soundness.

(b) subset-results-clients: For these clients, the union $K_X \cup K_Y$ remains conservative. Example: consider the client K to be the "set of monomorphic calls": the results (say, K_X and K_Y) derived from any two different variants of points-to analyses for this client, may differently under-approximate the actual set of monomorphic calls. Taking the union may reduce the missed cases, again improving precision while preserving soundness.

Such a composition can trivially lead to improved precision. Further, if the two analyses `X` and `Y` differ greatly in their dimensions of analysis (for example, flow-sensitive+context-insensitive vs. flow-insensitive+context-sensitive), then such composition can lead to significant precision gains.

We use this inspiration to combine the results of independent runs of *ComPoMHP* with the publicly available 1-object-sensitive (*D1obj*) and the efficient adaptive-2-object-sensitive+heap (*Da2objH*) analyses of Doop [9]; both of these are flow-insensitive analyses. (see RQ3 in Section 7.2).

5.3 Handling Other Concurrency-related Constructs of Java

We explain the proposed ideas using basic concurrency-related constructs of Java (namely, thread creating, joining, synchronizing, notifying and waiting). Java supports many other constructs, such as `ExecutorService`, `ForkJoinPool`, `Runnable`, `Callable`, and so on. We have handled those constructs by appropriately mapping them (sometimes conservatively) to the constructs described in this paper.

5.4 Scalability Issues with Flow-Sensitive Analysis

This paper presents a constraint-based solution to perform a combined points-to and MHP analysis *ComPoMHP*, and a standalone MHP analysis technique *PoMHP* (Section 7) that can be invoked by providing points-to analysis results computed by any independent points-to analysis scheme. It is

well known [49, 50] that FSPT analysis does not scale well always, and in those cases, we propose to fall back to computing points-to analysis independently, and then invoke the standalone MHP analysis scheme.

5.5 Handling of Libraries

Similar to prior works [9, 52, 53, 57], we also specially handle certain library methods to improve scalability and precision. For example, we skip the known side-effect free (native) methods like `java.lang.Object.hashCode()`, `java.lang.Object.<init>()`, and so on. Similarly, we represent immutable objects like objects of `String`, Wrapper classes like `Integer`, `Double`, and so on, by special objects and do not track the fields within them. For scalability, we handled objects of XML libraries in a field-insensitive manner.

6 Optimizations

Considering the overheads of constraint-based analysis, especially in the context of flow-sensitive analysis, we use a series of optimizations (many of these are known to practitioners) to make the proposed analysis scalable. The main target of our optimizations is to reduce the number of constraints and the amount of information being propagated during the constraint-solving phase.

6.1 Skip Uninteresting Statements

We categorize a statement as *interesting* if it can affect either MHP or points-to information (the 12 statements whose processing is discussed in Section 4.2); all the remaining statements are considered uninteresting and are not included in the set *Stmts* (Fig. 3). Further, among the copy, load, store statements we skip the statements that assign primitive type values.

6.2 Efficient Storage/Retrieval of Flow-Sensitive Maps

We use `HashMap` of Java collections framework to implement all the maps that we used in our analysis. A naive implementation of flow-sensitive maps would maintain multi-level maps leading to additional overheads. We explain the challenge and our solution by using *varPts* as an example. Using the naive scheme, for each statement, *varPts* would return a map of variables to values. This will require an additional hashmap lookup for each access. We avoid this overhead by using a HashMap from a tuple (Statement × Variables) to values. To make it work correctly, we appropriately override the `hashCode` and `equals` methods in the class corresponding to this tuple. Similar design was adopted for all the flow-sensitive maps.

6.3 Propagate Only the Required Fields Inside Each Method

Trivially, in a flow-sensitive analysis, we copy all the fields across all the methods to each program point. However, not all methods access all the fields. So we use a helper analysis to find the accessed fields in each method; this requires understanding the fields accessed in the callees of each method. We obtain these callees from our helper-analysis *HA*'s call graph.

6.4 Sparse Propagation of *varPts* Maps

We do an intra-procedural use-def analysis [35] where we find for each statement, the variable definitions reaching it. We propagate *varPts* from a definition to only the statements where it may be used.

6.5 Efficient Storage of *MHP* Information

An interesting observation in the *must* information calculation is that when determining whether to exclude a statement, the statements in a method with no parallel constructs exhibit the same behavior (either all excluded or all included), because the $must(s)$ depends on the method itself rather than the individual statement. This motivated us to safely treat all statements in a method with no parallel constructs as a single unit in the MHP maps. It is safe because in such a method, all statements share the same MHP except that an invocation statement may augment the MHP of its successors with the parallelism of the invoked method. So, we will not be missing any possible MHP

SNo	Bench	#Stmts	Size (KB)	SJ	WN	EC	Synch	Total Heap Updates	Strong Updates by *ComPoMHP*
1	avrora	21,030	3379	Y	Y	-	16	797	64
2	xalan	42,978	4403	Y	-	-	8	236	72
3	sunflow	10,313	1434	Y	-	-	3	249	106
4	batik	34,625	5427	Y	Y	-	15	599	127
5	graphchi	7571	1638	-	Y	Y	23	206	92
6	zxing	16,039	2662	-	-	Y	2	264	61
7	lusearch	9649	1331	Y	Y	-	16	264	80
8	jme	7896	6554	Y	Y	Y	7	120	55
9	lusearch-fix	9649	1331	Y	Y	-	16	264	80
10	luindex	17,306	1638	Y	Y	-	58	498	167
11	pmd	27,480	13312	-	-	Y	4	579	56
12	lucene	5276	3686	Y	Y	-	7	151	32
13	zookeeper	18,250	7680	Y	-	-	9	600	124

Fig. 12. Static characteristics of the benchmarks and the number of strong updates performed by *ComPoMHP*. Abbreviations used: #Stmts = Number of points-to and concurrency related statements in the reachable application methods of benchmarks, Size = the total size of the application class files in the benchmarks, SJ = Start/Join, WN = Wait/Notify, EC: Executor/Callable, Synch = Synchronizations.

pair. Such a treatment greatly reduces the size of MHP maps and improves efficiency. Note that while counting the number of *MHP* pairs, we would take into account the number of statements inside a method, in the case that it is represented by a single statement.

6.6 Parallelizing the Constraint Generation

We observe that the generation of constraints for each method is independent of one another, and hence, we mark the constraint generation of each method as a parallel task. These tasks were executed in parallel by a fixed pool of threads.

7 Implementation and Evaluation

We have implemented our proposed schemes in the Soot [57] framework. We use the PInter [25] scheme for efficiently generating/solving the constraints. Besides the formal correctness argument (Section 4.5), we also evaluated our code on 50 micro benchmarks covering various corner cases, and manually verified that the generated FSPT and MHP results are correct. Our implementation and the micro benchmarks are available on GitHub [24].

To evaluate different aspects of our analysis, we implemented three variants of our analyses: (i) *ComPoMHP*: combined FSPT and MHP analysis (ii) *PoMHP*: standalone MHP analyzer: FSPT analysis assuming worst-case MHP analysis, followed by MHP analysis (iii) *iPoMHP*: Flow-insensitive points-to analysis followed by MHP analysis. Overall, our implementation spanned a total of 5K lines of Java code.

Note that we do not use the implementation of SPARK+MHP analysis of Li and Verbrugge [29] in Soot, because this MHP implementation fails to analyze any of the benchmarks under consideration, as it requires inlining of all the functions.

We have performed an evaluation using 13 benchmarks on an Intel 2.3 GHz Xeon Gold 5218 system with 100 GB of memory, running Ubuntu 22.04.1 LTS. We took 11 benchmarks from 9.12 (DaCapo-bach) and 23.11 (DaCapo-chopin) versions of DaCapo [6] benchmark suite, and two real-world Java projects from GitHub repositories namely, zookeeper and lucene.

