Compare Less, Defer More

Scaling Value-Contexts Based Whole-Program Heap Analyses

Manas Thakur
Dept of CSE, IIT Madras
Chennai, TN, India
manas@cse.iitm.ac.in

V. Krishna Nandivada
Dept of CSE, IIT Madras
Chennai, TN, India
nvk@iitm.ac.in

ABSTRACT

The precision of heap analyses determines the precision of several associated optimizations, and has been a prominent area in compiler research. It has been shown that context-sensitive heap analyses are more precise than the insensitive ones, but their scalability continues to be a cause of concern. Though the value-contexts approach improves the scalability of classical call-string based context-sensitive analyses, it still does not scale well for several popular whole-program heap analyses. In this paper, we propose a three-stage analysis approach that lets us scale complex whole-program value-contexts based heap analyses for large programs, without losing their precision.

Our approach is based on a novel idea of level-summarized relevant value-contexts (*LSRV-contexts*), which take into account an important observation that we do not need to compare the complete value-contexts at each call-site. Our overall approach consists of three stages: (i) a fast pre-analysis stage that finds the portion of the caller-context which is actually needed in the callee; (ii) a main-analysis stage which uses LSRV-contexts to defer the analysis of methods that do not impact the callers' heap and analyze the rest efficiently; and (iii) a post-analysis stage that analyzes the deferred methods separately. We demonstrate the usefulness of our approach by using it to perform whole-program context-, flow- and field-sensitive thread-escape analysis and control-flow analysis of Java programs. Our evaluation of the two analyses against their traditional value-contexts based versions shows that we not only reduce the analysis time and memory consumption significantly, but also succeed in analyzing otherwise unanalyzable programs in less than 40 minutes.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Compilers**; Object oriented languages.

KEYWORDS

Static program analysis, Context-sensitivity, Value-contexts, LSRV-contexts

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
CC '19, February 16–17, 2019, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6277-1/19/02…$15.00
https://doi.org/10.1145/3302516.3307359

ACM Reference Format:
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 '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307359

1 INTRODUCTION

Heap analysis refers to a broad category of program analyses that statically approximate the information about the runtime heap of a program. For example, thread-escape analysis [2, 4] identifies objects that do not escape the thread of their allocation, interprocedural control-flow analysis [21, 25] identifies the potential targets of method calls, and so on. The precision of heap analyses determines the precision of several analyses and optimizations, and has been a prominent area in compiler research [14, 17, 27, 31, 33, 35].

The precision and scalability of interprocedural heap analyses vary based on whether the analysis is context-sensitive or not [23]. A context-insensitive analysis does not differentiate among the various calls to a method, and generates a single summary that can be used at all of its call-sites. A context-sensitive analysis, on the other hand, distinguishes between the "contexts" in which a method is called, and generates a summary for the method in each distinct context. Though the results generated by context-sensitive analyses have been shown to be more precise than context-insensitive analyses [14], the scalability of the former in analyzing large programs continues to be a cause of concern.

The classical call-strings approach [24, 25], which identifies contexts based on the call-string formed by its callers, is one of the oldest and widely-used approaches of defining the context abstraction. For example, consider the snippet of Java code shown in Figure 1a. A call-string based context-sensitive analysis would analyze the method `bar` in two contexts (created at lines 4 and 5), and the method `fb` in four contexts (two contexts for each context of `bar`). A major drawback of the call-string based approach is that in the presence of recursion and deep nesting of multiple calls, the length of the call-strings, and hence the number of contexts, may grow combinatorially. This makes the analysis unscalable to large real-world programs. Consequently, the call-string based analyses usually impose a limit on the call-string length, and treat the contexts of greater lengths conservatively. While such an approach improves the scalability of the analysis, it compromises on the resulting precision.

The clever value-contexts approach [10, 20] addresses the scalability challenges in the call-strings approach by using dataflow values to restrict the potentially unbounded growth of call-strings, without sacrificing precision. For the code shown in Figure 1a,

```
1 class A {
2   A f1,f2;
3   void foo(){ ...
4     c.bar(a);
5     d.bar(b);
6   } /*foo*/
7   void bar(A p){
8     A x = new A();
9     p.f1.f2 = x;
10    p.fb();
11    p.fb();
12  } /*bar*/
13  void fb(){
14   /*Doesn't read/affect
15   the caller's heap*/
16  } /*fb*/
17 }
```

(a)

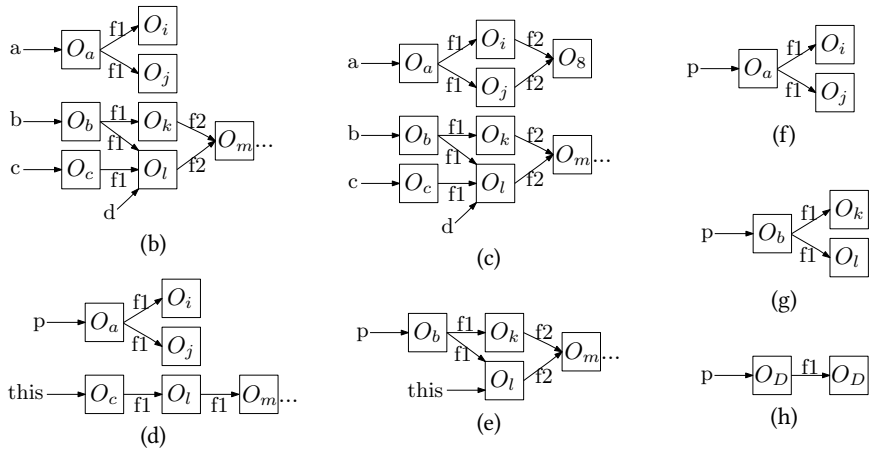


Figure 1: (a) A Java code snippet. (b) The assumed points-to graph at line 4. (c) The points-to graph at line 5. (d) The value-context for `bar` at line 4. (e) The value-context for `bar` at line 5. (f) The relevant value-context for `bar` at line 4. (g) The relevant value-context for `bar` at line 5. (h) The LSRV-context for `bar` at lines 4 and 5 (for escape analysis), assuming O_a, O_b, O_i, O_j, O_k and O_l do not escape; O_D represents a universal non-escaping object.

