Heap Abstraction via Early-Confluent Object Merging for Pointer Analysis

JINPENG WANG*, YUFEI LIANG*, and ZHONGSHENG ZHAN, Nanjing University, China
TIAN TAN† and YUE LI†, Nanjing University, China

Heap abstraction critically affects both the efficiency and precision of pointer analysis for Java programs. By merging heap objects allocated at different program points, heap abstractions can significantly improve analysis efficiency, but often at the cost of precision. Mahjong, a state-of-the-art heap abstraction based on object merging, demonstrates that object merging can substantially improve the efficiency of pointer analysis while preserving precision for type-dependent clients; however, this client-specific guarantee limits its general applicability. In this work, we investigate how to improve the efficiency of pointer analysis through object merging, while preserving precision in a manner independent of any particular client. Our key insight is that, from the perspective of pointer analysis, many heap objects exhibit *early flow confluence*: they are allocated at different program points and then quickly propagate to the same pointers (variables or fields), after which they continue to flow together through the program. Merging such early-confluent objects has negligible impact on overall analysis precision. In contrast, merging objects that do not flow to the same pointers, or that converge only much later, can introduce substantial precision loss.

Guided by this insight, we propose Valve, a new heap abstraction approach that efficiently identifies and merges early-confluent objects. Valve encodes the flow information needed for early-confluence detection as nondeterministic finite automata (NFAs) and approximates mergeability checking via an NFA-equivalence test, enabling efficient object merging while retaining high precision. We evaluate Valve on the largest benchmarks used in recent literature as well as modern large-scale Java applications, by integrating it with multiple state-of-the-art pointer-analysis techniques and directly comparing it with Mahjong. The results show that Valve achieves substantially higher precision than Mahjong for non-type-dependent clients, while maintaining comparable precision for type-dependent clients. At the same time, Valve delivers comparable or often better analysis efficiency across all evaluated cases. Overall, Valve, as a heap abstraction approach, significantly improves the efficiency of pointer analysis across several state-of-the-art techniques while maintaining high precision (99.61% on average).

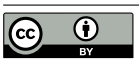
CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Pointer Analysis, Heap Abstraction, Java

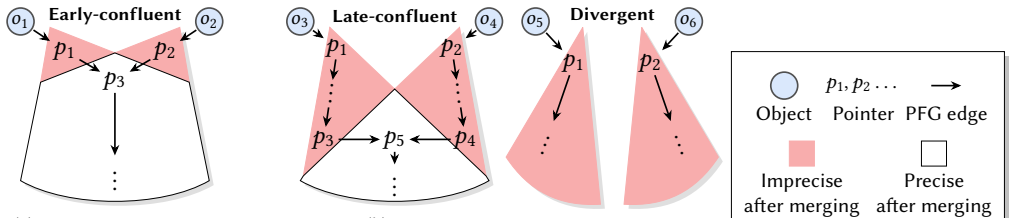
ACM Reference Format:
Jinpeng Wang, Yufei Liang, Zhongsheng Zhan, Tian Tan, and Yue Li. 2026. Heap Abstraction via Early-Confluent Object Merging for Pointer Analysis. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 353 (October 2026), 30 pages. https://doi.org/10.1145/3839485

*Both authors contributed equally to this work.
†Corresponding author.

Authors' Contact Information: Jinpeng Wang, jpwang@smail.nju.edu.cn; Yufei Liang, 602024330013@smail.nju.edu.cn; Zhongsheng Zhan, yahya_chan@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China, Nanjing; Tian Tan, tiantan@nju.edu.cn; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China, Nanjing.



ACM 2475-1421/2026/10-ART353
https://doi.org/10.1145/3839485



(a) Merging o_1, o_2 incurs nearly negligible imprecision

(b) Merging o_3, o_4 or o_5, o_6 incurs substantial imprecision

Fig. 1. Illustration of early flow confluence and how object merging affects pointer analysis precision. Red regions denote the portion of the PFG where merging loses precision (i.e., introduces additional over-approximation), whereas white regions preserve the original precision after merging.

1 Introduction

Pointer analysis is a fundamental static analysis technique that underpins numerous client applications, including security analysis [5, 14, 20, 26, 33, 66, 69], bug detection [10, 13, 34, 35, 40], compiler optimization [3, 27, 39, 61], and verification [2, 19, 29]. For modern object-oriented languages such as Java, balancing efficiency and precision of pointer analysis remains a long-standing challenge.

Two largely independent design dimensions shape this tradeoff: heap abstraction [24] and context sensitivity [44]. Heap abstraction governs how dynamically allocated objects are represented and grouped during analysis, while context sensitivity determines how calling contexts are distinguished. Over the past decade, research on context sensitivity has achieved substantial progress, leading to sophisticated techniques such as selective context sensitivity [31, 47, 65] and hybrid context sensitivity [25, 50, 53], which improve efficiency while maintaining precision. In contrast, heap abstraction has received less systematic attention, and allocation-site abstraction has remained the prevailing design choice in practice. This abstraction represents all objects allocated at the same allocation site by a single abstract object. This design continues to dominate widely used Java analysis frameworks such as Soot [56], WALA [12], Doop [9], and Tai-e [49].

Despite its relative stability as a design choice, heap abstraction plays a central role in shaping the effectiveness of pointer analysis. The number and granularity of abstract heap objects directly determine the size of the points-to relation and the cost of propagating points-to information. This influence becomes even more pronounced under context-sensitive configurations. When calling contexts are distinguished, each allocation site may give rise to multiple context-specific heap objects, potentially multiplying the size of the abstract heap and amplifying the cost of analysis. Thus, despite its importance, heap abstraction remains relatively underexplored, leaving a largely untapped design space for more effective Java pointer analysis.

To improve the efficiency-precision tradeoff, prior work [24, 51] has explored object merging, which selectively merges heap objects allocated at different program points to shrink the abstract heap. Most notably, the state-of-the-art Mahjong [51] shows that such merging can significantly accelerate pointer analysis while preserving precision for type-dependent clients—clients whose correctness depends only on the types of objects pointed to by variables, such as resolving virtual calls for call-graph construction or determining potentially failing cast operations. However, such approaches establish their precision guarantees only for specific classes of clients. For other clients, such as thread-escape analysis and side-effect analysis, they provide no precision guarantees and may incur precision loss. This client-centric perspective raises a broader question: can we design a heap abstraction that improves efficiency through object merging while preserving pointer analysis precision independently of any particular client?

Key Insight. In this work, we answer this question affirmatively by focusing on a client-independent notion of precision: the precision of the computed points-to relation, which serves as a common interface for many downstream clients that query points-to or derived aliasing information [62]. To reason about how points-to facts are produced, we view pointer analysis through a *pointer flow graph* (PFG) [12, 22, 31, 37, 55], whose nodes are pointers (variables or fields) and whose edges represent flows among pointers (akin to the constraint graph in Andersen-style analyses [4]). Intuitively, abstract objects are generated at allocation sites and then propagate through the PFG; whenever an object reaches a pointer node, the analysis derives the corresponding points-to fact.

Our key observation is that many heap objects exhibit *early flow confluence*: although allocated at distinct program points, they quickly reach an overlapping set of pointers (i.e., PFG nodes) and subsequently propagate through largely the same region of the PFG. We call such objects *early-confluent objects* and define them in Section 3. As illustrated in Fig. 1(a), once objects converge early, they propagate together and induce essentially the same points-to facts; thus, merging them introduces negligible imprecision, which is largely confined to their (typically small) pre-confluence regions. In contrast, as shown in Fig. 1(b), merging objects that converge only much later after traversing substantially different regions of the PFG, or that do not converge at all, can introduce many spurious points-to facts and increase false positives in downstream clients. Therefore, by identifying early-confluent objects, we can merge them aggressively to shrink the abstract heap—and thus accelerate pointer analysis—while largely preserving the precision.

Approach. Guided by the above insight on early flow confluence, we present Valve (Section 4), a new heap abstraction approach that accelerates pointer analysis by merging objects while retaining high precision. Identifying such merge candidates requires reasoning about how objects flow through the program. A natural approach is to analyze object flows on the pointer flow graph (PFG) introduced above. However, constructing a complete PFG typically requires a pre-analysis—running whole-program pointer analysis to a fixed point—and the resulting graph can be large, making both its construction and subsequent analysis expensive.

To avoid this overhead, Valve approximates early confluence using allocation-local flow information derived from lightweight program structure around allocation sites, without requiring the full PFG. This design matches our goal: observing convergence shortly after object allocation provides strong evidence that the objects become confluent early.

The key idea behind Valve is to identify early-confluent objects through nondeterministic finite automata (NFA) equivalence testing. We encode the flow information relevant to early confluence as NFAs and use an appropriate notion of NFA equivalence as an efficient approximation for identifying early-confluent objects. This approximation enables Valve to efficiently identify and merge early-confluent objects. The use of NFA equivalence to identify mergeable objects is inspired by Mahjong [51], while Valve's NFAs differ fundamentally in what they represent and how equivalence is defined (detailed in Section 4).

Evaluation. We evaluate Valve on the largest benchmarks used in recent studies [37, 61, 65] and additional modern large-scale Java applications (Section 5). We compare Valve with Mahjong [51] by integrating both into three state-of-the-art pointer-analysis techniques—Zipper^e [31], Moon [65], and Cut-Shortcut [37]—representing different design principles and efficiency-precision tradeoffs. Across all three analyses, Valve achieves substantially higher precision than Mahjong for non-type-dependent clients (avg. Valve: 99.37% vs. Mahjong: 75.48%), while maintaining comparable precision for type-dependent clients (Valve: 99.84%, Mahjong: 99.84%).

In terms of efficiency, Valve matches or often surpasses Mahjong across all evaluated configurations. For the especially large programs (9 of 13 benchmarks) where Zipper^e and Moon cannot

finish within 5 minutes, the Valve-enhanced analyses significantly outperform their Mahjong-enhanced counterparts: With the efficiency-oriented Zipper^e (avg. 1714 seconds), Valve achieves an average speedup of 41.3×, compared with Mahjong's 4.3×. With the precision-oriented Moon (avg. 3613 seconds), Valve attains an average speedup of 4.3×, versus Mahjong's 1.2×. For the largest programs where both Zipper^e and Moon are not scalable but the ultra-efficient Cut-Shortcut—even faster than context-insensitive analysis—still scales, Valve further reduces its analysis time by 2893 seconds on average (up to 7576 seconds). In contrast, Mahjong slows down the analyses because it relies on a whole-program pointer analysis as a pre-analysis to construct the required program information, whose cost becomes significant for programs of this scale.

In summary, this work makes the following contributions:

- We introduce the notion of *early-confluent objects* (Section 3) based on an observation about object flows: many objects allocated at different program points quickly reach overlapping pointers and then continue to flow together. This observation motivates a client-independent criterion for object merging, which tends to introduce negligible imprecision.
- We propose Valve (Section 4), a new heap abstraction that efficiently identifies and merges early-confluent objects. By guiding object merging using early flow confluence, Valve improves pointer analysis efficiency while maintaining high precision.
- We conduct an extensive evaluation of Valve on large benchmarks (Section 5). Compared to Mahjong, the state-of-the-art merging-based heap abstraction, Valve achieves substantially higher precision than Mahjong for non-type-dependent clients, while maintaining comparable precision for type-dependent clients. At the same time, Valve delivers comparable or often better analysis efficiency across all evaluated cases.
- We built Valve on top of Tai-e [49], a state-of-the-art static analysis framework for Java, and have fully open-sourced Valve as part of a publicly accessible artifact [58].

2 Motivating Example

As discussed in Section 1, existing heap object merging approaches can substantially accelerate pointer analysis, but they preserve precision only for specific clients. In this section, we motivate the need for a more general object merging approach using a simplified example distilled from a real-world application. We first use this example to explain the merging criterion employed by Mahjong, the state-of-the-art object merging technique, and show how this criterion degrades the overall precision of pointer analysis. We then revisit the same example using our approach, demonstrating that adopting a different perspective on object merging allows Valve to identify alternative merging opportunities while preserving overall precision. Throughout the paper, we consider flow-insensitive pointer analysis, following the standard of Andersen-style pointer analysis for Java.

Example. Figure 2 presents a simplified HTTP application with three classes: `Request`, `Usage`, and `Client`. Class `Request` (line 1) models HTTP requests with field `f` storing the request payload. Class `Usage` (line 5) demonstrates two usage scenarios: `usage1()` (line 6) builds a request using `Client` and then sends it, while `usage2()` (line 11) directly constructs a request without using the client. Class `Client` (line 17) provides two methods for request handling: `build()`, and `send()`. Method `build()` (line 18) allocates two `Request` objects, assigning either a `File` or `String` to the payload field depending on some condition. The `send()` method (line 29) first retrieves the request's payload to perform validation (line 30), and then proceeds to transmit the request (omitted for brevity).