As is standard [20, 54], we used TamiFlex [7] to handle reflection in these benchmarks. From the two versions of DaCapo benchmark suites, the skipped benchmarks are those that did not work with either TamiFlex or our implementation in Soot (errors at different stages).

Fig. 12 presents the list of the evaluated benchmarks, along with some static characteristics of the benchmarks, and the number of strong updates performed by *ComPoMHP*. It shows that all

the benchmarks have different parallelism related constructs. Further, we see that the number of strong updates that could be performed by *ComPoMHP* in each benchmark is significant (geomean 24.8%, median 30.3%).

We perform an evaluation to answer four research questions, classified under two headings: *(a) Utility of the combined analysis (Section 7.1)*: RQ1: What is the impact of *ComPoMHP* on the precision of the obtained MHP and points-to information, compared to *PoMHP*? *(b) Utility of the flow-sensitivity (Section 7.2):* RQ2: What is the impact of *ComPoMHP* on the precision of the obtained MHP and points-to information, compared to *iPoMHP*? RQ3: How does *ComPoMHP* compare to popular Doop based flow-insensitive, context-sensitive analyses?
(c) Cost study of combined analysis (Section 7.3): RQ4: What is the cost implication of *ComPoMHP*?

We show the impact of *ComPoMHP* by using seven clients (i) Points-to set sizes: The sum of sizes of the points-to sets of all variables and fields at the end of each method. (ii) The number of MHP pairs: When a statement s_2 is present in $MHP(s_1)$, we count (s_1, s_2) as an MHP pair. (iii) Call-graph edges: Number of edges in the call graph. (iv) Monomorphic calls: Number of calls where points-to set of receiver points to objects of the same type. (v) Possibly Failing Casts: Number of typecasts that may fail. (vi) The number of synchronizations that can be elided (a synchronization can be safely elided if its entry does not run in parallel with the entry of any other synchronization that shares an aliased lock variable). (vii) The number of possible races (we count an MHP pair (s_1, s_2) as a race if both access same heap and at least one of them is a store).

7.1 Utility of Combined Analysis

(RQ1) What is the impact of *ComPoMHP* on the precision of the obtained MHP and points-to information, compared to *PoMHP*?

Fig. 14 gives a comparison of *ComPoMHP* and *PoMHP*, for all the clients, across the thirteen benchmarks. We see that the gains due to *ComPoMHP* are visible across all the metrics.

Points-to set sizes. We see that across all benchmarks the precision has improved (between 2.3% and 36.6%, geomean 8.2%, median 8.0%); this attests to the importance of performing the two analyses together. We see that batik, jme, xalan, zxing, pmd show significant reduction in the sizes of the points-to sets. Note that this improvement does not always correspond to the reduction in the sizes of the MHP sets. For example, jme shows improvement in both these metrics, but xalan does not. As mentioned before, this is because the exact improvement amount depends on the characteristics of the specific benchmarks that lead to higher (or lower) dependency between the MHP and points-to analysis artifacts.

Number of MHP pairs. We see that the precision in terms of the number of MHP pairs has improved significantly (between 0.0% to 93.0%, geomean 3.0%, median 2.7%); this also attests to the importance of performing the two analyses together. We see that some benchmarks like jme, avrora, batik, sunflow and pmd show large improvements in precision, whereas some other benchmarks like zookeeper, graphchi, luindex, lusearch show much lower improvements.

The improvements in various metrics achieved by *ComPoMHP* over *PoMHP* (and *iPoMHP*, discussed in RQ2) can be attributed to several factors:

(1) The use of *must* information enables strong updates on the heap, leading to improved points-to precision compared to *PoMHP* and *iPoMHP*. Additionally, the flow-sensitivity of variables contributes to better points-to metrics in *ComPoMHP* (and *PoMHP*) relative to *iPoMHP*.
(2) The *must* information is also leveraged to eliminate certain false positives in MHP pairs. This filtering occurs at three points: monitor entry (constraint $\overline{9.1}$), thread join (constraint $\overline{8.1}$), and thread wait (constraint $\overline{11.3}$).
(3) The improvement in MHP metrics can further be attributed to identifying fewer compile-time threads and more precise reachable statements within each thread (*thStmts*). This reduction is

a consequence of more precise points-to information, which itself results from flow-sensitive heap updates – an approach not employed by *PoMHP* and *iPoMHP*.

We find that the exact improvement amount depends on the benchmark: (i) scope of improvement in that benchmark, and (ii) details of the specific benchmarks that lead to higher (or lower) dependency between the MHP and points-to analysis artifacts.

For example, jme benchmark showed a significant reduction in the points-to sizes, and the number of MHP pairs. However, in lucene benchmark, we observed no reduction in the number of MHP pairs. We manually analyzed the lucene benchmark and found that although points-to set sizes reduced overall (due to flow-sensitivity), these improvements did not occur at locations critical for MHP analysis, such as thread start statements or reachable methods within threads. Further, a closer analysis of the *must* information (see Section 4.2) brought an interesting insight: across all the thread join and wait statements, the *must*-condition check evaluation failed for at least one of the objects, or the statement. In the case of monitor entry statements, there were instances where the *must*-condition check evaluated to `true` for some objects, but the corresponding *MHP* sets were empty, leaving no opportunity for further filtering.

We also manually analyzed the jme benchmark. The strong updates in Fig. 12 demonstrate the effect of MHP over FSPT. To evaluate how this MHP-enhanced FSPT further impacted MHP, we examined the number of strong updates in the locations critical for MHP analysis, including thread runs and their reachable methods. We found that 47% of the strong updates in jme occurred in these locations, reducing the reachable methods and parallel statements, explaining the greater improvement in MHP pairs.

Fig. 14 also shows the impact of *ComPoMHP* on the number of call-graph edges (lower the better), the number of identified monomorphic calls (higher the better), and the number of possibly failing casts (lower the better). We can observe that in all the cases, *ComPoMHP* performs better than *PoMHP*. For call-graph edges, monomorphic calls, and failing casts, the improvements varied between 0.9%-24.0% (geomean 4.6%, median 4.7%), 0.0%-3.2% (geomean 0.5%, median 0.1%), and 0.0%-53.6% (geomean 3.1%, median 1.7%), respectively.

The impact of *ComPoMHP* can also be seen on the number of reported possible races and the number of elidable synchronization operations (impact varied between 0.0%-100.0% (geomean 2.5%, median 1.9%) and 0.0%-20.0% (geomean 1.3%, median 0.0%), respectively). As discussed before, the exact impact depends on the specific benchmarks. We have found that most of the synchronization operations are non-redundant, and naturally the scope for their elision is less.

As an illustration of the impact of *ComPoMHP*, Fig. 13 shows a code snippet found in luindex benchmark. The `merge` method in the class `org.apache.lucene.index.ConcurrentMergeScheduler` spawns many threads that share an object of type `org.apache.lucene.index.IndexWriter`. In the method `messageState` of class `IndexWriter`, the field `docWriter` is read, while a store to the same field occurs in the `closeInternal` method (of the same class). Both *PoMHP* and *ComPoMHP* correctly deduce that the base variables of both these operations (at Line 4 of Fig. 13c and Line 4 of Fig. 13b) are aliases (overlapping points-to sets). While *PoMHP* concludes that the load and store operations may execute in parallel and hence marks these accesses for potential data race, *ComPoMHP* determines that these operations cannot occur in parallel, and does not flag a data race. We explain the reasons for the same below:

The `IndexWriter` object is shared among main thread and spawned threads and its methods are coordinated using `wait` and `notifyAll` for synchronization. In *PoMHP*, weak updates on fields (for example, `docWriter`) of the shared `IndexWriter` object result in imprecise points-to sets. Because of this, the analysis cannot precisely identify the objects used in `wait` and `notifyAll` calls, nor can it accurately match specific `notifyAll` operations with the corresponding `wait` calls. Without precise synchronization ordering between the threads, *PoMHP* conservatively treats the load in

```
1 // method in ConcurrentMergeScheduler
2 void merge(IndexWriter x)
3 {
4   ...
5   while (...) {
6     ...
7    t.start(); // these threads may coordinate
       using wait and notifyAll calls inside the
       methods of IndexWriter class
8     ...
9   }
10  ...
11 }
```

(a) Thread origin

```
1 // method in class IndexWriter
2 void messageState() {
3   ...
4   ... = this.docWriter;
5   ...
6 }
```

(b) Field load

```
1 // method in class IndexWriter
2 void closeInternal(...) {
3   ...
4   this.docWriter = ..;
5   ...
6 }
```

(c) Field store

Fig. 13. Snippet of code from luindex (from DaCapo) to illustrate spurious data race

`messageState` and the store in the `closeInternal` as potentially concurrent, leading to a spurious race report. On the other hand, *ComPoMHP* performs strong updates on heap, which in turn leads to precise synchronization ordering between the threads in this program. Consequently, *ComPoMHP* concludes that the load and store operations will not execute in parallel, and hence there is no data race.