if Figure 1b and Figure 1c depict the points-to graphs at lines 4 and 5 respectively, then the value-contexts approach would analyze (i) `bar` in two contexts, as the value-contexts (points-to graphs reachable from formal parameters) at lines 4 and 5, shown in Figures 1d and 1e, respectively, are different; (ii) `fb` in only one context for each context of `bar`, as the points-to graphs at lines 10 and 11 match. Though the value-contexts approach is a breakthrough in performing precise context-sensitive analyses, it still does not scale for various popular heap analyses. We studied many such heap analyses and identified two main reasons for the lack of scalability: (i) time overheads involved in comparing the value-contexts and in redundantly analyzing many methods; and (ii) memory overheads due to a large number of contexts. For example, for the call to `bar` at line 5, the whole points-to graphs shown in Figures 1d and 1e are compared; in practice, depending on the size of the graphs and the number of existing contexts, this could be expensive.

In this paper, we propose several novel techniques that together let us perform complex top-down whole-program value-contexts based heap analyses for large programs in less than 40 minutes. Our scheme comprises of three analysis stages: *pre*, *main*, and *post*. The pre-analysis is a lightweight stage that gains insights about the context-dependency of all the methods of a program. It computes the portion of the caller contexts that is actually needed in the callee and stores this information as parameter-wise *access-depth*. The main-analysis uses the access-depths to not only reduce the comparison performed (compares "relevant" parts of contexts) in identifying whether two value-contexts are equal, but also in deferring the analysis of *caller-ignorable* methods that do not impact the callers' heap. Deferring reduces the overheads of analyzing those methods multiple times and in merging the results with the callers' heap, during the costly main-analysis. The main-analysis also uses a novel analysis-specific abstraction called *level-summarization* to improve the precision of identifying two contexts as equivalent. Finally, the post-analysis analyzes the deferred methods without losing precision. For the code in Figure 1a, an escape analysis based on our approach identifies that both `bar` and `fb` are invoked only in a single level-summarized relevant value-context each. Consequently, `bar` is analyzed only once in the main-analysis, and `fb` (deferred in the main-analysis) is analyzed only once in the post-analysis.

We demonstrate the effects of our proposed techniques by using them to perform fully context- and flow-sensitive thread-escape analysis and interprocedural control-flow analysis of Java programs (along with the JDK). We evaluated the analyses against their corresponding traditional value-context versions on a multitude of benchmarks. We find that our techniques not only reduce the analysis time and memory consumption of the presented analyses significantly, but also help analyze previously unanalyzable large programs in a reasonable time. To the best of our knowledge, this is the first work that scales these heap analyses while realizing the precision of unbounded call-strings, especially using the practical value-contexts approach. Further, the proposed techniques are general enough to scale other context-sensitive heap analyses, even for programs written in other OO languages.

Contributions:

- We present the novel idea of level-summarized relevant value-contexts (*LSRV-contexts*), which take into account an important observation that we do not need to compare the complete value-contexts while performing top-down context-sensitive heap analyses. This also helps classify more value-contexts as equivalent.
- We devise a lightweight pre-analysis stage that gathers insights about the impact of a method on the callers' heap. The results of the pre-analysis are used to (i) further reduce the comparison of

contexts in the "main-analysis" stage, and (ii) defer the analysis of "caller-ignorable" methods.

• We analyze the caller-ignorable methods in a "post-analysis" stage, again in a context-sensitive manner (that is, without loss of precision). Our three-stage approach achieves the precision of a fully context-sensitive analysis for the whole program (including the JDK).

• We present an elaborate evaluation of the proposed approach on two nontrivial heap analyses (escape analysis and control-flow analysis), for a range of DaCapo and JGF benchmarks. We find that we (i) succeed in analyzing previously unanalyzable benchmarks in less than 40 minutes; and (ii) significantly reduce the memory requirements.

• We also compare our control-flow analysis implementation with object-sensitive analyses for two popular clients. We find that our approach leads to comparable precision, while consuming much less time and memory.

2 BACKGROUND

We now give a brief description of thread-escape analysis, control-flow analysis, and value-contexts based approach of performing heap analyses.

1. Thread-escape analysis [2], hereafter called escape analysis, identifies objects that can be accessed by other threads (that is, escape). An object may escape to other threads if it is reachable (possibly via a sequence of field dereferences) from a static (global) variable, or from a thread object. Escape analysis has many applications, such as synchronization elimination [2, 4, 22], data-race detection [5], efficient garbage-collection [7], and so on. A common way to perform escape analysis is to propagate the *escape-status* from the nodes reachable from global variables (and thread objects) in the points-to graph. The lattice of dataflow values for escape analysis has two elements: *DoesNotEscape* (*D*) and *Escapes* (*E*). The meet ($\sqcap$) operation is defined as: $D \sqcap D = D$, and $D \sqcap E = E \sqcap D = E \sqcap E = E$.