The right side of Figure 2 shows the corresponding pointer flow graph (PFG) [31], in which each node represents a variable or an object field, and an edge from pointer `x` to pointer `y` indicates that any object pointed to by `x` may flow to `y`. The left half of the PFG tracks three distinct `Request` objects: o_{20}, o_{24}, and o_{12} (o_i denotes the abstract object allocated at line i). Objects o_{20} and o_{24}, created

```
1 class Request {
2   Object f;
3 }
4
5 class Usage {
6  void usage1(Client c){
7   r = c.build();
8   c.send(r);
9  }
10
11 void usage2() {
12   z = new Request(); // o12
13   z.f = new String();// o13
14   // process z...
15 }
16 }
```

```
17 class Client {
18  Request build() {
19   if (...) {
20    x = new Request();// o20
21    x.f = new File(); // o21
22    return x;
23   } else {
24    y = new Request(); // o24
25    y.f = new String();// o25
26    return y;
27   }
28 }
29 void send(Request s) {
30   p = s.f;
31   // validate p, send s...
32 }}
```

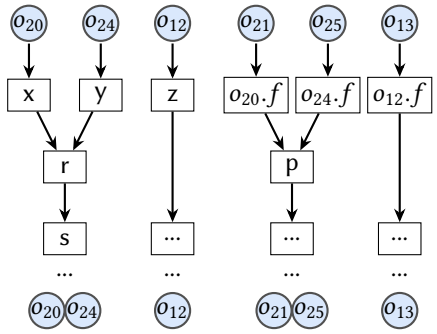
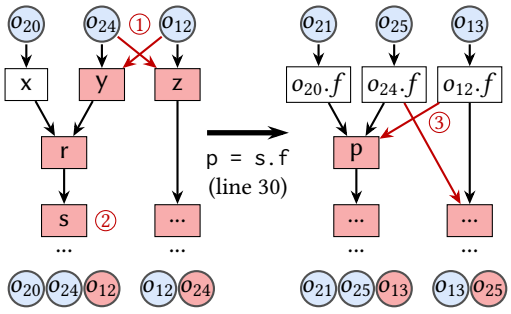
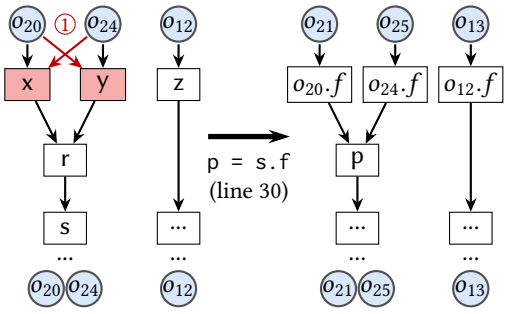


Fig. 2. A simplified HTTP application and its corresponding PFG. Nodes represent variables/object fields, and an edge from x to y indicates objects pointed to by x may flow to y. Here, r, x, y, z, and p are local variables.



(a) The impact of Mahjong merging o_{24} and o_{12}.



(b) The impact of Valve merging o_{20} and o_{24}.

Fig. 3. Comparison of Mahjong to Valve on how different merging affects precision in terms of points-to sets. Red elements indicate imprecision introduced by the merge: red pointers become imprecise, red arrows show how imprecision propagates, and red objects are spuriously pointed to by the imprecise pointers.

in `build()`, flow together as return values to `r` (line 7). Subsequently, they are passed together as arguments to method `send()` and continue to propagate jointly thereafter. In contrast, o_{12}, created in `usage2()`, propagates independently and never converges with o_{20} or o_{24}. The right half of the PFG tracks three payload objects. The payload objects in the fields of o_{20} and o_{24} are loaded into `p` in `send()`, whereas the payload of o_{12} is accessed independently in `usage2()`.

Mahjong. Mahjong aims to accelerate pointer analysis while preserving the type information in points-to sets. In particular, after object merging, any pointer that originally may point to objects of certain types should still point to objects of exactly those types. This enables Mahjong to preserve the precision of type-dependent clients, such as call graph construction.

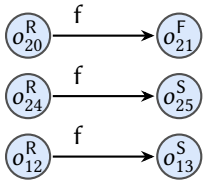


Fig. 4. Mahjong's FPG.

To achieve this goal, Mahjong merges objects that are *type-consistent*, a merging criterion defined on the field points-to graph (FPG) [51]. Figure 4 illustrates the FPG for our example, where nodes represent objects (superscripts denote initials of their types) and labeled edges represent instance field points-to relations. Two objects are type-consistent if traversing any identical sequence of fields from their respective nodes reaches objects of the same type. In our example, o_{24} and o_{12} are type-consistent: following

field f from either object reaches a `String` (o^{S}_{25} and o^{S}_{13}). Consequently, merging them minimally impacts object *types* in points-to sets: after merging, their field f still points to `String` objects.

However, this criterion can noticeably degrade precision in terms of points-to relations. Figure 3a illustrates the impact of merging o_{24} and o_{12} on the PFG under Mahjong, where ①-③ highlight the propagation of imprecision, as detailed below: ① Merging o_{24} and o_{12} can be viewed as adding both objects to their allocation-site left-hand-side variables (A-LHS variables). Consequently, A-LHS variables `y` and `z`, which previously pointed to either o_{24} or o_{12}, now imprecisely point to both; ② These imprecise A-LHS variables then propagate imprecision along PFG edges: in particular, any pointer that is reachable from exactly one of the A-LHS variables (but not the other) becomes imprecise. For instance, variable `s`, a successor of `y`, now imprecisely points to o_{12}; ③ Moreover, load, store, and call operations on all imprecise pointers may introduce additional PFG edges, causing imprecision to further propagate. Consider the load statement `p = s.f` (line 30): since `s` imprecisely points to o_{12}, an additional PFG edge from $o_{12}.f$ to `p` will be introduced (red edge in Figure 3a). As a result, `p` now imprecisely points to o_{13} and continues to propagate imprecision along the PFG. This cascading propagation illustrates that even a single merge under the type-consistent criterion can lead to compounding imprecision in points-to relations.

Our Approach: Merge Early-Confluent Objects. The example above exposes a key limitation of the type-consistent criterion: by focusing solely on type preservation, it overlooks how merging affects object propagation in pointer analysis itself, leading to cascading precision loss. To address this issue, we reconsider heap abstraction from the perspective of object flow. Our key insight is that many heap objects exhibit *early flow confluence*: although allocated at different program points, they quickly propagate to the same pointers and thereafter flow together throughout the program. Merging such early-confluent objects has negligible impact on overall points-to precision. In contrast, merging objects that never converge in the PFG, or that converge only much later, can lead to substantial precision loss. In our example, o_{20} and o_{24} are early-confluent: they are created in the same method and immediately converge at the same return pointer, after which they propagate together. Merging them introduces imprecision only at pointers x and y, while leaving the points-to relations of all other pointers unchanged.

Guided by this insight, we propose Valve, a precision-guided heap abstraction that identifies and merges early-confluent objects by analyzing object flow in the program. In our example, Valve merges o_{20} and o_{24}, which preserves overall analysis precision while capturing additional acceleration opportunities that Mahjong systematically misses. At the same time, Valve deliberately avoids merging o_{24} and o_{12}, which never converge in the PFG, thereby preventing the imprecision introduced by Mahjong. Under this merging criterion, whereas Mahjong causes imprecision for most pointers in Figure 3a, Valve introduces imprecision only at two of them.

In Section 3 we introduce the concrete notion of early-confluent objects. In Section 4 we present Valve, our approach for efficiently identifying and merging such objects.

3 Early-Confluent Objects

In this section, we begin by presenting our observation of how merging impacts the precision of pointer analysis, and based on this observation, we define *early-confluent objects*—objects that can be merged while largely preserving precision.

Our observation in Section 3.1 reveals two distinct forms of imprecision when two objects are merged: (1) *Direct imprecision* arises at pointers that the merged objects directly flow to; (2) *Indirect imprecision* arises at pointers that the merged objects do not reach directly, but are relevant to the

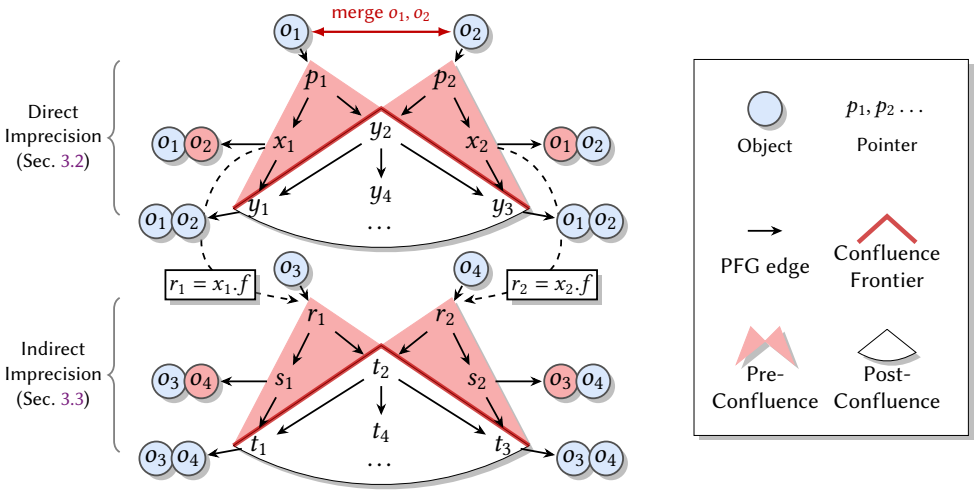


Fig. 5. The insight of Valve on which pointers may become imprecise after merging.

propagation of the merged objects. To preserve precision, both forms of imprecision must be considered. Accordingly, the definition of early-confluent objects is closely tied to these two forms of imprecision. We discuss how merging early-confluent objects confines each form in Sections 3.2 and 3.3.

3.1 Merging-Induced Imprecision and Early-Confluent Objects

This section presents our observation of how merging impacts the precision of pointer analysis, which motivates our merging criterion. By considering the confluence behavior of how objects flow through the program, imprecision due to merging them can be characterized as regions due to confluence. To better present our observation, we first introduce some concepts.

Definition 3.1 (Confluent Pointers). Two pointers p_1 and p_2 in a pointer flow graph are *confluent* if paths on the pointer flow graph from them eventually reach the same set of pointers. Specifically: (1) every path from p_1 eventually reaches a pointer reachable from p_2, and (2) symmetrically, every path from p_2 eventually reaches a pointer that is reachable from p_1.

Definition 3.2 (Confluence Frontier). For confluent pointers p_1 and p_2, their *confluence frontier* is the set of all pointers that: (1) are reachable from both p_1 and p_2, and (2) have some immediate predecessor that is reachable from either p_1 or p_2, *but not both*.

Intuitively, the confluence frontier of pointers p_1 and p_2 is the boundary where two previously separated flows from the two pointers begin to overlap. We illustrate this concept using Figure 5, focusing on its upper-left portion. In the figure, pointers p_1 and p_2 are confluent: every path from p_1 and every path from p_2 eventually reaches one of the pointers in $\{y_1, y_2, y_3\}$, which constitutes the confluence frontier of p_1 and p_2.

Definition 3.3 (Pre- and Post-Confluence Region). Given two confluent pointers p_1 and p_2, their confluence frontier divides the pointers reachable from p_1 or p_2 into two distinct regions: (1) the *pre-confluence region*, where pointers are reachable from either p_1 or p_2, but *not both*; and (2) the *post-confluence region*, where pointers are reachable from both p_1 and p_2.

Based on pre- and post-confluence regions, we now describe the two forms of merging-induced imprecision, using Figure 5. Building on this characterization, we then present a merging criterion that accounts for both forms of imprecision to achieve good precision during merging.

Direct Imprecision. We use direct imprecision to refer to imprecision arising from the pointers that the merged objects directly flow to—that is, the pointers residing in the pre-confluence region of the A-LHS variables of the merged objects. When two objects are merged, their A-LHS variables begin to imprecisely point to both objects. This imprecision then propagates along pointer-flow edges to subsequent pointers. As a result, all pointers in the pre-confluence region of the two A-LHS variables become imprecise. In contrast, pointers in the post-confluence region remain precise: since they are already reachable from both A-LHS variables without merging, they already point to both objects and are therefore unaffected by the merge.