Summary. We find that the proposed combined analysis leads to improved MHP and points-to results across all the benchmarks.

7.2 Utility of Flow-sensitivity

(RQ2) What is the impact of *ComPoMHP* on the precision of the obtained MHP and points-to information, compared to *iPoMHP*? We present a comparison between *ComPoMHP* and *iPoMHP* in Fig. 14, by considering the same seven clients used in Section 7.1. We see that compared to *iPoMHP* the precision in *ComPoMHP* has improved across all the clients and benchmarks. The improvement in the points-to set sizes and the number of *MHP* pairs varied between 19.8%-85.1% (geomean 37.8%, median 41.5%) and 0.0%-99.8% (geomean 5.5%, median 3.9%), respectively. For call-graph edges, monomorphic calls, and failing casts, the improvements varied between 0.9%-26.8% (geomean 6.4%, median 5.3%), 0.0%-6.1% (geomean 0.7%, median 0.7%), and 0.0%-74.2% (geomean 20.3%, median 29.0%), respectively. For number of possible races and elidable synchronization operations the improvements varied between 0.0%-100.0% (geomean 5.1%, median 7.8%) and 0.0%-50.0% (geomean 1.3%, median 0.0%) respectively. We see that for most benchmarks (for example, avrora, sunflow, batik and jme) where *ComPoMHP* showed improvements over *PoMHP*, we see improvements over *iPoMHP* as well. This attests to an interesting point that the precision improvement in the MHP information has an important contribution from the FSPT information, and vice-versa.

The comparison between *PoMHP* and *iPoMHP* shows the impact of flow-sensitivity on the points-to information of local variables (on the stack). Note: unlike *PoMHP*, the heap is updated flow-sensitively in *ComPoMHP* (see Fig. 12). These gains can also be seen as the gains we can get if we use an SSA based intermediate representation.

Summary. We find that flow-sensitivity plays an important role in improving the precision of MHP and points-to analysis results.

Bench	Metric	*iPoMHP*	*PoMHP*	*ComPoMHP* (A,B))
avrora	Pt set sizes↓	23155	13543	12771 (44.9, 5.8)
	MHP pairs↓	40675841	40378729	37738335 (7.3, 6.6)
	CG Edges↓	7015	6991	6611 (5.8, 5.5)
	Monomorphic↑	3988	3990	3992 (0.2, 0.1)
	Failing Casts↓	121	112	107 (11.6, 4.5)
	Rem. Synchs↑	3	3	3 (0.0, 0.0)
	Races↓	935	850	828 (11.5, 2.6)
xalan	Pt set sizes↓	61807	41120	31506 (49.1, 23.4)
	MHP pairs↓	519483	511811	499295 (3.9, 2.5)
	CG Edges↓	17127	16918	15468 (9.7, 8.6)
	Monomorphic↑	8213	8280	8384 (2.1, 1.3)
	Failing Casts↓	228	193	159 (30.3, 17.7)
	Rem. Synchs↑	8	8	8 (0.0, 0.0)
	Races↓	0	0	0 (0.0, 0.0)
sunflow	Pt set sizes↓	8814	5279	4859 (44.9, 8.0)
	MHP pairs↓	19568112	19568112	18753734 (4.2, 4.2)
	CG Edges↓	2810	2805	2663 (5.3, 5.1)
	Monomorphic↑	1995	1995	1999 (0.3, 0.3)
	Failing Casts↓	78	65	63 (19.3, 3.1)
	Rem. Synchs↑	2	2	2 (0.0, 0.0)
	Races↓	853	847	847 (0.8, 0.0)
batik	Pt set sizes↓	133735	31482	19984 (85.1, 36.6)
	MHP pairs↓	74199119	66088500	63034974 (15.1, 4.7)
	CG Edges↓	10380	10000	9540 (8.1, 4.6)
	Monomorphic↑	6723	6770	6785 (1.0, 0.3)
	Failing Casts↓	430	366	359 (16.6, 2.0)
	Rem. Synchs↑	4	4	4 (0.0, 0.0)
	Races↓	1799	1662	1448 (19.6, 12.9)
graphchi	Pt set sizes↓	2012	1663	1575 (21.8, 5.3)
	MHP pairs↓	4484720	4090373	4025056 (10.3, 1.6)
	CG Edges↓	1512	1464	1391 (8.1, 5.0)
	Monomorphic↑	1179	1180	1181 (0.2, 0.1)
	Failing Casts↓	11	10	10 (9.1, 0.0)
	Rem. Synchs↑	4	4	4 (0.0, 0.0)
	Races↓	389	381	373 (4.2, 2.1)
zxing	Pt set sizes↓	14059	9420	8231 (41.5, 12.7)
	MHP pairs↓	25502122	25473606	24562354 (3.7, 3.6)
	CG Edges↓	4020	4015	3830 (4.8, 4.7)
	Monomorphic↑	2974	2974	2974 (0.0, 0.0)
	Failing Casts↓	18	12	12 (33.4, 0.0)
	Rem. Synchs↑	2	2	2 (0.0, 0.0)
	Races↓	583	548	538 (7.8, 1.9)
lusearch	Pt set sizes↓	5636	3730	3626 (35.7, 2.8)
	MHP pairs↓	184865	183215	181895 (1.7, 0.8)
	CG Edges↓	2893	2823	2753 (4.9, 2.5)
	Monomorphic↑	1814	1826	1826 (0.7, 0.0)
	Failing Casts↓	30	17	17 (43.4, 0.0)
	Rem. Synchs↑	14	14	14 (0.0, 0.0)
	Races↓	4	4	4 (0.0, 0.0)