2. Interprocedural control-flow analysis [21, 25], used to determine the targets of a method call in dynamically-dispatched languages, is a prerequisite to, and controls the precision of, several interprocedural analyses (for example, call-graph construction [8], points-to analysis, escape analysis, and so on). A common way to perform control-flow analysis is by maintaining a points-to graph and using a dataflow lattice whose elements (indicating the possible type of each object) are the set of all the classes in the input program; the meet operation is simply the set-union operation.

3. Value-contexts, proposed by Khedker and Karkare [10], were used by Padhye and Khedker [20] to perform top-down context-, flow-, and field-sensitive points-to analysis for constructing a call-graph. The analysis starts from the main method, and maintains a points-to graph [34] at each statement. On reaching a call-statement for a method *m*, the method is (re-)analyzed, if the current value-context is different from the prior value-contexts (if any) in which *m* was analyzed. Here, the value-context is the points-to (sub) graph passed to *m* – referred to as the *parameter-reachable graph* of *m*. For example, for the code shown in Figure 1a, say Figures 1b and 1c represent the points-to graphs at lines 4 and 5, respectively. The corresponding value-contexts are shown in Figures 1d and 1e.

3 SCALABLE CONTEXT-SENSITIVE ANALYSES

Traditional value-contexts use dataflow values to restrict the combinatorial explosion of contexts in classical call-string based context-sensitive analysis. However, they do not scale well, both in terms of analysis time and memory usage, for performing common top-down context-sensitive heap-based analyses (such as escape analysis, control-flow analysis, and so on). In this section, we first illustrate the underlying problems and then describe our scalable precise solutions.

3.1 Challenges

Problem 1: *Too much comparison.* In value-contexts based analyses, when a previously analyzed method is called from a new site, the value-context at the new call-site is compared with those at the previous call-site(s). For heap-based analyses, this involves comparing the whole parameter-reachable points-to graphs. Comparing such potentially large graphs for exact equality can be costly and may lead to significant overheads.

Insight 1. The whole of the points-to graph reachable from the parameters is usually not *relevant* for the callee. For example, for the method `bar` in Figure 1a, the relevant part of the points-to graph (value-context) at its entry, for escape analysis, consists only of the objects pointed-to by `p` and `p.f1`. Hence, the relevant value-contexts for `bar` for the calls at lines 4 and 5 (shown in Figures 1f and 1g, respectively) are much smaller than the complete value-contexts (shown in Figures 1d and 1e, respectively).

Proposal: Identify/use *relevant value-contexts* for comparison.

Problem 2: *Too many contexts.* An important challenge that any context-sensitive analysis throws up for scalability is the number of contexts created during the analysis. As the number of contexts keeps increasing during the analysis, the associated method needs to be analyzed again and again in each new context. Each re-analysis consumes time (more so, if the analysis is flow-sensitive), and the generated summary increases the memory usage of the analysis, thus leading to scalability problems while performing precise analyses for large programs. Analyzing a method in a large number of contexts also implies more context-comparisons to be performed at subsequent call-sites for the method, thus aggravating the scalability issues further. This problem is pertinent even in case of the value-contexts based approaches [10, 20], where the number of contexts is bound by the size of the lattice of the points-to graph, though they improve the scalability compared to the traditional call-string based approaches [24, 25] (where the number of contexts can be unbounded).

Insight 2a. A major reason leading to a blow-up in the number of contexts is the failure to detect the equality of two contexts. For example, for performing escape analysis, consider the two relevant value-contexts (at lines 4 and 5) for the method `bar` in Figures 1f and 1g. Even though the relevant value-contexts are different, if the objects O_i, O_j, O_k and O_l do not escape, the escape-status of the object O_8 remains the same (*DoesNotEscape*) in both the contexts. Hence, once `bar` has been analyzed for the value-context at line 4, it need not be analyzed again at line 5, as the "level-wise" summary for the objects pointed to by `p` and `p.f1` match (see Figure 1h).

Proposal: Use *level-summarized relevant value-contexts.*

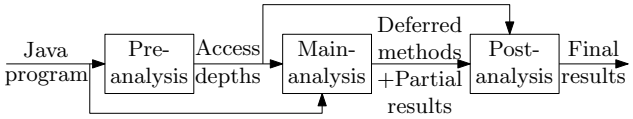


Figure 2: Block diagram of the proposed approach.

Insight 2b. Not all the methods of a program modify the heap of its callers (for example, the method `fb` in Figure 1a). The analysis of such methods does not affect the heap of their callers. When we encounter a call to such methods (*caller-ignorable*), we can defer the analysis of that method and simply proceed to the next statement. These deferred methods can be analyzed in a *post-analysis* pass after the main-analysis is over, without losing any precision. The advantage of such deferring is that we can save the time and memory spent in analyzing them (in multiple contexts), and merging their points-to graphs with those of the callers, while performing the costly main-analysis.
Proposal: Identify/defer *caller-ignorable methods*.

3.2 Proposed Approach

We now use the insights discussed above and describe our proposed three-stage approach (Figure 2 shows the block diagram) to scale value-contexts based top-down context-sensitive heap analyses. The three proposed stages are:

1. A fast flow-insensitive interprocedural *pre-analysis* that estimates, for each method m, the maximum depth of the parameter-reachable points-to graph till which the effects of m may be visible in its caller.

2. A flow- and context-sensitive *main-analysis* that takes advantage of the information gathered by the pre-analysis to (i) reduce the amount of comparison performed while checking value-contexts for equality (using the notion of relevant value-contexts), and (ii) identify and defer the analysis of caller-ignorable methods (methods with zero parameter-depth for all the parameters). Further, for comparing the relevant value-contexts for equality, we propose a novel abstraction called level-summarization that leads to fewer and more compact relevant value-contexts.