Consider the two objects o_1 and o_2 with A-LHS variables p_1 and p_2 in Figure 5. When o_1 and o_2 are merged, both p_1 and p_2 will imprecisely point to both objects. o_1 and o_2 will then propagate along pointer-flow edges, causing pointers in the pre-confluence region of p_1 and p_2 (i.e., p_1, p_2, x_1, x_2) to be imprecise. In contrast, pointers y_1, y_2, y_3, and y_4 remain precise: they were already reachable from both p_1 and p_2, and point to both o_1 and o_2 without merging.

Indirect Imprecision. Direct imprecision is intuitive. However, it does not fully characterize merging-induced imprecision. Merging may also create indirect imprecision at pointers that are *not* directly reached by the merged objects from their A-LHS variables. This happens when imprecise variables are used as the bases of operations such as field loads or method calls.

Consider the two field load statements ($r_1 = x_1.f$ and $r_2 = x_2.f$) in the lower part of Figure 5, where x_1 and x_2 lie in the pre-confluence region of p_1 and p_2, but r_1 and r_2 do not. After merging, direct imprecision arises from x_1 and x_2, causing them to point to both o_1 and o_2. Due to the two field load statements: (1) r_1 may now imprecisely point to o_4 from $o_2.f$ (in addition to o_3 from $o_1.f$), and (2) r_2 may now imprecisely point to o_3 from $o_1.f$ (in addition to o_4 from $o_2.f$). This imprecision, originating from r_1 and r_2, then propagates through the PFG until reaching their confluence frontier, causing imprecision in the pre-confluence region of r_1 and r_2. Note that this process can cascade: subsequent operations (e.g., loading another field g) from variables in the pre-confluence region of r_1 and r_2 may introduce and propagate imprecision further.

Early-Confluent Objects as Merging Criterion. Guided by our observation, both direct and indirect imprecision can be characterized by pre-confluence regions of variable pairs related to the merged objects. To preserve precision, we thus need to confine the pre-confluence regions that give rise to both forms of imprecision. This gives the following definition of early-confluent objects:

Definition 3.4 (Early-Confluent Objects). Two objects are early-confluent if:

(1) their A-LHS variables are early-confluent; and
(2) the variables where indirect imprecision may arise are early-confluent as well.

Central to Definition 3.4 is the notion of *early-confluent variables*, which allows us to confine the pre-confluence region of two variables. When this notion is applied to variables identified in Definition 3.4, it confines the imprecision introduced by merging early-confluent objects. Back to Figure 5, whether o_1 and o_2 would be merged by Valve depends on whether p_1, p_2 and r_1, r_2 are early-confluent. If the A-LHS variables p_1 and p_2 are early-confluent, the direct imprecision introduced by merging is confined to a small, local region (Section 3.2). If r_1 and r_2 are early-confluent as well, the indirect imprecision introduced by merging is likewise confined to a small region (Section 3.3). Consequently, the overall precision loss incurred by merging o_1 and o_2 is small, and they are therefore considered as a valid merging candidate by Valve.

We introduce early-confluent variables and show how this notion confines direct imprecision in Section 3.2. We discuss the variables from which indirect imprecision may arise under different program constructs, and confine indirect imprecision using early confluence on these variables in Section 3.3. Our approach of identifying and merging early-confluent objects is detailed in Section 4.

3.2 Confining Direct Imprecision

Since direct imprecision arises from the pre-confluence region of A-LHS variables, a smaller pre-confluence region leads to less direct imprecision induced by the merge. Merging should therefore be permitted only when this region is sufficiently small; otherwise, the merge could introduce imprecision in a large portion of the program, which may then be further amplified by cascading imprecision effects. Moreover, we aim to make merging decisions efficiently; otherwise, the computational overhead may outweigh the benefits of merging.

For both precision and efficiency, we confine the pre-confluence region of two variables to their intra-procedural scopes, so as to ensure imprecision remains within a small, localized region. To achieve this, we introduce *first-escaped pointers* that enclose the local scope. If two variables converge before or exactly at their first-escaped pointers, then their pre-confluence region will not extend beyond the local scope, and we call such variables *early-confluent variables*. We give the definition of first-escaped pointers and early confluence as follows.

Definition 3.5 (First-Escaped Pointers). The *first-escaped pointers* of a variable v are the set of all pointers q such that there exists a pointer-flow path from v to q where the edge entering q is the first inter-procedural pointer-flow edge on that path. Such edges fall into three representative categories:

- Store: q is an instance field (e.g., $o.f$) that v flows to.
- Return: q receives the return value from a return variable that v flows to.
- Call (argument or receiver): q is a formal parameter (resp. `this` variable) of a method, and v flows to the corresponding argument (resp. receiver) at the call site.

In fact, apart from the cases above, we consider all other pointer-related Java features—array indices and static fields—that may give rise to first-escaped pointers. However, since they can be handled similarly to the three cases above, we omit them here and from the rest of this paper for brevity.

Intuitively, the first-escaped pointers of a variable mark the earliest boundary where its pointer flow escapes the local (intra-procedural) scope. Imprecision that crosses this boundary may propagate widely throughout the program. Based on this, we define *early-confluent variables*.

Definition 3.6 (Early-Confluent Variables). Two variables v_1 and v_2 are *early-confluent* if they have the same first-escaped pointers.

Consider objects o_{20} and o_{24} whose A-LHS variables are x and y in the method `build()` in Figure 2. According to Definition 3.5, since x and y are both directly returned from `build()` to its caller at line 7 in `usage1()`, r is the first-escaped pointer of them both. Having the same first-escaped pointers, x and y are early-confluent according to Definition 3.6. Merging o_{20} and o_{24}, according to Section 2, thus incurs imprecision only to x and y that are local to their allocation sites.

Requiring two variables to have the same first-escaped pointers has two advantages. First, identical first-escaped pointers of two variables tend to confine their pre-confluence region to a small local scope, thereby bounding the imprecision introduced by merging. Second, this definition enables efficient merging decisions. Rather than reasoning about pointer flows across the entire program—which would require constructing the entire PFG—we only need to compare the first-escaped pointers of variables. These can be analyzed using lightweight analysis (detailed in Section 4).

Note that our definition is not restricted to variables that converge within the same method. It is general enough to capture cases where variables from different methods that nonetheless share the same first-escaped pointers. For example, the A-LHS variables of two objects allocated in different methods may both be stored to the same instance field, and are early-confluent according to Definition 3.6. As our evaluation demonstrates, using this definition preserves much of the precision of a precise pointer analysis while achieving substantial speedup (detailed in Section 5).

Confining Direct Imprecision. To preserve precision for object merging, we therefore account for direct imprecision at pointers directly reached by the merged objects through their A-LHS variables. For each candidate merge (o_1, o_2) with A-LHS variables (p_1, p_2), we require p_1, p_2 to be early confluent so as to restrict their pre-confluence region, thereby confining direct imprecision.

3.3 Confining Indirect Imprecision

Direct imprecision explains the effect of merging on pointers reachable from A-LHS variables. However, merging can also introduce indirect imprecision at pointers that are not directly reachable from the A-LHS variables, but become imprecise due to operations on imprecise variables (e.g., loading the field of some variable in the pre-confluence region of A-LHS variables).

Indirect imprecision may arise due to three kinds of pointer-related operations (1) instance field loads, (2) instance field stores, and (3) method calls. In the following, we focus on indirect imprecision caused by field loads, method calls, and their combinations, and show how variable early confluence can confine such imprecision. We do not explicitly consider field stores, because for indirect imprecision introduced by field store operations to propagate through the program, it needs to be loaded into a variable. Therefore, by considering imprecision at load sites, we prevent its propagation through the program. This may leave imprecision in instance fields themselves, but such imprecision does not further propagate to other pointers.

Loaded Instance Fields. Indirect imprecision may arise from the left-hand-side variables of field-load statements whose base variables are imprecise. Consider the example in Figure 6. Class `Box` has a field f and a method `get()` to obtain f. Two `Box` objects o_1 and o_2 are allocated (lines 9–10), their field f is assigned o_3 and o_4, respectively (lines 11–12), and the fields are loaded to r_1 and r_2 (lines 13–14). Finally, r_1 and r_2 are assigned to t (lines 15–16).

```
1 class A { }
2
3 class Box {
4  A f;
5  A get() {
6   return this.f;
7  }
8 }
```

```
9  Box x1 = new Box();//o1
10 Box x2 = new Box();//o2
11 x1.f = new A(); //o3
12 x2.f = new A(); //o4
13 A r1 = x1.f;
14 A r2 = x2.f;
15 t = r1;
16 t = r2;
```

Fig. 6. Example of indirect imprecision via instance field loads.

As illustrated by the dashed lines in Figure 5, merging o_1 and o_2 introduces indirect imprecision in r_1 and r_2 via the field-load statements at line 13–14. Specifically, x_1 and x_2—the respective A-LHS variables of o_1 and o_2—will imprecisely point to both objects, causing direct imprecision in their pre-confluence region. Then, because x_1 (resp. x_2) now points to both o_1 and o_2, loading field f through x_1 (resp. x_2) causes r_1 (resp. r_2) to receive objects from both $o_1.f$ and $o_2.f$. Consequently, as shown at the bottom of Figure 5, both r_1 and r_2 end up imprecisely pointing to both o_3 and o_4.

This indirect imprecision then propagates to all pointers reachable from r_1 or r_2, polluting their points-to sets with spurious objects in the pre-confluence region. Pointers in the post-confluence region of r_1 and r_2 (e.g., t), however, remain unchanged. Since these pointers are reachable from both r_1 and r_2 without merging, they already receive o_3 from $o_1.f$ via r_1 and o_4 from $o_2.f$ via r_2. Therefore, merging o_1 and o_2 does not alter their points-to sets. As a result, although merging o_1 and o_2 renders r_1 and r_2 imprecise, the variable t in their post-confluence region remains unchanged.

Method Calls. Indirect imprecision may also arise from variables that receive return values from methods dispatched on imprecise variables. Consider the example in Figure 7. Two `Box` objects o_1 and o_2 are allocated (lines 1–2), with their fields f set to o_3 and o_4, respectively (lines 3–4). The program then calls method `get()` on x_1 and x_2, storing the results in r_1 and r_2 (lines 5–6). When o_1 and o_2 are merged, their A-LHS variables x_1 and x_2 will imprecisely point to both objects, introducing direct imprecision. Because x_1 (resp. x_2) now points to both o_1 and o_2,

```
1 Box x1 = new Box(); //o1
2 Box x2 = new Box(); //o2
3 x1.f = new A(); //o3
4 x2.f = new A(); //o4
5 A r1 = x1.get();
6 A r2 = x2.get();
```

Fig. 7. Example of indirect imprecision via method call.

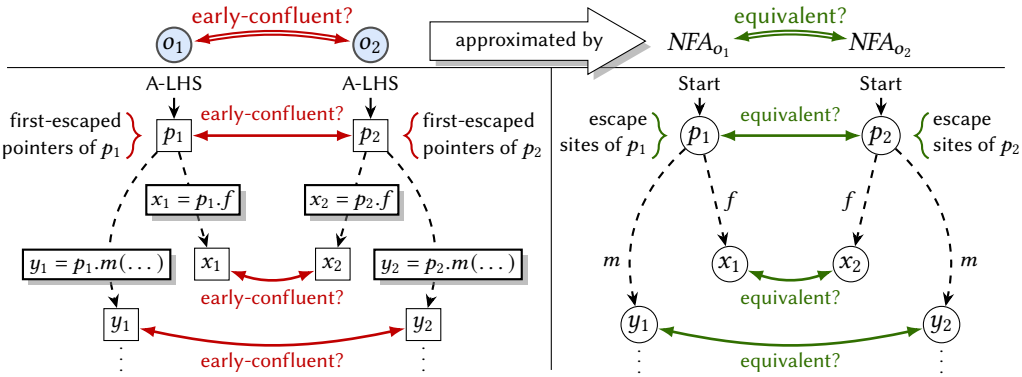


Fig. 8. From early-confluence determination to NFA equivalence testing.

each call `x1.get()` (resp. `x2.get()`) dispatches on both receiver objects (and hence multiple callees), so the return variable may receive the union of return values across those callees, retrieving the same instance field f from both o_1 and o_2. Therefore, r_1 and r_2 receive return values from both method calls and become imprecise by pointing to both o_3 and o_4, introducing indirect imprecision.