Bench	Metric	*iPoMHP*	*PoMHP*	*ComPoMHP* (A,B))
jme	Pt set sizes↓	2194	1533	1135 (48.3, 26.0)
	MHP pairs↓	4446981	178823	12633 (99.8, 93.0)
	CG Edges↓	2115	2036	1549 (26.8, 24.0)
	Monomorphic↑	933	936	936 (0.4, 0.0)
	Failing Casts↓	38	27	27 (29.0, 0.0)
	Rem. Synchs↑	4	5	6 (50.0, 20.0)
	Races↓	237	12	0 (100.0, 100.0)
lusearch-fix	Pt set sizes↓	5633	3731	3628 (35.6, 2.8)
	MHP pairs↓	184865	183215	181895 (1.7, 0.8)
	CG Edges↓	2893	2823	2753 (4.9, 2.5)
	Monomorphic↑	1814	1826	1826 (0.7, 0.0)
	Failing Casts↓	30	17	17 (43.4, 0.0)
	Rem. Synchs↑	14	14	14 (0.0, 0.0)
	Races↓	4	4	4 (0.0, 0.0)
luindex	Pt set sizes↓	6136	5034	4670 (23.9, 7.3)
	MHP pairs↓	51417139	51083175	50298692 (2.2, 1.6)
	CG Edges↓	3601	3579	3430 (4.8, 4.2)
	Monomorphic↑	2496	2505	2515 (0.8, 0.4)
	Failing Casts↓	31	16	8 (74.2, 50.0)
	Rem. Synchs↑	12	12	12 (0.0, 0.0)
	Races↓	2631	2361	2238 (15.0, 5.3)
pmd	Pt set sizes↓	18254	15721	14148 (22.5, 10.1)
	MHP pairs↓	259972590	259312040	249906506 (3.9, 3.7)
	CG Edges↓	10761	10640	10205 (5.2, 4.1)
	Monomorphic↑	6721	6748	6767 (0.7, 0.3)
	Failing Casts↓	275	226	105 (61.9, 53.6)
	Rem. Synchs↑	2	2	2 (0.0, 0.0)
	Races↓	97592	97540	97487 (0.2, 0.1)
lucene	Pt set sizes↓	1913	1570	1535 (19.8, 2.3)
	MHP pairs↓	5817	5817	5817 (0.0, 0.0)
	CG Edges↓	2116	2116	2098 (0.9, 0.9)
	Monomorphic↑	707	707	707 (0.0, 0.0)
	Failing Casts↓	9	9	9 (0.0, 0.0)
	Rem. Synchs↑	6	6	6 (0.0, 0.0)
	Races↓	9	9	4 (55.6, 55.6)
zookeeper	Pt set sizes↓	19677	7470	6823 (65.4, 8.7)
	MHP pairs↓	70425234	55583901	54129569 (23.2, 2.7)
	CG Edges↓	9541	7682	7148 (25.1, 7.0)
	Monomorphic↑	1676	1723	1777 (6.1, 3.2)
	Failing Casts↓	69	62	61 (11.6, 1.7)
	Rem. Synchs↑	0	0	0 (0.0, 0.0)
	Races↓	1155	958	954 (17.5, 0.5)

Fig. 14. Comparison of different metrics over the Points-to information and MHP information across various analyses. (↑ - higher the better, ↓ - lower the better, A- % improvement compared to *iPoMHP*, B-% improvement compared to *PoMHP*)

(RQ3) How does *ComPoMHP* compare to popular Doop based flow-insensitive, context-sensitive analyses? We compared our analysis with the publicly available 1-object-sensitive (*D1obj*) and the efficient adaptive-2-object-sensitive+heap (*Da2objH*) analyses of Doop [9]; both of these are flow-insensitive analyses. We used the publicly available support of the Doop tool to perform these analyses on the older DaCapo-bach benchmark suite; we could not get the support for the new DaCapo-chopin benchmarks, or the other two real-world applications. For all the six

DaCapo benchmarks (luindex, lusearch, pmd, avrora, sunflow, and xalan) that we had discussed earlier, for this evaluation, we picked the DaCapo-bach versions and ran both Doop and *ComPoMHP*.

In Fig. 15, we show a comparison of *ComPoMHP*, *D1obj*, and *Da2objH* by using the three (call-graph edges, possibly failing casts, and monomorphic calls) of the four clients used earlier in this section. We could not compare the information related to the points-to-sets owing to the difference in the representation between our implementation and Doop. For each metric, besides the quantitative measure of how a certain analysis fares, we also show two more columns. A column (labeled $\bigcap$) indicating the common entries between *ComPoMHP* and the Doop based clients and a column (labeled $\sum$) indicating the result of composing (see 5.2) the results of *ComPoMHP* and Doop. A quick look at the common entries indicates that there are many places where flow-sensitivity with MHP (*ComPoMHP*), and context-sensitivity (*D1obj* and *Da2objH*) gave similar precision. However, their different dimensions of precision led to one being precise at some places and the other being precise at other places.

Comparing call-graph edges, and looking at the overall numbers, we see that for avrora, sunflow, lunidex, and lusearch *ComPoMHP* performed better than both *D1obj* and *Da2objH*. In contrast, *D1obj* and *Da2objH* that showed comparable behavior between themselves, performed better than *ComPoMHP* for xalan, and pmd. On average, *ComPoMHP* identified 7.4% fewer call edges than *D1obj* and 8.9% fewer than *Da2objH*. Similarly, in the case of possibly failing casts, we see that *ComPoMHP* performed better for all the benchmarks except sunflow compared to *D1obj*, and all except avrora and sunflow compared to *Da2objH*. However, in the case of monomorphic calls, *D1obj* and *Da2objH* performed better for all the benchmarks except lusearch. On average, *ComPoMHP* identified 47.4% fewer possibly failing casts compared to *D1obj*, and 3.7% fewer compared to *Da2objH*. In the case of monomorphic calls, on average, *ComPoMHP* identified 18.3% more monomorphic calls compared to *D1obj* and 17.8% more compared to *Da2objH*.

We see that while *D1obj* and *Da2objH* both take more or less similar amounts of time, *ComPoMHP* takes significantly less time, except for avrora, xalan, pmd. Also, *ComPoMHP* maintains a similar trend for memory consumption, while *D1obj* and *Da2objH* take more or less similar amounts of memory except for the benchmarks sunflow, xalan and pmd.

This points to an excellent time-precision trade-off, by taking the previous precision related discussion into account.

Another interesting point to note is that composing the results (column labeled Σ) of both *ComPoMHP* and the Doop-based analyses can lead to significant precision gains, by taking advantage of different dimensions of analyses supported by each. For example, compared to *Da2objH*, the composed results lead to a geometric mean improvement of 24.6% (median 29.0%) for call-graph edges, 79.1% (median 80.0%) for possibly failing casts, and 58.5% (median 55.0%) for monomorphic calls. Similarly, compared to *ComPoMHP*, the composed results lead to a 16.2% (median 14.0%) improvement for call-graph edges, 60.7% (median 72.0%) for possibly failing casts, and 27.3% (median 33.0%) for monomorphic calls. Considering the fact that the flow- and context-sensitive analyses are known to be unscalable [49, 50], composing a fast flow-sensitive analysis like *ComPoMHP*, along with a context-sensitive analysis like *Da2objH*, gives us a great option for high precision at an affordable cost.

Summary. In terms of precision, we find that flow-sensitivity (enabled due to the MHP analysis) performs better for some benchmarks and context-sensitivity for others depending on the characteristics of the specific benchmark. The timing comparison shows that our proposed combined analysis gives an option with an excellent time-precision trade-off. Further, one can use both the analyses (*ComPoMHP* and *Da2objH*) to get a major boost in precision.

As is well known, context-insensitivity can get a hit (in terms of precision) when programs have many methods that are invoked from multiple call sites, or those involving a large number of

SNo	Bench Name	Analysis type	CG Edges			Possibly failing casts			Monomorphic calls			Analysis Time(sec) (↓ better)	Analysis Memory(MB) (↓ better)
			Cnt(↓)	⋂	∑	Cnt(↓)	⋂	∑	Cnt(↑)	⋂	∑		
1	avrora	*D1obj*	6884	4931	4931	96	42	42	3388	2377	4503	1267	2058
		ComPoMHP	5649			64			3492			1955	2573
		Da2objH	6912	4931	4931	41	18	18	3391	2381	4502	1185	2057
2	sunflow	*D1obj*	3796	2343	2343	49	35	35	1615	900	2634	1213	3081
		ComPoMHP	2526			60			1919			19	575
		Da2objH	3797	2344	2344	12	3	3	1623	902	2640	1236	2057
3	xalan	*D1obj*	12579	10932	10932	259	48	48	4421	3374	9342	2617	4105
		ComPoMHP	15025			156			8295			4962	8446
		Da2objH	12551	10959	10959	228	50	50	4437	3395	9337	2747	5129
4	luindex	*D1obj*	4247	2950	2950	93	2	2	2498	1429	3584	1672	2057
		ComPoMHP	3430			8			2515			225	1189
		Da2objH	4231	2954	2954	51	2	2	2500	1437	3578	611	2057
5	lusearch	*D1obj*	3818	2369	2369	86	4	4	1863	977	2712	603	2057
		ComPoMHP	2753			17			1826			8	470
		Da2objH	3710	2361	2361	17	3	3	1830	977	2679	596	2057
6	pmd	*D1obj*	7557	6550	6550	278	13	13	3629	2636	7760	1218	3081
		ComPoMHP	10205			105			6767			2805	3799
		Da2objH	8238	7050	7050	278	16	16	3906	2954	7719	1260	5129