3. A flow- and context-sensitive *post-analysis* that analyzes the deferred methods, without compromising on precision.

3.2.1 Pre-analysis. For each parameter p_i of a method m, the goal of the pre-analysis is to conservatively approximate its *access-depth*, which is the maximum depth of the caller-allocated portion of the points-to graph reachable from p_i that is accessed in m. We use $\texttt{mInfo}_m(i)$ to give the access-depth of p_i in m. Intuitively, $\texttt{mInfo}_m(i)$ is k, if there exists a maximal list of k fields such that either a caller-allocated object O_x pointed-to by $p_i.f_1.f_2...f_k$ is read in m, or $p_i.f_1.f_2...f_{k-1}$ points-to a caller-allocated object and m stores an object to $p_i.f_1.f_2...f_k$. We obtain this information by performing a lightweight bottom-up analysis on the call-graph. We assume the program to be in 3-address code [18]. For ease of presentation, we list the names used in this section in Figure 3.

The pre-analysis maintains a flow-insensitive points-to graph for each method m. The analysis begins by making each of the non-primitive parameters of m point to a dummy object. The fact that these dummy objects are created in one of the callers of m is noted by setting a special boolean field `callerNode` that is associated with the corresponding object node. We also maintain a field `accDpth` with each object, whose initial value is set to zero. For brevity, we skip the standard rules to update the points-to graphs, and show the extra processing required to obtain the depth information on processing load, return, method-exit and method-call statements, using the *proc*Pre* methods (Figure 4) described below. Note that copy (of the form a=b), store (of the form $a.f$=b) and other statements do not impact the access-depth.

Name	Meaning
G_m	Points-to map for method m.
$\texttt{ret}_m$	Set of objects returned by method m.
$\texttt{mInfo}_m$	Pre-analysis summary of method m.
`deferredMethods`	Map from deferred methods to context.
O_x.`accDpth`	Access-depth of object O_x.
O_x.`callerNode`	Indicates if O_x is passed from the caller.

Figure 3: Some names used in the proposed approach.

procLoadPre: At a *load* statement L: a=$b.f$, if b is a parameter, and say the object pointed-to by b is O_b, then the points-to set of $O_b.f$ would be empty (as we started with dummy nodes for the parameters). To handle such loads, we add to the points-to graph G_m a node O_L representing the object(s) obtained from the dereference at statement L, and add O_L to the points-to set of $O_b.f$. As the object O_L actually flows from the caller of the current method, we set the field O_L.`callerNode` to *true*. If the `accDpth` of O_b is zero, we set it to one to indicate a dereference. Next, we update the value of O_L.`accDpth` based on whether or not L is inside a loop: if not, we set O_L.`accDpth` using O_b.`accDpth`; else, we update O_L.`accDpth` to ∞ (to conservatively imply that we do not know how much access-depth does O_L.`accDpth` represent).

procRetPre: At a *return* statement $return\ r$, if r is a non-primitive variable, we add the pointees of r to a set $\texttt{ret}_m$ that contains the objects returned by the method m.

procCallPre: At a *call* statement L: r=$m'(b_1, ..., b_k)$, for each object O_i pointed-to by each b_i, we update its `accDpth` using the access-depth information of the corresponding parameter of the method m' (using the summary $\texttt{mInfo}_{m'}(i)$; line 15). Next, we add a new object O_L representing the objects returned by m', and set O_L.`accDpth` (line 19) to the maximum `accDpth` of the objects returned by m'. If the information about m' is not available (possible in the context of recursion), we conservatively set the `accDpth` fields of O_i and O_L to ∞ (to keep the analysis light; lines 16 and 20).

procExitPre: Once all the statements of a method m have been processed, we invoke the function *procExitPre*, which stores in $\texttt{mInfo}_m(i)$ the access-depth of the parameter p_i. For each object pointed-to by each parameter p_i, *procExitPre* invokes the recursive function *findLvl* (line 23), which returns the maximum access-depth of the argument object, by performing a depth-first search; the pseudo-code is not shown for brevity. For any object O_x that may be obtained from the caller (identified using O_x.`callerNode`), its access-depth is computed as the maximum of O_x.`accDpth` and 1 + the maximum access-depth of its children nodes.

If an object in the points-to graph is reachable from a static (global) field, then that object might be accessed by a parallel thread.

```
1  Function procLoadPre(st:load-stmt) // st is "L: a=b.f"
2      foreach O_b ∈ G_m(b) do
3          if G_m(O_b.f) ≠ ∅ then continue;
4          G_m(O_b.f) = O_L;
5          O_L.callerNode = true;
6          O_b.accDpth = max(O_b.accDpth, 1);
7          if L is inside a loop then O_L.accDpth = ∞;
8          else O_L.accDpth = max(O_L.accDpth, O_b.accDpth);
9  Function procRetPre(st:return-stmt) // st is "return r"
10     ret_m.add(G_m(r));
11 Function procCallPre(st:call-stmt) //st is "L: r=m'(b_1, ..., b_k)"
12     foreach b_i ∈ b_1, ..., b_k do
13         foreach O_i ∈ G_m(b_i) do
14             if mInfo_m' has been populated then
15                 O_i.accDpth = max(O_i.accDpth, mInfo_m'(i));
16             else O_i.accDpth = ∞; // possible due to recursion.
17     G_m(r) = O_L;
18     if mInfo_m' has been populated then
19         O_L.accDpth = max(∀O_r ∈ ret_m' O_r.accDpth);
20     else O_L.accDpth = ∞; // possible due to recursion.
21 Function procExitPre()
22     foreach parameter p_i (at index i) of m do
23         mInfo_m(i) = max(∀O_x ∈ G_m(p_i) findLvl(O_x));
```

Figure 4: Obtaining access-depths in the pre-analysis.