Combinations of Field Loads and Method Calls. The above effects can cascade, causing further indirect imprecision through combinations of field accesses and method calls. Consider statements $r_1 = x_1.m_1()$ and $r_2 = r_1.m_2()$ using a temporary variable r_1. If merging makes x_1 imprecise, then r_1 may receive imprecise results through the first call (as in the method-call case). Once r_1 is imprecise, the method call on r_1 can further introduce indirect imprecision at r_2. This can extend to longer chains of method calls and field accesses, making variables holding the intermediate results imprecise.

Confining Indirect Imprecision. To preserve precision for object merging, we therefore account for not only direct imprecision at A-LHS variables, but also the indirect imprecision introduced by field loads, method returns, and their combinations.

Concretely, for each candidate merge (o_1, o_2) with A-LHS variables (x_1, x_2), we collect all variables defined by loads/calls whose base is x_1 or x_2 (and, transitively, base variables that become imprecise through such definitions). For each matched variable pair (r_1, r_2) arising from the same (combinations of) field/method operation(s), we require them to be early-confluent. This keeps each affected variable's pre-confluence region small and local, thereby confining indirect imprecision.

4 Valve: Identifying and Merging Early-Confluent Objects

This section presents Valve, our novel approach to identify and merge early-confluent objects based on the insight developed in Section 3. Starting from an allocation-site-based abstraction, Valve identifies and merges groups of objects that are mutually early-confluent to coarsen the heap. For ease of exposition, we illustrate our approach using one pair of objects in the following section. The merged heap is subsequently fed to a pointer analysis, thereby improving its efficiency.

To enable efficient identification and merging, Valve encodes the flow information relevant to early-confluence determination as nondeterministic finite automata (NFA). Under this encoding, whether two objects are early-confluent is approximated using an appropriate notion of NFA equivalence. We present the NFA-based approximation of early-confluence determination and an overview of Valve in Section 4.1. We then describe the construction of the NFAs and their equivalence checking in Sections 4.2 and 4.3.

4.1 From Early Confluence to NFA Equivalence

To ease the understanding, we first detail the approximation from early confluence to NFA equivalence, and then give an overview of Valve at the end of this section.

As discussed in Section 3, determining whether two objects are early-confluent involves checking early-confluence across pairs of variables at which imprecision may arise due to merging (e.g., the objects' A-LHS variables, as well as variables derived from them via field loads, method calls, and subsequent operations), so as to confine merging-induced imprecision. Specifically, if two variables have identical first-escaped pointers, they are early-confluent, and imprecision is confined to a small localized region; if all such variable pairs are early-confluent, then the two objects are early-confluent and constitute a good merging candidate.

Consider the left side of Figure 8: to determine whether objects o_1 and o_2 are early-confluent, we need to examine early-confluence by comparing the respective first-escaped pointers for (1) their A-LHS variables p_1 and p_2; (2) variables x_1 and x_2 introduced by the field-load statements $x_1 = p_1.f$ and $x_2 = p_2.f$; (3) variables y_1 and y_2 introduced by the method-call statements $y_1 = p_1.m(\dots)$ and $y_2 = p_2.m(\dots)$; and (4) any further variables derived via subsequent operations on them.

NFA Equivalence. We observe that NFAs are well-suited for encoding the information required for early-confluence analysis, as illustrated on the right side of Figure 8. We define the *object-flow NFA* for an object o, denoted as NFA_o, as a 6-tuple sequential automaton $(Q, \Sigma, \delta, q_0, \Gamma, \gamma)$ [1], where each component captures a specific aspect of the object's local flow from its allocation site.

(1) States (Q): Each variable appearing on the left corresponds to a state in Q. The set of states comprises the A-LHS variable of o and all variables reachable from it via a sequence of operations, such as assignments, instance field loads, and method calls.
(2) Transitions ($\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$): Each operation on a variable that gives rise to another variable on the left corresponds to a labeled transition between the two corresponding states. Transition labels are: ϵ for copy assignments, f for instance field load of field f, and m for a call to method m. For example, the field-load statement $x_1 = p_1.f$ corresponds to a transition from state p_1 to state x_1 with label f, i.e., $x_1 \in \delta(p_1, f)$. Because each state may have multiple successor states under the same label (e.g., some variable's field f may be loaded multiple times), the resulting automaton is an NFA.
(3) Initial State (q_0): The A-LHS variable of o serves as the initial state on the right, since it is the origin of all subsequent flows to be tracked for o.
(4) State Labeling ($\gamma : Q \rightarrow \mathcal{P}(\Gamma)$): To facilitate early-confluence determination, each variable on the left has the set of its first-escaped pointers, and we annotate its corresponding NFA state $q \in Q$ on the right using escape sites as $\gamma(q)$, which encodes the corresponding first-escaped pointers. We give the precise definition of escape sites in Section 4.2.
(5) State Labels (Γ): Γ is the set of all possible escape sites.
(6) Transition Labels (Σ): Σ is the set of all possible transition labels.

Under this encoding, determining whether two objects are early confluent can be approximated via checking the equivalence of their corresponding object-flow NFAs. Intuitively, two objects' NFAs are equivalent if, for every sequence of operations from their start states, the reached states carry equivalent state labels [51]. This indicates that, for all variable pairs requiring examination, they are early confluent, and the two objects are thus early-confluent (detailed in Section 4.3).

The high-level NFA-equivalence-checking idea is inspired by Mahjong [51], which uses NFA equivalence to identify type-consistent objects for merging. However, Valve introduces a fundamentally different NFA formulation tailored to early-confluence detection, in which the states, transitions, and state labels are defined to capture object-flow structure rather than type consistency.

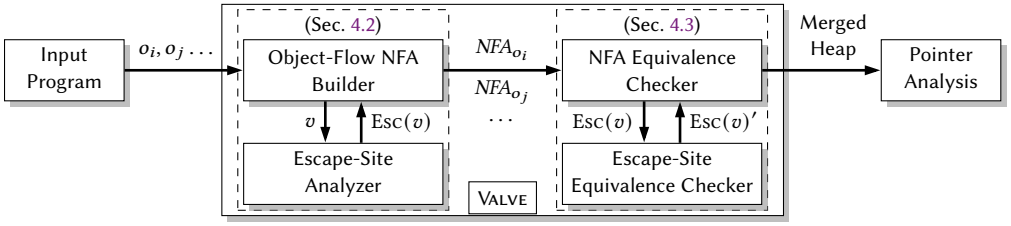


Fig. 9. Overview of Valve.

Overview of Valve. Figure 9 presents an overview of Valve that illustrates its key components and their interactions. Valve begins with an allocation-site-based heap abstraction of the input program, and considers each pair of objects o_i and o_j. For each object, the NFA Builder constructs an object-flow NFA that models its flow information after allocation. During construction, the *Escape-Site Analyzer* identifies the escape sites—which encode first-escaped pointers—for each variable, and attaches them as labels to the state corresponding to that variable.

The resulting NFAs are then passed to the Equivalence Checker. The checker determines whether the two objects are early-confluent by checking NFA equivalence based on the equivalence relation we defined over the state labels, which is implemented by the *Escape-Site Equivalence Checker*. When the NFA_{o_i} and NFA_{o_j} are deemed equivalent, we merge o_i and o_j in the resulting heap abstraction, which is supplied to subsequent pointer analysis to improve its efficiency.

We detail each of the components in the following subsections (Sections 4.2 and 4.3).

4.2 Constructing Object-Flow NFAs

This section presents the first step of Valve: constructing object-flow NFAs that will subsequently be used for equivalence checking to guide merging decisions. This is achieved by first running the NFA Builder to construct an NFA for each object, and then running the Escape-Site Analyzer to attach escape sites as state labels to the NFA.

4.2.1 NFA Construction Overview. The NFA for object o is constructed by performing a lightweight, intra-procedural traversal of the statements in the method containing o's allocation site.

BuildNFA() in Algorithm 1 details the construction for object-flow NFAs. The traversal starts from o's A-LHS variable (line 4). For each variable x, we add it to the set of states Q (line 7), and then identify all assignments $y = x$, field accesses $y = x.f$, and method calls $y = x.m(...)$, where x is used as a base variable or copied to the left-hand side variable (lines 8–16). For each such operation, we add a new state representing the left-hand side variable y and create a transition from x to y labeled with the operation. We recursively repeat this process for each newly discovered variable state until no new variables or operations can be reached from the allocation site within the current method. Finally, for each state x that encodes a variable, we identify its escape sites $\text{Esc}(x)$ via the Escape-Site Analyzer, and record them as x's state label $\gamma(x)$ (lines 17–19).

4.2.2 Defining and Identifying Escape Sites. As discussed above, each state in the object-flow NFA should be annotated with a state label that encodes the first-escaped pointers of the corresponding variable, thereby supporting the determination of early confluence for the corresponding variable pairs. However, identifying the exact first-escaped pointers to serve as state labels requires a PFG—whose construction typically depends on a whole-program pointer analysis that is costly—as it involves reasoning about how objects flow inter-procedurally through the program.

To avoid this overhead, Valve uses *escape sites* to approximate first-escaped pointers. Intuitively, an escape site is a reference at a statement (e.g., a field reference $y.f$ at a field store statement

Algorithm 1: Object-flow NFA construction.

Input: Object o
Output: o's object-flow NFA $(Q, \Sigma, \delta, q_0, \Gamma, \gamma)$
Function BuildNFA(o):
  $Q, \Sigma, \delta, \Gamma, \gamma \leftarrow \{\}, \{\}, \{\}, \{\}, \{\}$
  WorkList $\leftarrow$ {o's A-LHS variable}
  while *WorkList is not empty* **do**
    retrieve x from *WorkList*
    add x to Q
    foreach $\boxed{y = x}$ **do**
      add y to $\delta(x, \epsilon)$
      add y to *WorkList*
    foreach $\boxed{y = x.f}$ **do**
      add y to $\delta(x, f)$, add f to Σ
      add y to *WorkList*
    foreach $\boxed{y = x.m(\dots)}$ **do**
      add y to $\delta(x, m)$, add m to Σ
      add y to *WorkList*
  foreach $x \in Q$ **do**
    $\gamma(x) \leftarrow \text{Esc}(x)$
    $\Gamma \leftarrow \Gamma \cup \gamma(x)$
  return $(Q, \Sigma \cup \{\epsilon\}, \delta, \text{A-LHS}(o), \Gamma, \gamma)$

Input: Variable v
Output: Escape sites of v
Function Esc(v):
  WorkList $\leftarrow \{v\}$
  $E \leftarrow \{\}$
  while *WorkList is not empty* **do**
    retrieve x from *WorkList*
    foreach $\boxed{y.f = x}$ **do**
      add $\langle y.f \rangle$ to E
    foreach $\boxed{\text{return } x}$ **do**
      let m be the enclosing method
      add $\langle m \rangle$ to E
    foreach $\boxed{r = y.m(a_1 \dots x \dots a_n)}$ **do**
      add $\langle y.m(a_1 \dots \star \dots a_n) \rangle$ to E
    foreach $\boxed{y = x}$ *and y is not visited* **do**
      add y to *WorkList*
  return E

$y.f = x$) through which objects escape the current method, flow to some first-escaped pointers in other methods or the heap. The key insight is that, although computing the exact first-escaped pointers requires inter-procedural reasoning about pointer flows, the escape sites through which objects escape to them can be identified in a lightweight manner. This observation enables us to analyze whether two variables share the same first-escaped pointers by applying a dedicated equivalence relation over their escape sites, thereby efficiently approximating early confluence.

Function Esc(v) in Algorithm 1 computes the escape sites for variable v. It maintains a worklist initialized with v and repeatedly processes a variable x reachable from v by assignments (e.g., $x = v$). For each such x, the algorithm records every escape site at which x may escape:

- Store. For each store statement $y.f = x$, it adds the escape site $\langle y.f \rangle$, indicating that the objects pointed to by x may escape into the heap location denoted by field f of objects pointed to by y.
- Return. For each return statement return x, it adds the escape site $\langle m \rangle$, indicating objects are returned to the caller of the enclosing method m.
- Call (argument and receiver). For a call $r = y.m(a_1, \dots, x, \dots, a_n)$ where x is passed as an argument, the escape site $\langle y.m(a_1, \dots, \star, \dots, a_n) \rangle$ is added, with $\star$ marking the argument position. The case for receivers is similar, with a mark $\star$ at the receiver position.

4.3 Equivalence Checking of Object-Flow NFAs

Once all object-flow NFAs are constructed, we identify early-confluent objects by checking the equivalence of these NFAs, which is based on the equivalence relation on escape sites. We first describe how Valve determines whether two object-flow NFAs are equivalent under a straightforward identity relation on escape sites. We then present two relaxation techniques that relax this equivalence relation, enabling the identification of early-confluent objects that are missed under the strict identity relation, thereby creating more opportunities for merging.