Fig. 15. Comparison of Points-to information of *ComPoMHP* vs 1-object-sensitive (*D1obj*) and adaptive-2-object-sensitive+heap (*Da2objH*) analyses. The "⋂" column indicates the common (overlapping) entries with *ComPoMHP*. The "∑" column indicates the effect of using the individual Doop Based analysis along with *ComPoMHP*. Abbreviations: Cnt = count, CG = Call Graph, ↓: lower the better, ↑: higher the better.

polymorphic calls with multiple possible receiver types. Similarly, flow-insensitivity can get a hit when control-flow information is critical – for instance, in applications with frequent variable/field reassignments, where results may be overly conservative in the absence of MHP analysis. Thus, the choice between these sensitivities depends on the specific dimension in which precision is most needed. Our experience shows that the benchmarks have a mix of such features, thus reinforcing the importance of both. As demonstrated above, we can get significantly better results by using both the analyses (*ComPoMHP* and *Da2objH*).

7.3 Cost of Combined Analysis

(RQ4) What is the cost implication of *ComPoMHP***?** To understand the cost implication of the combined nature of *ComPoMHP*, we compare the analysis time and the memory consumption of *ComPoMHP* (combined) and *PoMHP* (non-combined) in Fig. 16 and Fig. 17, respectively. The figures show that *ComPoMHP* clearly takes more time (geomean 5.5×, median 2.6×) and more memory (geomean 5.8×, median 4.3×) than *PoMHP*. This is mainly due to the flow-sensitive nature of the points-to analysis component of *ComPoMHP*. Note that even though *PoMHP* includes a flow-sensitive analysis component, it assumes conservative MHP information, and thus all the heap updates are "weak" [46] and hence flow-insensitive. Despite the extra cost compared to *PoMHP*, *ComPoMHP* takes reasonably comparable time and memory (significantly less for some benchmarks) compared to a context-sensitive flow-insensitive analysis (see Fig. 15). To establish our hypothesis that the main cause of overhead is the flow-sensitivity, we ran a simpler version of *PoMHP*, named *PoA*, which is a standalone points-to analyzer: FSPT analysis assuming worst-case MHP analysis. *PoMHP* and *PoA* exhibit similar memory usage, supporting the hypothesis that additional memory is due to tracking flow-sensitive values at each program point. We found that the additional time taken by *PoMHP* (compared to *PoA*) was comparable to the difference between *ComPoMHP* and *PoMHP*. This observation discourages us from invoking *PoMHP* more than once

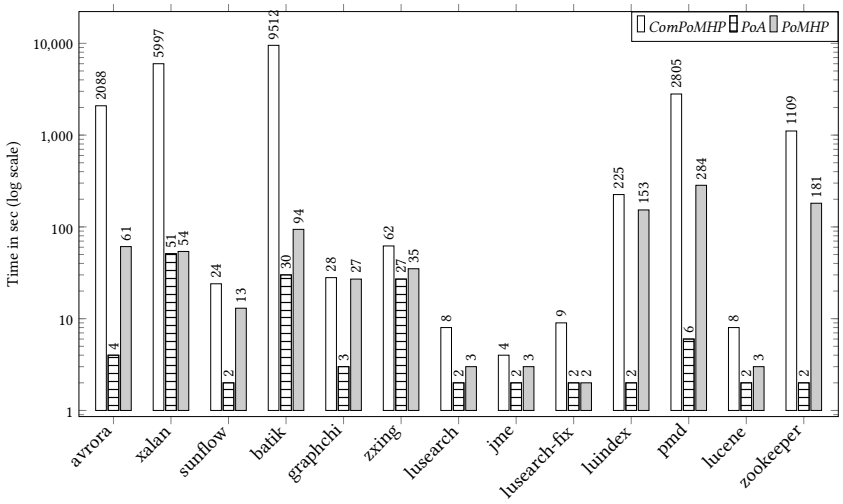


Fig. 16. Time comparison; lower the better.

(by using the results of previous instance in the next) because, due to the availability of MHP information, after the first round, *PoMHP* can do more flow-sensitive updates, and not see any time gains.

Looking at the analysis time and memory usage values, it is evident that both metrics follow a similar trend across the benchmarks. Benchmarks requiring longer analysis time also tend to consume more memory. In general, larger benchmarks (for example, xalan and batik) exhibit higher time and memory overhead. Moreover, benchmarks involving more statements in parallelism experience greater overhead due to the larger size of the *MHP* maps. For instance, although batik is smaller than xalan (see the #Stmts column in Fig. 12), it requires more analysis time because it has a larger number of *MHP* pairs compared to xalan (see Fig. 14).

Summary. We find that the combined analysis takes a hit on the execution time and memory usage when compared to *PoMHP*, but mainly due to the flow-sensitive component therein. But, interestingly the overhead is reasonably comparable to a context-sensitive and flow-insensitive analysis. We conclude that, if the time budget is very tight to obtain MHP information we suggest the usage of *PoMHP*. And if the goal is to obtain more precise points-to and MHP information, then we suggest the usage of *ComPoMHP*. Further, if we can afford more time, then we can get more precise results by composing the results of *ComPoMHP* with any off-the-shelf flow-insensitive context-sensitive analysis results.

8 Threats to Validity

We now describe a few threats to the conclusions we have drawn in our evaluation, and discuss how we have attempted to mitigate the same.

Benchmark Selection. Our results are based on the specific set of benchmarks we used for evaluation. Hence, the findings are tied to this selection, and may not necessarily hold for other types of programs or domains. We tried to mitigate this common external threat by including programs from well-known benchmark suites as well as real-world applications.

Soundness and Precision of *ComPoMHP*. As is standard, our implementation uses TamiFlex to resolve dynamic features like reflection, callbacks and so on. Thus, even though our proposed

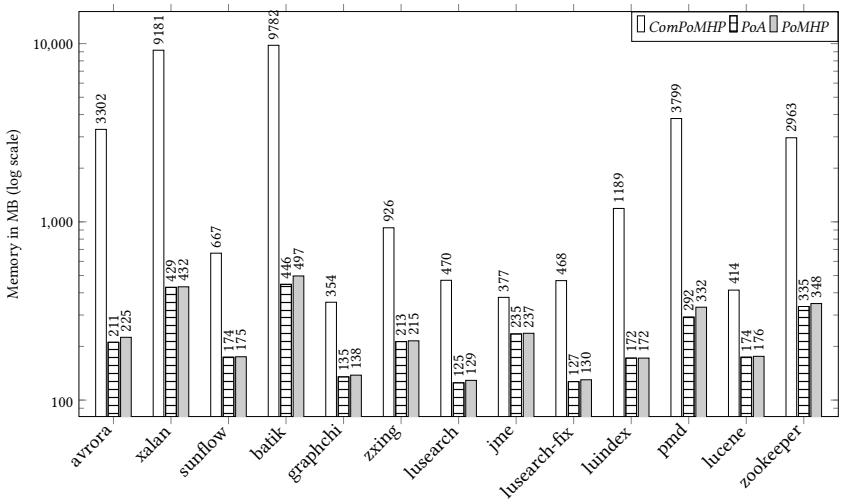


Fig. 17. Memory comparison; lower the better.

combined analysis is sound (as discussed in Section 4.5), our presented evaluation is soundy [33]. This is a standard and unavoidable internal threat to validity for analysis of Java programs with such dynamic features.

A second internal threat arises from the fact that our proposed approach uses a pre-analysis (*HA*) to compute non-summary statements and objects. Naturally, its precision will impact the precision of the results of *ComPoMHP*. For our evaluation, we have used a simple pre-analysis (based on *PoA*). Using a more powerful *HA* will only lead to further improved results.