We handle this case (not shown in Figure 4) by setting the value of the accDpth field of all such objects to ∞, before performing the depth-first search.

Example. For Figure 1a, as the method bar does not access parameter# 0 (this) and stores to p.f1.f2 (p is parameter# 1), $\texttt{mInfo}_{\texttt{bar}}$ would be $\{\langle 0,0\rangle, \langle 1,2\rangle\}$. Similarly, as the method fb takes no explicit arguments and does not affect the heap of its callers, $\texttt{mInfo}_{\texttt{fb}}$ would be $\{\langle 0,0\rangle\}$.

After pre-analyzing all the methods of the input program, the computed mInfo is made available to the main-analysis. We next describe how we scale the main value-contexts based analysis using the information in mInfo.

3.2.2 Main-analysis. We now describe our changes to existing points-to graph based context-, flow-, and field-sensitive analyses that use value-contexts (in terms of the points-to graphs reachable from actual arguments at call-sites). We assume that the underlying analyses maintain flow-sensitivity in the standard way, that is, as "in" and "out" points-to graphs, before and after each statement, respectively. We now highlight how we scale such context-sensitive analyses (using the results of the pre-analysis), by focusing on the method-call statements. The handling of all other statements is standard [4, 34], and skipped for brevity.

The function *procCallMain* in Figure 5 shows the handling of a call-statement st, with H_{in} as the points-to graph, in the main-analysis stage. For a method m, the entry points-to graph H_e is a copy of the points-to graph reachable from the formal parameters of m, obtained by invoking *getEntryPTG* (code not shown). If the access-depth for every parameter of m is zero (based on $\texttt{mInfo}_m$ computed in the pre-analysis), we say that m is *caller-ignorable*, that is, m does not affect the heap of its caller(s). If so, we defer the analysis of m at st, and use H_{in} as the points-to graph after st (stored in $H_{out'}$; line 4). We use the map deferredMethods to store the fact that the analysis of m was deferred in the value-context H_e.

If m is not caller-ignorable, we invoke the function *getSummary*. For each context c in which m was previously analyzed, we compare H_e with the entry points-to graph H_c at c. To avoid the overheads of comparing the whole of H_e and H_c, we use two important optimizations: (i) For each considered parameter, we compare the value-contexts only till the access-depth for that parameter (obtained from $\texttt{mInfo}_m$, populated in the pre-analysis) – *relevant value-contexts*. (ii) We use an efficient analysis-specific technique, which we call *level-summarization*, to summarize the contexts level-wise, and compare the level-summarized relevant value-contexts (*LSRV-contexts*) level by level. We describe the details of level-summarization for two analyses, thread-escape analysis and control-flow analysis, in Section 4. These two optimizations compact the points-to graphs representing the heaps being compared into smaller subgraphs, and importantly the comparison of just those subgraphs is sufficient to conclude about the equality of the heaps under consideration.

Say the LSRV-contexts for H_e and H_c are curSumm and oldSumm, respectively. We conclude that the current context does not match c (line 20) and analyze m afresh in the context H_e, if curSumm and oldSumm do not match, or if the recursive comparison of the next levels of the LSRV-contexts (done by calling the function *eqDown*) fails; see line 19.

The function *eqDown* performs a level-wise comparison for each field of the objects being compared. The function is recursive; and it returns *true* (implying that the level-wise comparison for both the graphs is equal) under one of the following conditions: (i) the maximum relevance level (that is, the access-depth) for the current parameter (as computed by the pre-analysis) is less than the current level (line 24); or (ii) the objects at the current level for both the graphs have already been compared (line 25) – a situation possible in points-to graphs with cycles. The function *eqDown* returns *false* if at any level it finds that the level-summary for the two points-to graphs does not match (line 32).

If the value of the variable matched remains *true*, that is, the entry points-to graph H_e for the method m at statement st matches a previous value-context (say prevContext), then we need not analyze m at st. In such a case, we can simply use the points-to graph at the exit of the analysis performed in prevContext as the summary of m at st (line 21). Note that using the summary computed in prevContext is sound as prevContext is equivalent to the current context for the analysis under consideration. If H_e does not match any of the previously analyzed value-contexts, or if m was not analyzed earlier, the variable matched is *false*, and we (re-)analyze m in the new context H_e (line 22). In all the cases, after obtaining the summary $H_{out'}$, we merge $H_{out'}$ with the non-parameter reachable portions of H_{in} (line 8) to obtain the points-to graph H_{out} after st (using standard merging rules [34]).

```
1  Function procCallMain(st: call-stmt, m: method, H_in: ptg)
2      ptg H_e = getEntryPTG(H_in, st); ptg H_out';
3      if m is caller-ignorable then
4          H_out' = H_in;
5          deferredMethods.put(m, H_e);
6      else H_out' = getSummary(m, H_e);
7      putSummary(m, H_e, H_out');
8      // H_out = H_out' combined with relevant parts of H_in.
9  Function getSummary(m: method, H_e: ptg)
10     let prevContext = null; matched = false;
11     outer: foreach context c in which m was analyzed do
12         prevContext = c; matched = true;
13         let H_c be the entry-ptg at c;
14         foreach non-primitive parameter p_i of m do
15             maxLvl = mInfo_m(i);
16             if maxLvl == 0 then continue; //p_i has no impact
17             curSumm = level-summary(H_e(p_i), maxLvl);
18             oldSumm = level-summary(H_c(p_i), maxLvl);
19             if curSumm ≠ oldSumm OR
                ¬eqDown(H_e,H_c,H_e(p_i),H_c(p_i),1,maxLvl) then
20                 matched = false; break outer;
21     if matched then return exit-ptg at prevContext;
22     else return analyze(m, H_e);
23 Function eqDown(H_e: ptg, H_c: ptg, curPts: set, oldPts:
     set, curLvl: int, maxLvl: int)
24     if maxLvl < curLvl then return true;
25     if curPts and oldPts have been visited then
         return true;
26     Mark curPts and oldPts as visited;
27     foreach field f do
28         curPts = ∪_{∀O∈curPts} H_e(O.f);
29         oldPts = ∪_{∀O∈oldPts} H_c(O.f);
30         curSumm = level-summary(curPts, maxLvl);
31         oldSumm = level-summary(oldPts, maxLvl);
32         if curSumm ≠ oldSumm then return false;
33     return
       eqDown(H_e,H_c,curPts,oldPts,1+curLvl,maxLvl);
```