4.3.1 The NFA Equivalence Checking. Following [51], two NFAs are considered equivalent if, for every sequence of transitions from their respective initial states, the states reached by that sequence carry the identical set of escape sites as state labels. Because the automata are nondeterministic, a single sequence of transitions may lead to multiple states; in this case, we require that the *union* of the labels of all reachable states be the same. We adopt a standard two-step approach for NFA equivalence checking, which tests the equivalence of two sequential automata in almost linear time [51].

To collect object pairs for checking, Valve first scans all statements in the input program to identify every allocation site (and its corresponding allocated object). Since all objects are obtained directly by scanning the whole program, this process does not require constructing or traversing a call graph. For each identified object, Valve constructs a corresponding object-flow NFA. It then performs equivalence checking on all pairs of these NFAs, including those whose corresponding objects are allocated in different methods or classes. To reduce the number of meaningless object pairs to check, Valve uses pruning to avoid unnecessary comparisons. Specifically, it ignores object pairs of different types, and it also ignores pairs whose initial NFA states have non-equivalent state labels (i.e., escape sites), since such NFAs cannot satisfy the definition of NFA equivalence in our setting.

4.3.2 The Equivalence Relation on Escape Sites. Using identity relation on escape sites for NFA equivalence checking is too strict for Valve to identify early-confluent objects, as distinct escape sites may also resolve to the same first-escaped pointers and result in early confluence. For example, consider escape sites $\langle a.f \rangle$ and $\langle b.f \rangle$ from statements $a.f = x$ and $b.f = y$, respectively, where a and b differ. Even though they differ, $a.f$ and $b.f$ may still resolve to the same instance fields, causing objects that flow to them to converge at those fields. Thus, using an identity relation on escape sites in our NFA equivalence checking misses objects that are truly early-confluent, causing Valve to overlook merging opportunities that might be beneficial for efficiency.

However, perfectly identifying all such cases is challenging in general, as it would require us to solve the exact first-escaped pointers, which is what we are trying to avoid in the first place. We therefore propose two techniques to normalize the escape sites, so as to relax the strict identity relation on them. Specifically, two escape sites are deemed equivalent if and only if they become identical after the normalization.

Must-Alias Variable Substitution. The first normalization technique is required when variables in the escape sites have distinct names, but refer to the same objects. Back to the example on $a.f$ and $b.f$. If a and b are must-aliases—i.e., they are guaranteed to refer to the same object o—then $a.f$ and $b.f$ resolve to the same instance field $o.f$, causing objects that flow to $a.f$ and $b.f$ to converge at that field.

To capture such scenarios, we define two escape sites as equivalent if they become identical by replacing each variable with the representative of its must-alias equivalence class. To achieve this, Valve employs a lightweight must-alias analysis that determines when two variables are must aliases, following the approach described in [45]. Specifically, the must-alias analysis is performed as a lightweight pre-pass before NFA construction. It scans all program statements and constructs must-alias information. During equivalence checking, whenever Valve compares two escape sites, it first replaces every variable appearing in the escape sites with the representative of its corresponding must-alias equivalence class. The resulting normalized escape sites are then compared syntactically.

Context-Irrelevant Argument Filtering. The second normalization technique relaxes the identity-based equivalence relation used for escape sites at call sites. Under the strict relation, corresponding arguments in two escape sites must be either identical or must-alias. However, some arguments may be irrelevant to the first-escaped pointers associated with the escape sites: they do not affect how the corresponding call sites are dispatched in pointer analysis.

This observation follows from how first-escaped pointers are determined. For an escape site at a call site, the corresponding first-escaped pointer is determined by the target method and context to which the call resolves. An argument never affects the target method, and in some cases it does not affect the context either. For instance, under object-sensitive pointer analysis, the context depends only on the receiver object, not on the arguments; in such cases, the context-irrelevant argument can be safely ignored when comparing escape sites.

Since there are many variants of context sensitivity, deriving an optimal, general-purpose procedure for identifying all context-irrelevant arguments is likely intractable. Instead, we approximate this decision using Cut-Shortcut's principled pattern-based analysis [37], which efficiently identifies arguments whose propagation should be distinguished across specific contexts to preserve pointer-analysis precision. If an argument is selected by Cut-Shortcut, it indicates that the argument should propagate under a distinct context (intuitively, the argument itself serves as a context element), and we therefore retain it in the escape site. Otherwise, we exclude the argument when checking escape-site equivalence.

4.4 Soundness and Precision

This section briefly discusses the soundness and precision of pointer analysis using the heap abstraction produced by Valve compared to pointer analysis using the allocation-site-based heap abstraction. The soundness of Valve-enhanced pointer analysis is straightforward to establish. Because Valve constructs a heap abstraction that is strictly coarser than the allocation-site-based heap, all points-to relations are preserved under this coarser abstraction. Specifically, if a pointer p points to an object o without merging, then after merging it either continues to point to the same object o (if o is not merged), or to a merged abstract object o' that incorporates o.

As for precision, Mahjong, which focuses on type-dependent clients, still incurs some precision loss for such clients. Since Valve targets a more general notion of precision beyond type-dependent clients, it faces a more challenging task of preserving precision. First, precision loss in the pre-confluence region described in Section 3.2 is generally unavoidable when merged objects have distinct A-LHS variables. However, Valve's early-confluence merging criterion confines pre-confluence regions to a small region of the program, and hence introduces limited imprecision due to merging. Second, we do not explicitly handle indirect imprecision introduced by field store operations. Nevertheless, as discussed in Section 3.3, such indirect imprecision can be largely confined by our handling of field load operations. These potential sources of imprecision are reflected in our empirical evaluation of Valve in Section 5, and the results indicate that such cases are rare in practice, resulting in nearly negligible imprecision.

5 Evaluation

This section empirically evaluates Valve in terms of both precision and efficiency. As Valve is a heap abstraction based on object merging, we compare it against the state-of-the-art merging-based heap abstraction, Mahjong [51], which has demonstrated significant speedups in pointer analysis while maximally preserving precision for type-dependent clients. In addition, we include the standard allocation-site abstraction (i.e., no merging) as a baseline. This baseline enables us to quantify both the precision impact and the performance gains introduced by object merging.

To comprehensively assess the impact of Valve across *modern* context-sensitivity pointer-analysis techniques, which generally provide substantially better scalability than conventional context-sensitivity analyses while still maintaining high precision, we combine both Valve and Mahjong with three representative state-of-the-art pointer analyses and evaluate them on large-scale Java programs. First, we consider two selective context-sensitive analyses: Zipper^e [31], which emphasizes efficiency, and Moon [65], which prioritizes precision. These analyses represent two distinct

design points along the efficiency-precision spectrum of selective context sensitivity. Second, we combine Valve and Mahjong with Cut-Shortcut [37], a recent pointer-analysis technique that achieves very high efficiency while maintaining competitive precision. Importantly, Cut-Shortcut scales to large programs where advanced selective context-sensitive analyses fail to handle and, in our experiments, runs even faster than context-insensitive pointer analysis on such programs. We exclude conventional k-context-sensitive analyses (e.g., k-object-sensitive analyses [38]) because they do not scale to most programs in our benchmarks to achieve comparable precision.

Specifically, our evaluation is designed to answer the following research questions.

RQ1. How does Valve compare with Mahjong in terms of precision and efficiency when combined with state-of-the-art selective context-sensitive analyses (Zipper^e and Moon)?

RQ2. For large-scale Java programs that selective context-sensitive analyses cannot scale, how does Valve compare with Mahjong in terms of precision and efficiency when combined with Cut-Shortcut?

RQ3. How does Valve behave in practice with respect to object merging, particularly in terms of the extent and characteristics of the merged objects?

5.1 Experimental Setup

Implementation. To ensure a fair and consistent comparison, we implement and execute all evaluated pointer analyses within the same framework, since running them in different frameworks may introduce differences unrelated to the techniques themselves due to variations in IR, solver implementations, or engineering optimizations. We implement Valve on top of Tai-e [49], which already provides mature implementations of Mahjong, Zipper^e, and Cut-Shortcut. For a uniform experimental setting across all techniques, we port Moon to Tai-e. Moon selects the objects to be analyzed context-sensitively. We adopt its original artifact implementation to guide selective context-sensitive analysis in Tai-e, ensuring that the results faithfully reflect Moon's effectiveness. The full implementation is included in our artifact and released as open source [58].

Benchmarks. We evaluate Valve on 17 large Java programs. 15 of them are the largest benchmarks adopted from recent literature [37, 61, 65], and the remaining 2 are large real-world Java projects: h2o and flink. These programs span across benchmark suites including DaCapo [7, 8] and Renaissance [43], as well as standalone projects such as flink [11] (stream processing framework), columba (email client) and soot (Java analysis framework). These programs are large and complex; in particular, the largest ones cannot be analyzed to completion within a few hours even by context-insensitive pointer analysis. To best match each program's runtime environment, we analyze h2o, flink, biojava, als, and chirper with JDK 17, which these projects target, and the other programs with JDK 1.6 as in previous literature.

Precision Metrics. To assess pointer-analysis precision under different heap abstractions, we adopt four representative client analyses that are widely used in prior work [36–38, 41, 51, 61, 65, 68]. Specifically, we measure: (1) cast-resolution analysis, by the number of may-**F**ailing **C**asts (#**FC**); (2) call-graph construction, by the number of **C**all-graph **E**dges (#**CE**); (3) **T**hread-**E**scape analysis, by the number of objects that may escape their allocation thread (#**TE**); and (4) **S**ide-**E**ffect analysis, by the average number of objects each method may modify (#**SE**). Since object merging preserves soundness, it can only over-approximate these client results; thus, smaller values consistently indicate higher precision across all metrics.

These clients also capture two qualitatively different notions of precision. #FC and #CE are type-dependent: their results depend only on the types of the objects that may appear in a pointer's points-to set. For example, cast-resolution analysis (#FC) checks `x = (T)y` by testing whether any

possible object type in y's points-to set is incompatible with T, regardless of which specific objects are involved. In contrast, #TE and #SE are non-type-dependent: they require finer-grained points-to information about which objects each pointer may reference (e.g., whether a particular object may escape or be modified). Consequently, they impose substantially stronger demands on points-to precision and more directly reflect object-identity imprecision introduced by heap abstraction.

Analysis Time. To evaluate efficiency, we measure the end-to-end speedup achieved by analyses enhanced with Valve or Mahjong, including the overhead of computing the heap abstraction (i.e., performing object merging). We define speedup as $T_{\text{pta}}/(T_{\text{pta}}' + T_{\text{merge}})$, where T_{pta} denotes the pointer analysis time without merging, T_{pta}' the pointer analysis time with merging, and T_{merge} the time spent computing the heap abstraction.

Experimental Configuration. All experiments are conducted on a machine equipped with an Intel Xeon 2.30 GHz CPU. For each analysis, we allocate 256 GB of memory and use 8 threads; the pointer analyses run single-threadedly, while Valve and Mahjong exploit parallelism only for computing heap abstractions. We set the time budget to 12 hours and report the average runtime over 3 runs.

We adopt 3-object sensitivity for Zipper^e and Moon, while Cut-Shortcut is intentionally designed to operate without contexts [37]. Object sensitivity has been widely validated as effective for object-oriented languages [28, 38, 46]. We set $k = 3$ as it provides high precision in practice; larger k is not supported by Moon and also causes Zipper^e to exceed our time and memory budget.

5.2 RQ1: Valve vs. Mahjong for Selective Context Sensitivity

This section compares Valve to Mahjong in terms of precision and efficiency when combined with state-of-the-art selective context-sensitive analyses (Zipper^e and Moon), using the standard allocation-site abstraction (Alloc) as a baseline.

Combining three heap abstraction techniques with two context sensitivity configurations yields six analyses in total. Table 1 reports the results of these analyses across all benchmarks, except four (h2o, flink, als, and chirper) that do not scale with either Zipper^e or Moon. For each analysis and each benchmark, we report the total analysis time T_{total}, which consists of the time spent computing the heap abstraction T_{merge} and the pointer analysis time with merging T_{pta}'. We also report four precision metrics: #TE, #SE, #FC and #CE, as described in Section 5.1. Notably, row CI in the table corresponds to the result of context-insensitive pointer analysis using Alloc.

Efficiency. When combined with selective context-sensitive analyses, Valve-enhanced analyses achieve comparable or often superior efficiency to Mahjong. With the efficiency-oriented analysis Zipper^e (scaling on 11 programs averaging 1273 seconds), Valve achieves an average speedup of 34.7×, compared with Mahjong's 13.3×. With the precision-oriented Moon (scaling on 9 programs averaging 2223 seconds), Valve attains a speedup of 2.8×, versus Mahjong's 1.1×.