9 Related Work

Flow-sensitive Points-to Analysis and MHP Analysis for Parallel Programs. Prior works like that of De and D'Souza [15], Toussi and Rasoolzadegan [56] propose FSPT analysis for Java, which can perform strong updates on heap; but they do not consider the presence of threads in the programs. Rugina and Rinard [46] propose a pointer analysis for multi-threaded C programs. They build an interference graph to understand the interference of threads for C programs and use this information to perform FSPT analysis. Salcianu and Rinard [47] propose a scheme to perform pointer and escape analysis for multi-threaded programs. In this process, they maintain a parallel-interaction-graph which includes points-to escape information, as well as information on threads that may be started in the current analysis region and use this information to improve the precision of escape information. Unlike us, they use a pre-built call graph and further, they do not consider any of the challenging concurrency constructs like join(), monitor blocks, wait, notify, notifyAll and, hence they do not compute/maintain precise MHP information.

Naumovich et al. [43] were the first to propose an MHP analysis for Java programs. Barik [5], Chen et al. [11], and Li and Verbrugge [29] presented extensions to the work of Naumovich et al. to make the analysis more precise and efficient. All these analyses require pre-computed points-to information and cannot handle arbitrary non-inlinable function calls. There have also been tools [22, 48] that perform these two analyses one after the other.

In contrast to all these works, we present a scheme to perform FSPT analysis and MHP analysis together and our scheme has no requirement of inlining all the functions. Further, we do not require a pre-computed call-graph either.

Constraint-based Analysis and Solutions. There have been many prior works that use constraints to perform program analysis. Andersen [1] proposed a flow-insensitive, context-insensitive constraint based points-to analysis for C. Our chosen representation of constraints is similar to that of Oxhøj et al. [45], who propose a type inference algorithm where the type information is represented as inclusion constraints and solved to obtain the final types.

Recently many Doop [9]-based analyses have been proposed that specify the analysis as a sequence of Datalog rules. The underlying Datalog engine is then invoked to derive the results. However, prior work [50] has shown that flow-sensitive analysis does not scale well in their system. Hence, we implemented our scheme using the traditional subset based constraints and find that our scheme scales reasonably well.

10 Conclusion

MHP analysis and points-to analysis form two fundamental analyses for parallel programs written in OO languages like Java. In this work, we showed that there is a phase-ordering relation between MHP and FSPT analyses. We proposed a combined FSPT and MHP analysis, represented as inclusion-based constraints. Our scheme can also be used to derive a practical inclusion constraint based solution for MHP analysis, given any conservative points-to analysis results. We performed an elaborate evaluation and showed that the combined analysis improves the precision of both MHP and points-to analysis results, compared to a scheme where these phases are done one after the other. We showed that given points-to information, the derived MHP analysis can be used to compute the MHP information on real-world benchmarks with a very low overhead in terms of analysis time. We showed that flow-sensitivity with MHP can be a scalable alternative to flow-insensitive+context-sensitive analyses (like the ones implemented in the Doop [9] framework) for parallel programs. Furthermore, by composing both the variants, we achieve substantial precision gains, making this approach highly promising for practical parallel program analysis.

Acknowledgments

This work is partially supported by the SERB CRG (sanction number CRG/2022/006971) and IBM CAS project 1156. The first author would also like to acknowledge the financial support given by the Prime Minister's Research Fellowship (PMRF), Ministry of Education, Government of India [Fellowship ID: 2500911], and IIT Madras Women Leading IITM (WLI) grant [for the year 2026].

References

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. Datalogisk Institut, Københavns Universitet.

[2] Ken Arnold, James Gosling, and David Holmes. 2005. *The Java programming language.* Addison Wesley Professional.

[3] Gogul Balakrishnan and Thomas Reps. 2006. Recency-Abstraction for heap-allocated storage. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) *(SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 221–239. https://doi.org/10.1007/11823230_15

[4] Vasanth Balasundaram and Ken Kennedy. 1989. Compile-Time Detection of Race Conditions in a Parallel Program. In *Proceedings of the 3rd International Conference on Supercomputing* (Crete, Greece) *(ICS '89)*. Association for Computing Machinery, New York, NY, USA, 175–185. https://doi.org/10.1145/318789.318809

[5] Rajkishore Barik. 2005. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, 152–169. https://doi.org/10.1007/978-3-540-69330-7_11

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of*

the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[8] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. 1991. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. Association for Computing Machinery, New York, NY, USA, 122–131. https://doi.org/10.1145/106972.106986

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[10] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1126–1140. https://doi.org/10.1145/3453483.3454099

[11] Congming Chen, Wei Huo, Lung Li, Xiaobing Feng, and Kai Xing. 2012. Can We Make It Faster? Efficient May-Happen-in-Parallel Analysis Revisited. *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 59–64. https://doi.org/10.1109/PDCAT.2012.59

[12] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1–19.

[13] Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 181–196. https://doi.org/10.1145/201059.201061

[14] Jack W. Davidson, Gary S. Tyson, David B. Whalley, and Prasad A. Kulkarni. 2007. Evaluating Heuristic Optimization Phase Order Search Algorithms. In *International Symposium on Code Generation and Optimization (CGO'07)*. 157–169. https://doi.org/10.1109/CGO.2007.9

[15] Arnab De and Deepak D'Souza. 2012. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 665–687.

[16] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[17] Antonio E. Flores-Montoya, Elvira Albert, and Samir Genaim. 2013. May-Happen-in-Parallel Based Deadlock Analysis for Concurrent Objects. In *Formal Techniques for Distributed Systems*, Dirk Beyer and Michele Boreale (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 273–288.

[18] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[19] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 226–238. https://doi.org/10.1145/1480881.1480911

[20] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. *IEEE Transactions on Software Engineering* 49, 2 (Feb 2023), 719–742. https://doi.org/10.1109/TSE.2022.3162236

[21] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. 2008. *The C# Programming Language* (3rd ed.). Addison-Wesley Professional.

[22] Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-Based Races in Android Apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 257–270. https://doi.org/10.1145/3173162.3173173

[23] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. 2020. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *Verified Software. Theories, Tools, and Experiments*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer International Publishing, Cham, 1–18.

[24] Ramya Kasaraneni. 2026. Combined analysis. https://github.com/ramyakasar/combinedanalysis.git.

[25] Ramya Kasaraneni and V. Krishna Nandivada. 2026. Compact Representation and Interleaved Solving for Scalable Constraint-Based Points-to Analysis. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction* (Sydney, NSW, Australia) *(CC '26)*. Association for Computing Machinery, New York, NY, USA, 205–217. https://doi.org/10.1145/3771775.3786280

[26] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press, USA.

[27] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. 2003. Finding Effective Optimization Phase Sequences. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (San Diego, California, USA) *(LCTES '03)*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/780732.780735

[28] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/503272.503298

[29] Lin Li and Clark Verbrugge. 2005. A Practical MHP Information Analysis for Concurrent Java Programs. In *Languages and Compilers for High Performance Computing*, Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–208.

[30] Yuan Lin. 2005. Static Nonconcurrency Analysis of OpenMP Programs. In *Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming* (Eugene, OR, USA) *(IWOMP'05/IWOMP'06)*. Springer-Verlag, Berlin, Heidelberg, 36–50.