Figure 5: Handling call-stmts in the main-analysis. Abbreviation: *ptg*: points-to graph. The function *putSummary* stores the summary; details skipped.

Example. The method `bar` in Figure 1a does not affect any object(s) reachable from the receiver (`this`). Further, the access-depth of the parameter `p` of `bar` is 2. Thus, the relevant value-contexts for `bar`, for the calls made at lines 4 and 5, are shown in Figures 1f and 1g, respectively. Similarly, the method `fb` does not affect any object(s) reachable from the receiver, and hence, the relevant value-context for `fb` is empty. Thus, `fb` is a caller-ignorable method and need not be analyzed at any of its call-sites during the main-analysis.

3.2.3 Post-analysis. The main-analysis generates a summary for each method in the program in each value-context in which it was called, except for the deferred methods. The deferred methods do not affect the heap reachable from the parameters of their callers. However, their own summary may depend on the heap of the caller. For example, say we are performing an analysis to determine which of the dereferences in a program are guaranteed to be made on concrete (non-null) objects. Say the method m being analyzed has a statement *L: p.foobar()*, where p is a parameter of m and *foobar()* is a caller-ignorable method. The pre-analysis (Section 3.2.1) would infer that m is caller-ignorable as it does not affect the heap reachable from the parameter p, and hence for each value-context, the main-analysis would defer the analysis of m. However, whether the dereference performed at statement L is done on a concrete object or not, depends on the points-to information of the actual argument in the points-to graph of m's caller(s). Thus, in each skipped value-context, we still need to analyze m. We perform this task in the post-analysis stage by iterating over the entries in the `deferredMethods` map and invoking *getSummary*(m, H_e) for each (m, H_e) in the map, to generate the corresponding context-sensitive summary. Thus, by the end of the post-analysis, we obtain context-sensitive summaries for all the methods in the program.

There are two clear advantages of deferring the analysis of caller-ignorable methods and performing the post-analysis stage as a separate pass. First, as the deferred methods are guaranteed not to affect the caller's heap, we need not spend time in merging the heap of the callee with that of the caller, after each call statement (a potentially costly operation). Second, top-down context-sensitive heap analyses in general may consume a large amount of memory, as the points-to graphs reachable from the callers keep flowing towards the leaves of the call-graph till the points-to graphs of the callees are merged back. Deferring the analysis of certain methods may avoid the traversal of certain long branches of the call-graphs and thus save the memory required to propagate the points-to graphs therein, during the costly main-analysis.

4 INSTANTIATIONS

In this section, using the techniques proposed in Section 3, we present two popular context, flow-, and field-sensitive points-to graph based analyses: (i) thread-escape analysis; and (ii) control-flow analysis. Though both these analyses are based on points-to information, the corresponding lattices of dataflow values are quite different (see Section 2 for an overview), and hence they offer a wide illustration of our proposed techniques. For both the analyses, we mainly highlight how the level-summary functions (see Figure 5) are defined and computed. Note that apart from the function *level-summary*, the rest of the handling of a method-call statement remains the same as in Figure 5.

4.1 Thread-escape Analysis

Consider a method m that is being analyzed. Say p is one of the parameters of m, and H_e is the value-context (points-to graph at the entry of m). Say the graph-front induced by the edge sequence $f_1, f_2, ..., f_k$ in H_e is given by the set $S = \{O_1, O_2, ..., O_n\}$. For escape analysis, we claim that the relevant information represented by the objects in S is the set of escape-statuses of the

objects $O_1, O_2, ..., O_n$. For example, if a variable q is set to the object obtained by dereferencing p using an expression of the form $p.f_1.f_2...f_{k-1}$, then for a sound escape-analysis technique, the following two observations hold: (i) If none of the objects in S escapes, then any object stored into any field f_i of q will not escape. (ii) If any of the objects in S is marked as escaping, then any object stored into $q.f_i$ will also be marked as escaping. Hence for escape analysis, we define the level-summary of a set S of objects as the meet of the escape-statuses of the objects in S.

The *level-summary* function for escape analysis takes two arguments: H and k, where H is a points-to graph (value-context) and k is the access-depth (see Section 3.2.2). It computes the level-summaries for each possible edge-sequence of size at most k and then returns a graph, which includes a unique node for each unique edge sequence and if s_1 and s_2 are the nodes (level-summaries) corresponding to the edge sequences $f_1, f_2, \cdots f_{k_1}$ and $f_1, f_2, \cdots f_{k_1}, f_k$, respectively, then there is an edge between s_1 and s_2, labeled k.