Notably, for the especially large programs (9 of 13 benchmarks) where Zipper^e and Moon cannot finish within 5 minutes, the average speedups achieved by Valve become even larger. With the efficiency-oriented Zipper^e, Valve achieves an average speedup of 41.3×, versus Mahjong's 4.3×. With the precision-oriented Moon, Valve attains an average speedup of 4.3×, versus Mahjong's 1.2×.

The efficiency of Valve stems from its methodology: (1) its unique merging criterion enables a lightweight analysis to identify mergeable objects; (2) this distinct criterion also leads Valve to merge a more diverse set of objects compared to Mahjong, resulting in more pronounced improvements. We discuss each of these two factors below.

First, by focusing on early-confluent objects, Valve can perform efficient merging using allocation-local flow information derived from lightweight program structures, without requiring a whole-program pointer analysis; in contrast, Mahjong depends on such a whole-program analysis to

Table 1. Efficiency and precision results for CI, Zipper^e and Moon, using Alloc, Mahjong, and Valve. OoM indicates the analysis crashes after running out of memory. Entries marked with '+' show changes relative to the Alloc baseline. For all metrics, smaller is better.

		Zipper^e [31]					Moon [65]				
	Analysis	T_{total} (T_{pta}' + T_{merge})	#TE	#SE	#FC	#CE	T_{total} (T_{pta}' + T_{merge})	#TE	#SE	#FC	#CE
gruntspud	CI	27	14715	7873	6710	274653	27	14715	7873	6710	274653
	Alloc	641	12333	7207	5131	225279	OoM	–	–	–	–
	Mahjong	166 (67+98)	+4753	+1079	+22	+2382	OoM	–	–	–	–
	Valve	34 (27+7)	+9	+15	+0	+2	OoM	–	–	–	–
freecol	CI	32	16744	12040	9625	312882	32	16744	12040	9625	312882
	Alloc	565	13966	10848	6657	267290	OoM	–	–	–	–
	Mahjong	151 (21+130)	+9312	+832	+107	+294	OoM	–	–	–	–
	Valve	36 (28+8)	+4	+36	+0	+36	OoM	–	–	–	–
briss	CI	33	17521	11214	7752	294616	33	17521	11214	7752	294616
	Alloc	636	14750	10369	5838	259645	OoM	–	–	–	–
	Mahjong	176 (72+104)	+7283	+1070	+26	+269	OoM	–	–	–	–
	Valve	43 (36+8)	+15	+33	+29	+0	OoM	–	–	–	–
columba	CI	63	19686	11941	10413	419632	63	19686	11941	10413	419632
	Alloc	1473	17021	10962	7867	328611	OoM	–	–	–	–
	Mahjong	1561 (1295+266)	+5098	+816	+9	+331	OoM	–	–	–	–
	Valve	1286 (1277+8)	+5	+14	+0	+8	OoM	–	–	–	–
hsqldb	CI	3	3078	1768	1635	63720	3	3078	1768	1635	63720
	Alloc	3201	2485	1521	823	56219	689	2395	869	777	55784
	Mahjong	199 (195+4)	+1391	+426	+1	+3	617 (613+4)	+1417	+589	+0	+5
	Valve	14 (9+5)	+3	+5	+13	+0	258 (254+5)	+7	+26	+13	+17
jedit	CI	13	8046	4491	4056	150296	13	8046	4491	4056	150296
	Alloc	10	6455	3940	2821	122800	1707	6200	3436	2493	121191
	Mahjong	41 (8+33)	+2580	+511	+0	+8	1812 (1779+33)	+2668	+730	+0	+8
	Valve	13 (8+5)	+3	+12	+0	+7	968 (963+5)	+30	+35	+10	+45
h2	CI	4	3635	1968	1857	99945	4	3635	1968	1857	99945
	Alloc	OoM	–	–	–	–	2956	2840	1507	1124	89584
	Mahjong	18 (12+6)	4425	2096	1206	90224	2548 (2542+6)	+1439	+448	+0	+2
	Valve	45 (40+5)	3041	1738	1206	90224	861 (856+5)	+5	+16	+0	+4
soot	CI	39	11102	6504	16633	415845	39	11102	6504	16633	415845
	Alloc	4534	7003	5174	8760	270801	3979	7264	2609	8601	272715
	Mahjong	4317 (4234+83)	+1917	+438	+0	+11	3675 (3592+83)	+1950	+746	+0	+11
	Valve	1505 (1493+11)	+28	+12	+10	+3	1310 (1298+11)	+18	+36	+0	+5
batik	CI	6	7416	3236	3285	127819	6	7416	3236	3285	127819
	Alloc	868	5924	2571	2057	110012	220	5201	1105	1954	108163
	Mahjong	105 (91+14)	+2618	+604	+0	+9	230 (216+14)	+2663	+825	+0	+11
	Valve	195 (189+6)	+8	+15	+0	+0	306 (300+6)	+18	+43	+0	+40
findbugs	CI	4	5752	1777	3370	107446	4	5752	1777	3370	107446
	Alloc	578	4712	1400	2288	93421	11	3730	831	1639	88288
	Mahjong	12 (5+8)	+2384	+596	+0	+1	16 (8+8)	+2457	+706	+5	+17
	Valve	15 (9+6)	+4	+5	+0	+1	16 (10+6)	+7	+33	+0	+4
checkstyle	CI	3	4570	1391	1825	80611	3	4570	1391	1825	80611
	Alloc	555	3804	1125	1207	69594	11	2616	774	930	66536
	Mahjong	9 (2+7)	+1791	+600	+0	+0	14 (7+7)	+1729	+677	+5	+20
	Valve	11 (5+6)	+0	+7	+0	+0	14 (8+6)	+2	+20	+0	+3
biojava	CI	815	11061	5651	5710	265876	815	11061	5651	5710	265876
	Alloc	953	10718	5431	4860	240955	7769	9514	4663	3569	202711
	Mahjong	1436 (493+943)	+2537	+726	+0	+2	5334 (4390+943)	+2868	+890	+50	+30
	Valve	869 (857+13)	+36	+21	+87	+2745[1]	5865 (5852+13)	+12	+37	+0	+6
eclipse	CI	13	9020	5353	5074	184970	13	9020	5353	5074	184970
	Alloc	OoM	–	–	–	–	2673	6613	4003	3507	163831
	Mahjong	OoM	–	–	–	–	2175 (2136+39)	+2883	+834	+0	+8
	Valve	2678 (2674+4)	7301	4568	3640	165230	245 (241+4)	+23	+40	+3	+20

support its merging. For example, in columba, Mahjong requires 266 seconds to identify and merge type-consistent objects, while Valve identifies and merges early-confluent objects in only 8 seconds.

[1]Due to space constraints, we discuss the reason for this unexpected precision loss in the supplementary material.

The second reason is more subtle. Valve adopts a different merging criterion from Mahjong, resulting in substantially different sets of merged objects (detailed in Section 5.4), which affect analysis efficiency differently. Valve identifies a diverse set of mergeable objects, including those whose invoked methods are selected for context-sensitive treatment by selective context-sensitive techniques. Merging such objects reduces the number of contexts for these methods, thereby accelerating the analysis. In contrast, for many objects merged by Mahjong, their invoked methods are already treated context-insensitively by selective context-sensitive techniques, largely diminishing the potential efficiency benefits of merging. Consequently, merging these objects tends to yield more modest improvements than those achieved by Valve. Additionally, the objects merged by Valve often involve more complex instance-field usage than those merged by Mahjong. Since object merging also merges their associated instance fields, Valve can reduce more instance fields that the enhanced pointer analysis needs to distinguish and process, thereby yielding greater efficiency improvements.

Precision. For non-type-dependent clients, Valve substantially outperforms Mahjong, preserving a much higher average precision of 99.35%, whereas Mahjong achieves only 73.52%. Even for type-dependent clients that Mahjong prioritizes over others, Valve still achieves a comparable average precision of 99.82% compared to Mahjong's 99.84%. The potential sources of precision loss in Valve have been discussed in Section 4.4.

Valve's high precision stems from its object-merging strategy. By merging only early-confluent objects—objects whose flows quickly converge and then propagate together—Valve confines the imprecision introduced by merging to a small localized region of the PFG. Consequently, it avoids the cascading propagation of spurious points-to information in general and achieves higher precision in practice, independently of downstream clients. In contrast, Mahjong's type-consistency criterion enables each object merge to preserve the precision of type-dependent clients [51]. While effective for such clients, it does not preserve more general object-flow information, which can substantially affect the precision of non-type-dependent clients.

We further compare Valve and Mahjong on three additional common metrics: devirtualized calls, average points-to set size, and the number of may-alias variable pairs. The average precision values are 99.96% vs. 99.81% (devirtualized calls), 98.54% vs. 35.22% (average points-to set size), and 99.66% vs. 93.28% (the number of may-alias variable pairs), respectively (detailed results are provided in the supplementary material due to space constraints). These results further demonstrate that Valve can maintain high precision in a client-independent manner.

5.3 RQ2: Valve vs. Mahjong for Cut-Shortcut

This section compares Valve to Mahjong in terms of precision and efficiency when combined with Cut-Shortcut. Cut-Shortcut can be viewed as an alternative pointer analysis that prioritizes scalability: selective context-sensitivity approaches such as Zipper^e and Moon typically achieve higher precision, but are considerably less efficient than Cut-Shortcut.

We therefore focus on extremely large programs where selective context sensitivity struggles to scale but Cut-Shortcut remains scalable, to evaluate whether Valve can accelerate Cut-Shortcut in the settings where it is most applicable. Table 2 reports the results of the three heap-abstraction techniques (Alloc, Mahjong and Valve) under Cut-Shortcut on the 4 largest benchmarks where Zipper^e and Moon both fail to scale (cannot finish analysis within 12 hours or run out of memory).

Accelerating Cut-Shortcut poses unique challenges compared with selective context-sensitive analyses. First, it already exhibits exceptional efficiency: as shown in Table 2, it is even faster than CI. Second, unlike context-sensitive techniques that analyze each method multiple times under different contexts and thus instantiate many objects, Cut-Shortcut analyzes each method

Table 2. Results for the 4 largest programs under CI, Cut-Shortcut with Alloc, Mahjong, and Valve; these programs remain unscalable even for state-of-the-art selective CS approaches such as Zipper^e and Moon.

	Analysis	T_{total} ($T_{\text{pta}}{}'$ + T_{merge})	#TE	#SE	#FC	#CE
h2o	CI	>12h	–	–	–	–
	Alloc	31492	27350	16438	11806	4331899
	Mahjong	>12h	–	–	–	–
	Valve	23916 (23866+49)	+45	+351	+1	+83
flink	CI	19667	23101	12653	12859	721542
	Alloc	5034	20423	11532	7126	492573
	Mahjong	23390 (517+22872)	+4756	+1692	+43	+388
	Valve	3506 (3415+91)	+17	+93	+0	+5
als	CI	15336	53549	29328	47770	3431659
	Alloc	8066	46568	26194	35201	2581138
	Mahjong	24742 (2572+22170)	+5995	+1999	+56	+1775
	Valve	5625 (5577+47)	+73	+65	+38	+102
chirper	CI	413	15733	8277	9943	512580
	Alloc	148	15199	8111	7140	429798
	Mahjong	855 (98+757)	+1878	+673	+6	+315
	Valve	124 (113+10)	+7	+26	+2	+0

only once. This results in fewer objects being propagated on the PFG, making it harder to achieve significant speedups through object merging.

Efficiency and Precision. Despite the limited optimization opportunities, Valve pushes the efficiency boundaries further. Across the four largest benchmarks, Valve reduces the analysis time of Cut-Shortcut by 2893 seconds on average (up to 7576 seconds), yielding a modest but practically meaningful average speedup of 1.34×. In contrast, Mahjong incurs a slowdown in this setting. This is because Mahjong first relies on a whole-program context-insensitive pointer analysis to identify type-consistent objects; for programs of this scale, the cost of this pre-analysis becomes significant and outweighs the efficiency benefits brought by Mahjong's heap abstraction.

As for precision, Valve achieves higher precision than Mahjong on all four clients across all programs in Table 2. For non-type-dependent clients, Valve substantially outperforms Mahjong, preserving a much higher average precision of 99.51%, whereas Mahjong achieves only 88.53%. Even for type-dependent clients that Mahjong prioritizes over others, Valve still achieves a better average precision of 99.98% compared to Mahjong's 99.82%.