[31] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3192366.3192390

[32] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When Threads Meet Events: Efficient and Precise Static Race Detection with Origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 725–739. https://doi.org/10.1145/3453483.3454073

[33] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[34] A. Samuel Moses and V. Krishna Nandivada. 2026. Practical MHP Analysis for Java. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction* (Sydney, NSW, Australia) *(CC '26)*. Association for Computing Machinery, New York, NY, USA, 218–229. https://doi.org/10.1145/3771775.3786279

[35] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[36] Mayur Naik and Alex Aiken. 2007. Conditional Must Not Aliasing for Static Race Detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/1190216.1190265

[37] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[38] Mangala Gowri Nanda and S. Ramesh. 2003. Pointer Analysis of Multithreaded Java Programs. *Proceedings of the 2003 ACM Symposium on Applied Computing*, 1068–1075. https://doi.org/10.1145/952532.952741

[39] V. Krishna Nandivada and David Detlefs. 2005. Compile-Time Concurrent Marking Write Barrier Removal. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 37–48. https://doi.org/10.1109/CGO.2005.12

[40] V. Krishna Nandivada and Jens Palsberg. 2006. SARA: Combining Stack Allocation and Register Allocation. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3923)*, Alan Mycroft and Andreas Zeller (Eds.). Springer, 232–246.

[41] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (apr 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[42] G. Naumovich, G.S. Avrunin, and L.A. Clarke. 1999. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 399–410. https://doi.org/10.1145/302405.302663

[43] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Toulouse, France) *(ESEC/FSE-7)*.

Springer-Verlag, Berlin, Heidelberg, 338–354.
[44] Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala: Updated for Scala 2.12* (3rd ed.). Artima Incorporation, Sunnyvale, CA, USA.
[45] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. 1992. Making type inference practical. In *ECOOP '92 European Conference on Object-Oriented Programming*, Ole Lehrmann Madsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–349.
[46] Radu Rugina and Martin Rinard. 1999. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/301618.301645
[47] Alexandru Salcianu and Martin Rinard. 2001. Pointer and Escape Analysis for Multithreaded Programs. *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 12–23. https://doi.org/10.1145/379539.379553
[48] Navid Salehnamadi, Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. ER Catcher: A Static Analysis Framework for Accurate and Scalable Event-Race Detection in Android. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 324–335.
[49] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (apr 2015), 1–69. https://doi.org/10.1561/2500000014
[50] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390
[51] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. *Proceedings of the 14th International Symposium on Code Generation and Optimization, CGO 2016* (2 2016), 160–170. https://doi.org/10.1145/2854038.2854043
[52] The WALA Team. 2025. WALA: T.J. Watson Libraries for Analysis. https://github.com/wala/WALA. GitHub repository.
[53] Manas Thakur and V. Krishna Nandivada. 2019. Compare less, defer more: scaling value-contexts based whole-program heap analyses. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/3302516.3307359
[54] Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3377555.3377902
[55] Sid-Ahmed-Ali Touati and Denis Barthou. 2006. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers* (Ischia, Italy) *(CF '06)*. Association for Computing Machinery, New York, NY, USA, 147–156. https://doi.org/10.1145/1128022.1128042
[56] Hamid A. Toussi and Abbas Rasoolzadegan. 2014. Flow-sensitive points-to analysis for Java programs using BDDs. In *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*. 380–386. https://doi.org/10.1109/ICCKE.2014.6993367
[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.
[58] Steven R. Vegdahl. 1982. Phase Coupling and Constant Generation in an Optimizing Microcode Compiler. *SIGMICRO Newsl.* 13, 4 (oct 1982), 125–133. https://doi.org/10.1145/1014194.800942
[59] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 187–206. https://doi.org/10.1145/320384.320400
[60] Jiang Wu, Jianjun Xu, Xiankai Meng, Haoyu Zhang, Zhuo Zhang, and Long Li. 2020. Compilation Optimization Pass Selection Using Gate Graph Attention Neural Network for Reliability Improvement. *IEEE Access* 8 (2020), 150422–150434. https://doi.org/10.1109/ACCESS.2020.3016758
[61] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. 2005. A Model-Based Framework: An Approach for Profit-Driven Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, USA, 317–327. https://doi.org/10.1109/CGO.2005.2

A Correctness

We now present an argument that the points-to information and MHP information obtained by *ComPoMHP* are sound. For soundness, we first focus on the possible issues that may arise due to

the new conditional constraints involving cardinality conditions. Consider a conditional constraint involving cardinality condition $|S| == 1$ and the then constraint C_1 and the else constraint C_2 as shown below:

$$\begin{cases} \text{If } (|S| == 1): \\ C_1 \\ C_2 \end{cases}$$

Since the constraints can be processed in any order, we may have processed C_1, when the first element got added to S, and if later one or more elements get added to S, then the condition becomes false and C_2 will also be processed. But since $|S| > 1$, it is expected that C_2 alone should have been processed. This is unlike other regular inclusion conditions (of the form $O_i \in A$) in conditional constraints, where the condition will never change from true to false. We now present an argument that such a scenario with cardinality conditions, does not affect the precision and soundness of our solution, because of the way in which we use these cardinality constraints.

Lemma A.1. *The cardinality conditions in the constraints do not affect the soundness of the solution.*

Proof. (*Sketch*) In our formulation, a cardinality condition *cond* may only be of the form $|S| == 1$, and may appear only in conditional constraints of the following form, where $X \subseteq Z$:

$$\begin{cases} \text{If } (cond): \\ [\![X \subseteq Y]\!] \\ [\![Z \subseteq Y]\!] \end{cases}$$

Thus, even if the condition turns from true to false, no unwanted elements flow into Y, nor any expected element is missing from Y. Hence, the soundness of the analysis isn't affected. We now detail the proof for the same.

Consider a set S, and assume that the following cardinality constraint C_0, where $X \subseteq Z$, and S is a set, has been triggered to be processed:

$$\begin{cases} \text{If } (|S| == 1): \\ [\![X \subseteq Y]\!] \\ [\![Z \subseteq Y]\!] \end{cases}$$

Say, currently $|S| = k$. We may get unsound results if the truth value of the cardinality condition changes from true to false and the set of elements of Y decreases (instead of increasing monotonically).

We now show that such a scenario is guaranteed to not occur, by considering the three possibilities based on the value of k: $k = 0$, $k = 1$, and $k > 1$.

(1) Case $k = 0$: That is, S is empty. Such a conditional constraint is always nested inside a membership constraint on S, and is triggered only when an element gets added to set S. Hence $k = 0$ is a contradiction.
(2) Case, $k = 1$. The cardinality condition becomes true and the elements of X get added to Y. Now, say, in future, $|S|$ becomes > 1 (due to some element/s getting added to S) and the condition becomes false:
   (a) The constraint C_0 gets triggered again and now the else part gets evaluated which adds some additional elements to Y (as $X \subseteq Z$). This does not lead to any unsoundness, as the set of elements of Y has only monotonically increased (not decreased).
(3) Case k>1: The cardinality condition can never become true in future.

□

Correctness: Constraint Generation

Theorem A.2. *As is standard, each runtime object O_r has a corresponding abstract object O_a; say, we represent this relation by $O_a = \rho(O_r)$. The proposed combined analysis has the following three properties:*

(1) For any given variable v, at runtime if at statement s, v points to an object O_r, then $\rho(O_r) \in varPts(s, v)$.
(2) For any runtime object O_{r_1}, if at statement s, $O_{r_1}.f$ points to an object O_{r_2}, then $\rho(O_{r_2}) \in fieldPts(s, \rho(O_{r_1}), f)$.
(3) During execution, if a statement S_1 may run in parallel with S_2 then $S_2 \in MHP(S_1)$ and $S_1 \in MHP(S_2)$.

Proof. *(Sketch)* The proof can be obtained via contradiction that if any of the properties is violated, then the corresponding *varPts* or *fieldPts* or *MHP* maps must have certain missing elements at some statement S. This in turn would indicate that the statement that should have added the element was not processed/present in the program and hence a contradiction. This can be obtained by doing an induction on the sequence of statements and performing a case-analysis on each of the possible forms of the additional statement considered at the induction basis and the inductive step. We can obtain such a result because, the points-to/MHP information reaching the successor of any statement S is only due to the impact of S or due to the impact of the statements running in parallel with S ($MHP(S)$), and our proposed constraints represent all such relations. We now detail the proof.