Example. Consider the calls to the method `bar` in Figure 1a, with the respective relevant value-contexts as shown in Figures 1f and 1g. As none of the objects O_i, O_j, O_k and O_l escape (implying that O_a and O_b also must not escape), Figure 1h represents the LSRV-context for both Figures 1f and 1g. As a result, with LSRV-contexts, we need not analyze `bar` for the call made at line 5, and can use the analysis-results obtained for the call at line 4 itself.

4.2 Control-flow Analysis

Consider a method m that is being analyzed. Say p is one of the parameters of m, and H_e is the value-context (points-to graph at the entry of m). Say the graph-front induced by the edge sequence $f_1, f_2, ..., f_k$ in H_e is given by the set $S = \{O_1, O_2, ..., O_n\}$. For control-flow analysis, we claim that the relevant information represented by the objects in S is the set of types represented by the objects $O_1, O_2, ..., O_n$. For example, if a variable q is set to the object obtained using an expression of the form $p.f_1.f_2...f_{k-1}$, and if m consists of a statement *L: q.foo()*, then the set of possible callees at L depends only on the types of the objects in S. Hence for control-flow analysis, we define the level-summary of a set S of objects as the union of the types of the objects in S.

Example. Consider the calls to the method `bar` in Figure 1a, with the respective relevant value-contexts as shown in Figures 1f and 1g. Here, if the types of O_a and O_b are same, then the two LSRV-contexts at level 1 from the parameter p would be the same. Similarly, if the set of types for $\{O_i, O_j\}$ and $\{O_k, O_l\}$ are same, then the LSRV-contexts at level 2 would also be the same. In such a case, we need not analyze `bar` for the call made at line 5.

In Section 6, we show that LSRV-contexts not only lead to a significant reduction in the escape and control-flow analysis time and memory of several benchmarks, but also facilitate the analysis of previously unanalyzable benchmarks.

5 DISCUSSION

1. Scope. In this paper, in order to scale context-sensitive heap analyses that use value-contexts, we have presented three main ideas: relevant value-contexts, level-summarization, and deferred methods. While the second idea is analysis-specific and needs to be customized for each analysis, the other two are general in nature and can be used directly for any heap analysis.

2. Cost. It can be argued that an approach using LSRV-contexts is likely to never increase the cost compared to that using traditional value-contexts. The relevant value-contexts, as discussed in Section 3.2, are always a subset of the points-to graphs, and level-summarization is analysis-specific but usually leads to a smaller lattice of dataflow values (as shown in Section 4). For the analyses under consideration, we show in Section 6 that LSRV-contexts are much cheaper (in terms of time as well as memory) than traditional value-contexts.

3. Special treatment of methods. In the JDK library, the methods `equals` and `toString` are overridden heavily, and often call other `equals` and `toString` methods. For these methods, we found that the Spark tool [13], which we use to build our base call-graph, led to huge strongly-connected components (clique with up to 356 JDK methods), which blew up the analysis time and memory. Hence, just for these `equals` and `toString` methods, we suggest using the approach of Smaragdakis et al. [28] and analyzing them conservatively (intra-procedural). We believe it to be a reasonable design decision, as based on our analysis of the complete JDK library (version 8), these methods do not explicitly modify their callers' heap. To be consistent, we handle these methods uniformly, in all the implementations evaluated in Section 6. Note: we do analyze their implementations in the applications like any other method.

4. Correctness. As described in Section 3, in order to determine whether two value-contexts for a method are equivalent, we compare only the LSRV-contexts. Further, we defer the analysis of caller-ignorable methods, and analyze them in a separate pass. We now sketch the correctness argument of our design (proofs omitted).

Lemma 5.1. For a given value-contexts based analysis $\mathcal{A}$, say the LSRV-contexts variant be represented by $\mathcal{A}'$. During the analysis, at a certain call-site, if $\mathcal{A}$ re-analyzes a target method m, then either $\mathcal{A}'$ also re-analyzes m, or there exists a prior instance of $\mathcal{A}'$ analyzing m such that: (i) the abstract heap obtained by $\mathcal{A}$ on re-analyzing m matches the one obtained by $\mathcal{A}'$ in the prior analysis-instance; and (ii) in $\mathcal{A}$, during the re-analysis of m, for each accessed abstract heap-location, the abstract value read/written matches that of $\mathcal{A}'$ in the prior analysis-instance.

This lemma ensures that for a particular analysis under consideration, compared to the complete value-contexts, the LSRV-contexts do not lose any relevant information. Consequently, instead of comparing the complete value-contexts, it is sufficient to check the equality of the level-summaries of the corresponding relevant value-contexts.

Lemma 5.2. For each caller-ignorable (and hence deferred) method m, if $\mathcal{H}$ is the reachable abstract heap present before analyzing the call to m, and $\mathcal{H}'$ is the reachable abstract heap after analyzing the call-statement, then $\mathcal{H} = \mathcal{H}'$.

This lemma, along with the fact that the caller-ignorable methods are analyzed in a post-analysis in all the contexts in which they are called, ensures that analyzing a deferred method m in the post-analysis does not affect the precision of its callers, and m.

Lemmas 5.1 and 5.2 ensure that the precision of an analysis using our proposed approach is the same as that using traditional value-contexts.