5.4 RQ3: How Does Valve Merge Objects in Practice?

This subsection examines how Valve merges *early-confluent objects* in our experiments. We focus on three aspects: (1) the *extent* of merging, i.e., how many abstract heap objects are eliminated compared to the standard allocation-site abstraction; (2) the *structure* of merging, i.e., what the merged groups (equivalence classes) look like in terms of their number and size; and (3) the *equivalence relations* induced by merging, i.e., how the object pairs merged by Valve compare with those merged by Mahjong.

Extent of Merging. Figure 10 reports the number of heap objects under different heap abstractions, including Alloc, Valve, and Mahjong. We exclude h2o because Mahjong requires a context-insensitive pointer analysis (CI) as a pre-analysis step, and CI does not scale to it. Overall, relative to Alloc, Valve reduces the number of heap objects by 26.6% per program on average, ranging from 18.9% on chirper to 35.8% on freecol. As a reference point, Mahjong achieves an average reduction of 44%, and, except for soot, it merges more objects than Valve on every program.

Interestingly, although Valve typically merges *fewer* objects than Mahjong, it can still deliver *higher* speedups (as shown in Sections 5.2 and 5.3). This suggests that the acceleration from object

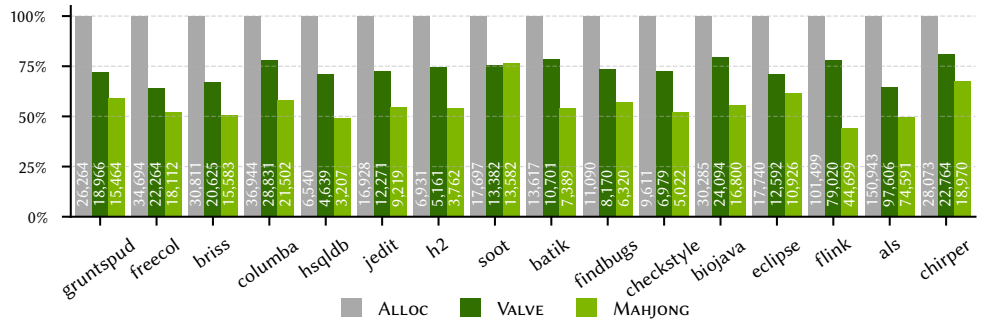


Fig. 10. Number of abstract objects created by Alloc, Valve, and Mahjong, normalized to the highest.

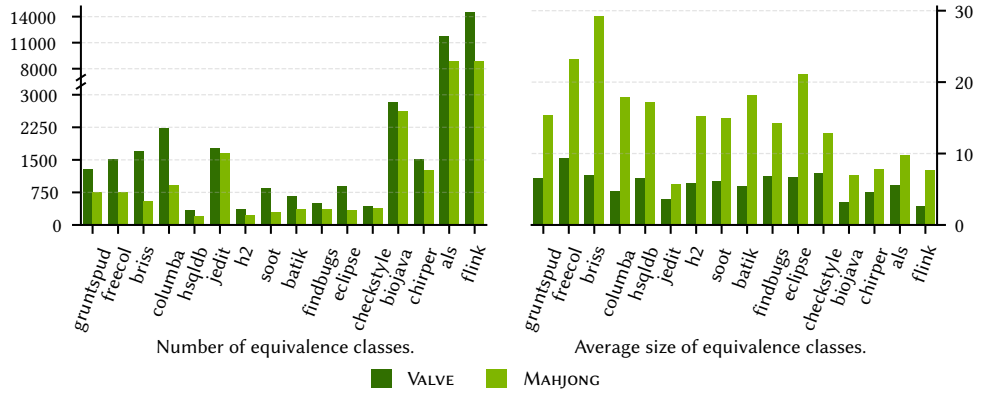


Fig. 11. Characteristics of merged objects by Valve and Mahjong. The left chart shows the number of equivalence classes merged per program, and the right chart shows the average size of equivalence classes.

merging is not determined solely by *how many* objects are merged, but also by *which* objects are merged. Intuitively, merging objects that quickly become confluent tends to reduce redundant points-to propagation along the most shared flow paths, thereby improving efficiency even when the total reduction in object count is relatively modest.

Structure of Merged Objects. To further characterize the merging induced by Valve, we analyze the *equivalence classes* of early-confluent objects identified by Valve in each program. Concretely, Valve groups mutually early-confluent objects into one equivalence class and merges all objects in the class into a single abstract object.

Figure 11 reports, for each program, (i) the number of equivalence classes produced by Valve and (ii) the average equivalence-class size (i.e., the number of objects in a class). We include the same statistics for Mahjong for comparison; Mahjong merges objects according to its *type-consistency* criterion [51], and each class corresponds to a set of objects merged by Mahjong. The results show a clear contrast: despite merging fewer objects overall, Valve partitions the objects into noticeably more equivalence classes on average (Valve: 2682 vs. Mahjong: 1772), and its average equivalence-class size is significantly smaller (Valve: 5.7 vs. Mahjong: 14.8). In other words, compared to Mahjong, Valve's merging is more fine-grained: merging decisions are more *distributed* across the heap (more classes), while each merged group is more *cohesive* (smaller classes).

This behavior is consistent with Valve's early-confluence criterion. If multiple objects become confluent shortly after allocation, then before confluence they typically traverse nearby and closely

related regions of the program; merging them therefore confines the introduced imprecision to a small pre-confluence portion of the pointer flows. This merging behavior helps explain the consistently good precision achieved by Valve, as empirically demonstrated in Sections 5.2 and 5.3.

Comparison of Equivalence Relations. We also conduct a fine-grained comparison between the equivalence relations produced by Valve and Mahjong. Specifically, for each technique, we record the set of object pairs identified as equivalent and therefore merged. We then compare the intersection and difference between the sets produced by Valve and Mahjong, in order to better understand the similarities and differences between their merging strategies.

For object pairs merged by both Valve and Mahjong, on average, they account for 56.5% of the pairs merged by Valve and 35.0% of those merged by Mahjong. This result indicates that the two techniques identify substantially different and cross-cutting merge opportunities, rather than one being simply a refinement of the other.

For object pairs merged exclusively by each, we further inspected the top-5 most frequently merged object types for each technique. Overall, the types of objects merged only by Valve and those merged only by Mahjong are quite different, due to their different merging criteria. The types merged only by Valve vary across benchmark programs, whereas those merged only by Mahjong are mostly String and primitive-array types across different programs.

For example, on the eclipse benchmark, among the object pairs merged only by Mahjong, the top-5 types are `java.lang.String`, `java.lang.String[]`, `char[]`, `int[]`, and `java.io.File`. In contrast, among the object pairs merged only by Valve, the top-5 types are `java.lang.Object[]`, `org.eclipse.Status`, `java.lang.String[]`, `org.eclipse.JavaModelStatus`, and `org.eclipse.Label`, including several application-specific types from the eclipse benchmark, demonstrating the diversity of the mergeable objects identified by Valve.

Discussion. Given that Valve and Mahjong identify cross-cutting merge opportunities, it is natural to ask whether combining them can yield a more favorable precision-efficiency tradeoff. However, because these techniques can influence each other, their combinations can be varied. We briefly discuss two combination variants below, leaving a systematic exploration of the rich interactions among multiple heap abstractions as an interesting direction for future work.

The first combination merges only objects that are both early-confluent and type-consistent, by running both techniques *independently* over the original allocation-site-based heap and intersecting their equivalence relation. Since this combination is a refinement over Valve and Mahjong, it is expected to deliver better precision with less efficiency benefits than either technique alone. The second combination applies Mahjong and Valve *sequentially*. For example, we may first run Mahjong to obtain a coarsened heap and then feed it into Valve for further coarsening. Note that this is distinct from the previous combination: because the coarsening performed by Mahjong is observed by Valve, Mahjong affects which objects Valve identifies as mergeable. On a coarser heap, more objects become early-confluent, enabling Valve to discover additional merging opportunities beyond using each technique alone, with the resulting heap embodying imprecision introduced by both techniques. However, given Valve and Mahjong's good precision on type-dependent clients, it is expected to not lose much precision for type-dependent clients.

5.5 Ablation Study

To further understand the effectiveness of our two relaxation techniques (Section 4.3.2), we conduct an ablation study over three incremental configurations: (1) *Strict*, which uses identity-based escape-site equivalence only, i.e., no relaxation; (2) *+MustAlias*, which additionally enables must-alias variable substitution; and (3) *+Filtering*, which further adds context-irrelevant argument filtering on top of (2). For each configuration, we report the speedup over the no-merge baseline and the

resulting precision (i.e., the ratio between the precision metric of the no-merge baseline and that of the corresponding configuration). Due to space constraints, we summarize only the average results here; detailed per-benchmark results are provided in the supplementary material.

Across the three configurations (*Strict*, *+MustAlias*, and *+Filtering*), the speedups are 1.15×, 3.43×, and 45.91× for Zipper^e, with precisions of 99.96%, 99.88%, and 99.76%, respectively. For Moon, the corresponding speedups are 1.03×, 1.12×, and 2.87×, while the precisions are 99.92%, 99.92%, and 99.35%. For Cut-Shortcut, the speedups are 0.96×, 1.06×, and 1.38×, with precisions of 99.88%, 99.86%, and 99.74%. Across all three modern pointer analyses, the two relaxation techniques contribute additional merging opportunities and efficiency gains, while the resulting precision loss remains negligible. The slightly sub-unit speedup of *Strict* on Cut-Shortcut is likely due to run-to-run performance variation.

Additionally, we examine how the two relaxation techniques affect the extent of merging itself by measuring the remaining heap size. Relative to the no-merge baseline, *Strict* reduces the remaining heap size to 87.6%; *+MustAlias* lowers it further to 84.0%; and *+Filtering* brings it down further to 73.4%. This progressive reduction demonstrates that each relaxation technique uncovers additional early-confluent objects that the strict identity relation alone would miss.

6 Related Work

We discuss two major directions for improving the efficiency and precision of Java pointer analysis: (1) designing better heap abstractions and (2) improving context sensitivity.

Heap Abstractions. Type-based heap abstraction [48] can be seen as merging all objects of the same type. It achieves better precision than CHA [16, 18] and RTA [6] with high efficiency, and is therefore used in production compilers [61]. However, it is substantially less precise than the allocation-site-based abstraction [22], which remains predominant for precise pointer analysis [51].

Tan et al. [51] proposed Mahjong to merge type-consistent objects for type-dependent clients. Mahjong encodes field points-to relations of a context-insensitive pointer analysis and object types as NFAs, and reduces type-consistency checking to NFA equivalence testing. Valve also leverages an NFA-based encoding, but from a fundamentally different perspective. Instead of reasoning about types in field points-to sets, Valve reasons about object-flow confluence, encoding object-flow information in the NFAs to determine early confluence, without a whole-program pointer analysis.

Xu and Møller [63] proposed an adaptive heap abstraction for JavaScript pointer analysis that merges objects when points-to sets grow large. An appealing aspect is that, similar to Valve, their approach does not rely on a separate whole-program pointer analysis. However, this method makes merging decisions on-the-fly, based on internal metrics during pointer analysis, whereas Valve identifies early-confluent objects beforehand, via object-flow information.

For machine-learning-based techniques, Jeon et al. [22] proposed Graphick, a general framework that learns so-called graph-based heuristics to optimize pointer analysis. When applied to Mahjong's field points-to graph, it yields a heap abstraction comparable to Mahjong in precision and efficiency. Similarly, Chen et al. [15] presented 4DM, a learning-based technique for optimizing Datalog-based program analyses that can also produce a heap abstraction by learning a merging heuristic targeting objects in the JDK. Unlike these learning-based approaches, Valve is designed under the insight of merging early-confluent objects, and derives merging decisions directly from early flow confluence. By targeting objects that converge quickly to the same set of pointers, Valve is able to confine the imprecision due to merging while accelerating the pointer analysis. Additionally, learning-based approaches share a limitation by relying on a training phase (e.g., Graphick requires approximately 200 hours to train on four selected programs [22]). Generally, learning-based approaches are sensitive to the training programs and metrics, and the learned

heuristics are usually difficult to comprehend and explain [31]. As a training-free and insight-driven approach, Valve is complementary to these approaches. For example, Valve's insight into merging early-confluent objects may provide a useful perspective for designing learning objectives in learning object-merging heuristics for heap abstractions.