The points-to related constraints (shown in Fig. 8) are all standard except $\langle \underset{\sim}{3.1} \rangle$ and $\langle \underset{\sim}{2.1} \rangle$, which use *MHP* information. So, it suffices to prove the correctness of *MHP* generating constraints, along with these two constraints.

Consider a finite sequence of statements *Seq*. Let the i^{th} statement in *Seq* be denoted as S_i.
We will prove the correctness of the constraints by induction on the number of statements processed in *Seq*.

Base case. If $n == 0$, The maps *varPts*, *fieldPts*, *MHP* contain the initial values and hence are correct.

Induction hypothesis. Let us assume that there exists $n \geq 0$, such that the Theorem A.2 holds for the first sequence of n statements in *Seq*. That is, the *varPts*, *fieldPts*, *MHP* at S_i ($i \leq n$) are correct.

Induction step. We will now show that if the induction hypothesis holds, then the theorem will hold for any successor S_x of S_n. For any of the concurrency-related statements, they themselves do not affect the *varPts*, *fieldPts* maps; thus, the theorems A.2[1,2] are true for S_x as they are true for Seq_n. We will focus on the correctness of *MHP* maps for these statements. Similarly, for the load and store statements (rules $\langle \underset{\sim}{3.1} \rangle$ and $\langle \underset{\sim}{2.1} \rangle$), we will only focus on the correctness of the *varPts*, *fieldPts* maps.
First, we will prove the correctness of the *MHP* maps (by contradiction), by focusing on the concurrent statements processed in Fig. 10. Say, we claim that there exists a statement s which runs in parallel with S_x during run time, but $s \notin MHP(S_x)$.

(1) Say, S_n is a thread-start statement.
Since s runs in parallel with S_x, s should either
  (a) run in parallel with at least one of its immediate CFG predecessors, or
  (b) be added by the thread-start statement during run time, or
  (c) S_x is a *wait* instruction and s started running in parallel with it, after the wait instruction released the held lock.

For case (c), since rule 11.1 ensures that all the monitor statements are added to the *MHP* set of S_x, it cannot be true. For cases (a) and (b), based on the induction hypothesis, the possible options for the claim to be true do not arise, due to our handling of S_n as described below.

(i) If s runs in parallel with S_n, then according to the induction hypothesis $s \in MHP(S_n)$. If $s \in MHP(S_n)$ then as per constraint 7.1, $s \in MHP(S_x)$. A contradiction.

(ii) Say, s is added by the start statement, but $s \notin MHP(S_x)$. Then it means that $s \notin thStmts$ map. But constraint 7.2 ensures that if s is present in the run method of the thread, or any of its reachable methods, then s is present in *thStmts*. A contradiction.

(2) Say, S_n is a thread join statement (of the form $t.join()$).
Since s runs in parallel with S_x, and S_n is not a thread-create statement, s should either (a) have been running in parallel with S_n (an immediate CFG predecessor) at run time, or (b) S_x is a *wait* instruction and s started running in parallel with it, after the wait instruction released the held lock. The analysis for (b) is similar to the analysis of case 1(c).
We now analyze case (a):
By the induction hypothesis, we know that $s \in MHP(S_n)$. If $s \in MHP(S_n)$ and $s \notin MHP(S_x)$ $\Rightarrow s$ is in the set of statements to be not carried forward from $MHP(S_n)$ to $MHP(S_x)$. It implies that as per constraint 8.1, the predicates involving *must*, and *thStmtsHA* must have been true.
Recall that the must conditions for a set of references, and a statement, respectively, are as follows.
For a points-to set A,

$$must(A) = (|A| == 1 \wedge nonSummaryRefs(S_1).\mathsf{has}(x)), \text{ where } x \text{ is the element in } A$$

Given a statement s,

$$must(s) = (|callSites(func(s))| == 1 \wedge nonSummaryStmts.\mathsf{has}(x)),$$
where x is the element in $callSites(func(s))$

We get $nonSummaryRefs$, *thStmtsHA* and $nonSummaryStmts$ from the helper-analysis *HA* which computes a conservative over-approximation, and is therefore sound.
From the Lemma A.1, processing constraints involving cardinality conditions does not lead to unsoundness.
This shows that s is guaranteed to be removed from the MHP information, it means that s indeed cannot run in parallel with S_x, which is a contradiction to the claim that s runs in parallel with S_x at run time.

(3) Say, S_n is a *syncStart* statement (of the form *syncStart p*). The proof for this case follows an argument very similar to case 2 (join statement) above. The main difference is that rule 9.1 explicitly stores the *mustRemoved* information and it uses *syncHA*, instead of *thStmtsHA*.

(4) Say, S_n is a *syncEnd* statement (of the form *syncEnd p*). Two sub-cases arise:

(a) either s was running in parallel with S_n (an immediate CFG predecessor of S_x), or s was running in parallel with the corresponding monitor-entry statement. Both these lead to a contradiction, as per rule 10.1, $MHP(S_x)$ should have included s in either case.

(b) S_x is a *wait* instruction and s started running in parallel with it, after the wait instruction released the held lock. This leads to a contradiction using the same argument given in 1(c).

(5) Say, S_n is a *notify* or *notifyAll* statement. This case is trivial, as the rule 12.1 does not do any special handling of MHP information, and simply makes sure that $MHP(S_n) \subseteq MHP(S_x)$. In addition, this rule helps update the *notifyStmts* map.

(6) Say, S_n is a *wait* statement of the form $t.wait()$. We will again prove the correctness of the *MHP* map by contradiction. Say, we claim that there exists a statement s which runs in parallel with S_x during run time, but $s \notin MHP(S_x)$. There are two sub-cases:

(a) s runs in parallel with a notify (or notifyAll) statement matching the wait statement. Since rule ⌊1,2⌋ is guaranteed to be applied, it leads to a contradiction.
(b) s is not added to $MHP(S_x)$ due to rule ⌊1,3⌋. The analysis for this case follows the same logic as that of case 2, and will lead to a contradiction.

We now will prove the correctness of the *varPts* and *fieldPts* maps (by contradiction) by focusing on the load and store statements defined in Fig. 8. Note that the remaining statements are standard and do not get impacted by MHP information, as they only modify the local variables - not shared across the threads.

(1) Say, S_n is a load statement of the form $a = b.f$.
Say, at runtime, variable a may point to O_1 at S_x, but $varPts(S_x, a)$ does not include $\rho(O_1)$.
Say, at runtime just before executing S_n, b points to O_b and $O_b.f$ points to O_1 – must have been set by the current thread or one of the threads running in parallel. Based on the induction hypothesis, $\rho(O_b) \in varPts(S_n, b)$, and $\rho(O_1) \in fieldPts(S_n, \rho(O_b), f) \cup_{\forall s \in MHP(S_n)} fieldPts(s, O_b, f)$. Rule ⟨3,1⟩ implies that $\rho(O_1) \in varPts(S_x, a)$. A contradiction.
(2) Say, S_n is a store statement, of the form $a.f = b$.
Say, at runtime, variable b may point to O_b, variable a may point to O_a, and $O_a.f$ points to O_1 at S_x, but $\rho(O_1) \notin \cup_{r \in varPts(S_x,a)} fieldPts(S_x, r, f)$. Based on induction hypothesis, $\rho(O_a) \in varPts(S_n, a)$, and $\rho(O_b) \in varPts(S_n, b)$. Two sub-cases arise:
   (a) $O_b = O_1$.
   Rule ⟨2,1⟩ (irrespective of the condition) ensures that $\rho(O_1) \in fieldPts(S_x, \rho(O_a), f)$. A contradiction.
   (b) $O_b \neq O_1$.
   This case can arise, if a statement S_y in another parallel thread has written to $O_a.f$. It implies, based on the induction hypothesis, $S_n \in MHP(S_y)$ and $\rho(O_1) \in fieldPts(S_y, \rho(O_a), f)$. Again, Rule ⟨2,1⟩ ensures that $\rho(O_1) \in fieldPts(S_x, \rho(O_a), f)$. A contradiction.

□