Abstraction refinement [32, 64, 67] is a general technique for improving static analysis precision and can be applied to construct heap abstractions for pointer analysis. However, its goal fundamentally differs from ours. Refinement-based approaches start from a coarse abstraction and iteratively refine it to improve analysis precision, while Valve takes the opposite direction: starting from an allocation-site-based heap abstraction, it merges early-confluent objects to coarsen the heap to accelerate pointer analysis while preserving precision. Despite this contrast, it would be interesting to combine these two approaches. For example, refinement could focus on objects that are not identified by Valve, as early-confluent objects merged by Valve tend to have negligible impact on precision and are therefore less likely to benefit from further refinement.

Vivien and Rinard [57] proposed to identify escape sites through an expanding pointer analysis for compiler optimization purposes. Valve also identifies escape sites to merge early-confluent objects, but adopts a more lightweight approach without a pointer analysis.

Context Sensitivity. Context-sensitivity has been proven as an effective approach to improve the precision of Java pointer analysis while maintaining efficiency, attracting substantial research attention in recent years. Prior work on context sensitivity largely falls into two categories: (1) selecting specific elements as contexts, such as objects [38], call-sites [17, 23], types [46], context transformations [54], value-contexts [42, 52], or combinations thereof [25, 30, 53]; (2) selectively applying context sensitivity to objects [65], methods [60], or variables [36], guided by heuristics [47], patterns [21, 31], machine learning [22], or combinations thereof [25, 50]. More recently, Ma et al. [37] proposed Cut-Shortcut, obtaining precision comparable to context sensitivity via PFG manipulation, at the cost of context insensitivity. Among these, LSRV [52] is technically close to Valve. It uses levelized summaries, a technique similar to NFAs, to abstract the impact of value contexts, thereby deduplicating redundant contexts. In contrast, Valve leverages NFAs to encode object-flow information and compares them to merge objects to coarsen the heap. These techniques have undoubtedly advanced pointer analysis.

Valve, as a heap abstraction, is orthogonal and complementary to these advancements. As is demonstrated by our evaluation in Section 5 using three representative context-sensitivity techniques—Zipper^e [31], Moon [65] and Cut-Shortcut [37]—with distinct principles and tradeoffs, the integration yields substantial efficiency gains with nearly negligible precision loss.

7 Conclusions

This paper proposed Valve, a heap abstraction that improves pointer analysis efficiency while preserving precision in a client-independent manner by identifying and merging early-confluent objects—objects that quickly converge to the same pointers and propagate together. By approximating early-confluence determination via an NFA equivalence problem, Valve identifies early-confluent objects efficiently while retaining high precision. Our extensive evaluation on large-scale Java benchmarks shows that Valve consistently improves the efficiency–precision tradeoff. Compared with the state-of-the-art merging approach Mahjong, Valve achieves substantially higher precision for non-type-dependent clients while maintaining comparable precision for type-dependent ones. Meanwhile, it matches or often surpasses Mahjong in efficiency across multiple state-of-the-art pointer-analysis techniques with different design principles and precision-efficiency tradeoffs. We expect Valve to highlight the importance of heap abstraction in Java pointer analysis and to serve as a useful reference for future work in this area.

8 Data Availability Statement

We have provided an artifact [58] to reproduce all experimental results presented in Section 5; it is available at https://doi.org/10.5281/zenodo.21508558. To reproduce the results, please follow the instructions in the `README.pdf` included in the artifact package. In addition, we have provided supplementary material [59] detailing the results of the ablation study in Section 5.5, available at https://doi.org/10.5281/zenodo.21857709.

Acknowledgments

We sincerely thank the anonymous reviewers for their thoughtful and constructive comments, which helped us strengthen the paper. We also thank Shengyuan Yang and Nairen Zhang for their helpful suggestions on developing and improving Valve. This work was supported in part by National Key R&D Program of China (Grant No. 2023YFB4503804), National Natural Science Foundation of China (Grant No. 62402210), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (Grant No. JYB2025XDXM118), and 111 Center (Grant No. B26023). We thank the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China, for its support. Tian Tan, the co-corresponding author, was also supported by the Xiaomi Foundation.

References

[1] Jiri Adamek and Vera Trnkova. 1990. *Automata and Algebras in Categories* (1st ed.). Kluwer Academic Publishers, USA.

[2] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. 2002. Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2477)*, Manuel V. Hermenegildo and Germán Puebla (Eds.). Springer, Berlin, Heidelberg, 230–246. doi:10.1007/3-540-45789-5_18

[3] Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. 2025. CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers. *Proc. ACM Program. Lang.* 9, OOPSLA2 (Oct. 2025), 2759–2785. doi:10.1145/3763149

[4] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, Edinburgh United Kingdom, 259–269. doi:10.1145/2594291.2594299

[6] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1996, San Jose, California, USA, October 6-10, 1996*, Lougie Anderson and James Coplien (Eds.). ACM, 324–341. doi:10.1145/236337.236371

[7] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 940–954. doi:10.1145/3669940.3707217

[8] 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 L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, Portland, Oregon, USA, 169–190. doi:10.1145/1167473.1167488

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, New York, NY, USA, 243–262. doi:10.1145/1640089.1640108

[10] Yuandao Cai and Charles Zhang. 2023. A Cocktail Approach to Practical Call Graph Construction. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1001–1033. doi:10.1145/3622833

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink : Stream and batch processing in a single engine. *IEEE Data(base) Engineering Bulletin* 36 (2015), 28–33. https://api.semanticscholar.org/CorpusID:263802939

[12] IBM T.J. Watson Research Center. 2025. *T.J. Watson Libraries for Analysis.* https://github.com/wala/WALA

[13] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 363–374. doi:10.1145/1542476.1542517

[14] Menglong Chen, Tian Tan, Minxue Pan, and Yue Li. 2025. PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2803–2815. doi:10.1109/ICSE55347.2025.00208

[15] Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. 2021. Accelerating Program Analyses in Datalog by Merging Library Facts. In *Static Analysis*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 77–101. doi:10.1007/978-3-030-88806-0_4

[16] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 77–101. doi:10.1007/3-540-49538-X_5

[17] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9458)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, Cham, 465–484. doi:10.1007/978-3-319-26529-2_25

[18] Mary F. Fernández. 1995. Simple and effective link-time optimization of Modula-3 programs *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 103–115. doi:10.1145/207110.207121

[19] 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 (2008), 9:1–9:34. doi:10.1145/1348250.1348255

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

[21] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 971–1000. doi:10.1145/3622832

[22] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 179:1–179:30. doi:10.1145/3428247

[23] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. doi:10.1145/3498720

[24] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. doi:10.1145/2931098

[25] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 423–434. doi:10.1145/2491956.2462191

[26] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 892–903. doi:10.1145/3597926.3598104

[27] Ondrej Lhoták and Laurie J. Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2622)*, Görel Hedin (Ed.). Springer, Berlin, Heidelberg, 153–169. doi:10.1007/3-540-36579-6_12

[28] Ondrej Lhoták and Laurie J. Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1 (2008), 3:1–3:53. doi:10.1145/1391984.1391987

[29] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2024. SPATA: Effective OS Bug Detection with Summary-Based, Alias-Aware, and Path-Sensitive Typestate Analysis. *ACM Trans. Comput. Syst.* 42, 3-4 (2024), 1–40. doi:10.1145/3695250

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, New York, NY, USA, 129–140.

doi:10.1145/3236024.3236041

[31] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. doi:10.1145/3381915

[32] Percy Liang and Mayur Naik. 2011. Scaling abstraction refinement via pruning. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 590–601. doi:10.1145/1993498.1993567

[33] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1417–1441. doi:10.1145/3729307

[34] 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, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, New York, NY, USA, 359–373. doi:10.1145/3192366.3192390

[35] 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 *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, Virtual, Canada, 725–739. doi:10.1145/3453483.3454073

[36] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 148:1–148:29. doi:10.1145/3360574

[37] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 539–564. doi:10.1145/3591242

[38] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. doi:10.1145/1044834.1044835

[39] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating abstraction overhead of Java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 168:1–168:29. doi:10.1145/3428236

[40] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, New York, NY, USA, 308–319. doi:10.1145/1133981.1134018

[41] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. 2012. Abstractions from tests. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, New York, NY, USA, 373–386. doi:10.1145/2103656.2103701

[42] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural data flow analysis in Soot using value contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) *(SOAP '13)*. Association for Computing Machinery, New York, NY, USA, 31–36. doi:10.1145/2487568.2487569

[43] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. doi:10.1145/3314221.3314637

[44] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. doi:10.1561/2500000014

[45] Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. Set-based pre-processing for points-to analysis. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 253–270. doi:10.1145/2509136.2509524

[46] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, New York, NY, USA, 17–30. doi:10.1145/1926385.1926390

[47] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, Edinburgh, United Kingdom, 485–495. doi:10.1145/2594291.2594320

[48] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. Association for Computing

Machinery, New York, NY, USA, 264–280. doi:10.1145/353171.353189

[49] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

[50] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485524

[51] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, New York, NY, USA, 278–291. doi:10.1145/3062341.3062360

[52] 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, CC 2019, Washington, DC, USA, February 16-17, 2019*, José Nelson Amaral and Milind Kulkarni (Eds.). ACM, New York, NY, USA, 135–146. doi:10.1145/3302516.3307359

[53] Manas Thakur and V. Krishna Nandivada. 2020. Mix your contexts well: opportunities unleashed by recent advances in scaling context-sensitivity. In *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, Louis-Noël Pouchet and Alexandra Jimborean (Eds.). ACM, New York, NY, USA, 27–38. doi:10.1145/3377555.3377902

[54] Rei Thiessen and Ondrej Lhoták. 2017. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, New York, NY, USA, 263–277. doi:10.1145/3062341.3062359

[55] Paolo Tonella and Alessandra Potrich. 2005. *Reverse Engineering of Object Oriented Code*. Springer. doi:10.1007/b102522

[56] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. 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, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, Mississauga, Ontario, Canada, 13. https://dl.acm.org/citation.cfm?id=782008

[57] Frédéric Vivien and Martin Rinard. 2001. Incrementalized Pointer and Escape Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 35–46. doi:10.1145/378795.378804

[58] Jinpeng Wang, Yufei Liang, Zhongsheng Zhan, Tian Tan, and Yue Li. 2026. *Heap Abstraction via Early-Confluent Object Merging for Pointer Analysis (Artifact)*. doi:10.5281/zenodo.21508558

[59] Jinpeng Wang, Yufei Liang, Zhongsheng Zhan, Tian Tan, and Yue Li. 2026. Heap Abstraction via Early-Confluent Object Merging for Pointer Analysis (Supplementary Material). doi:10.5281/zenodo.21857709

[60] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, William W. Pugh and Craig Chambers (Eds.). ACM, New York, NY, USA, 131–144. doi:10.1145/996841.996859

[61] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. 2024. Scaling Type-Based Points-to Analysis with Saturation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 990–1013. doi:10.1145/3656417

[62] Xiao Xiao, Qirun Zhang, Jinguo Zhou, and Charles Zhang. 2014. Persistent pointer information. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 463–474. doi:10.1145/2594291.2594314

[63] Wenyuan Xu and Anders Møller. 2026. JavaScript Pointer Analysis with Adaptive Heap Abstraction. *Proceedings of the ACM on Software Engineering* 3, FSE (2026). doi:10.1145/3797133

[64] Zhenyu Yan, Xin Zhang, and Peng Di. 2024. Scaling Abstraction Refinement for Program Analyses in Datalog using Graph Neural Networks. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1532–1560. doi:10.1145/3689765

[65] Chaoyue Zhang, Longlong Lu, Yifei Lu, Minxue Pan, and Xuandong Li. 2025. Towards a Theoretically-Backed and Practical Framework for Selective Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 1698–1725. doi:10.1145/3763111

[66] Teng Zhang, Yufei Liang, Ganlin Li, Tian Tan, Chang Xu, and Yue Li. 2025. Bridge the Islands: Pointer Analysis for Microservice Systems. *Proc. ACM Softw. Eng.* 2, ISSTA (2025), 504–526. doi:10.1145/3728896

[67] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 239–248. doi:10.1145/2594291.2594327

[68] Yifan Zhang, Yuanfeng Shi, and Xin Zhang. 2024. Learning Abstraction Selection for Bayesian Program Analysis. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 954–982. doi:10.1145/3649845

[69] Yiheng Zhang, Ming Wen, Shunjie Liu, Dongjie He, and Hai Jin. 2025. Precise and Effective Gadget Chain Mining through Deserialization Guided Call Graph Construction. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, USA, 2947–2964. https://www.usenix.org/conference/usenixsecurity25/presentation/zhang-yiheng

Received 2026-03-17; accepted 2026-08-06