1	2	3	4
Bench-mark	Application		#Referred JDK classes*
	#classes	size (MB)	
avrora	527	2.7	1588
batik	1038	6.0	3700
eclipse	1608	14.0	2589
luindex	199	1.3	1485
lusearch	198	1.3	1481
pmd	697	4.1	1607
sunflow	225	1.7	3509
moldyn	13	0.15	1555
montecarlo	19	0.67	1555
raytracer	19	0.21	1555

Figure 6: Details of the benchmarks used. App: Application. *Referred classes computed using Spark's [13] call graph.

1	2	3	4	5	6	7	8
Bench-mark	Analysis time (seconds)					Average contexts	Memory (GB)
	B_e	Pre	Post	PM	PMP	PMP	PMP
avrora	-	1.0	0.4	603	225	1.4	21
batik	-	2.2	1.8	3483	1722	1.4	45
eclipse	-	2.7	6.0	-	2275	1.9	57
luindex	-	1.1	0.4	204	70	1.3	6
lusearch	-	1.0	0.5	343	87	1.3	10
pmd	-	1.3	0.4	531	157	1.3	11
sunflow	-	2.1	1.6	1477	486	1.3	21
moldyn	-	0.9	0.4	178	55	1.3	6
montecarlo	-	0.9	0.4	183	57	1.3	6
raytracer	-	0.9	0.4	183	54	1.3	6
geomean	-	1.3	0.7	444	192	1.4	13

Figure 7: Evaluation results for escape analysis. Abbreviations: B_e: Base$_e$; PM: Pre and Main; PMP: Pre, Main and Post. A '-' implies that the analysis did not terminate in 3 hours.

6 IMPLEMENTATION AND EVALUATION

We have implemented both of our proposed instantiations of thread-escape analysis and control-flow analysis, in the Soot framework [32]. The implementation spans 3967 lines of Java code for the escape analysis, and 3899 lines of Java code for the control-flow analysis. We have performed our experiments using the OpenJDK HotSpot JVM (version 8), on a 2.3 GHz AMD system with 64 cores and 512 GB of memory.

We have evaluated our techniques on seven benchmarks from the DaCapo-9.12 suite [1], and the three multithreaded benchmarks from Section C of the JGF suite [6] (listed in Figure 6). We used the extremely helpful tool TamiFlex [3] to resolve reflective calls in the original DaCapo benchmarks, so that they could be analyzed by Soot. The benchmarks excluded from the DaCapo suite are the ones which either could not be translated by TamiFlex, or could not be analyzed by Soot (using OpenJDK8) after the TamiFlex pass.

Figure 6 shows some static characteristics of the used benchmarks. The sizes of the benchmarks (excluding the JDK library) varied from 150 KB (small programs) to 14 MB (large applications), and the number of application classes varied from 13 to 1.6K. Figure 6 also shows the number of JDK classes referred by each benchmark (gives the total number of analyzed classes), computed using the call-graph generated by the Spark [13] tool (our default call-graph).

We now present an evaluation to study the impact of our proposed techniques on the scalability of context-, flow- and field-sensitive escape and control-flow analyses. For both the analyses, we compare four different versions: (i) Base: a standard value-contexts based implementation, where the points-to graphs at method entries are considered as the contexts. (ii) OM: a version where only the main analysis (as proposed in Sections 3 and 4) is performed. Thus, the level-wise summaries are used as contexts, but no pre-analysis (that is, trimming) and post-analysis (that is, deferring) are performed. (iii) PM: both the pre and the main analyses are performed. Thus, the level-wise summaries are trimmed based on the access-depths computed by the pre-analysis. (iv) PMP: the full proposed version where all the three analyses (pre, main, and post) are performed. We compare these versions on three parameters: (i) analysis time, (ii) average number of created contexts, and (iii) peak memory consumption. In Section 6.4, we further compare the scalability and precision of LSRV-contexts with k-object sensitivity based approaches, for the control-flow analysis. As prior works [10, 20] have already shown that the classical call-string based approach does not scale well compared to the value-contexts based approach, we omit a comparison against the same.

6.1 Analysis Time

Figures 7 and 8 show the time taken for performing the escape analysis and the control-flow analysis, respectively, by the different versions. Base$_e$ implements the (Base) thread-escape analysis of Whaley and Rinard [34], and Base$_c$ implements the (Base) control-flow analysis of Padhye and Khedker [20]; both Base$_e$ and Base$_c$ use value-contexts based context-sensitivity. We found that Base$_c$ did not terminate within 3 hours (our set cutoff) for three large DaCapo benchmarks, and Base$_e$ did not terminate in 3 hours for any of the benchmarks. It clearly shows the scalability issues with the Base variations, which simply use the parameter-reachable points-to graph at the entry of a method as the value-context.

Columns 3-4 in Figures 7 and 8 show the times taken by the pre and the post analyses while performing the escape and the control-flow analyses, respectively. As the proposed pre-analysis is agnostic to the heap analysis being performed, it takes the same time (on average, 1.3 seconds) for both the analyses. We observe that together the pre and the post analyses take very less time – just 2.0 seconds for escape analysis, and 2.2 seconds for control-flow analysis, on average.

Column 6 in Figure 7 and column 7 in Figure 8 (labeled PMP) show the full analysis time of our proposed technique, which includes the pre, the main, and the post-analysis times. Not only is our proposed technique able to analyze all the benchmarks within the set cutoff, the average required time is just 192 seconds for escape analysis, and 130 seconds for control-flow analysis (the largest value being less than 40 minutes, for ECLIPSE). It can be seen that the time required depends mostly on the size of the benchmark. Note that though SUNFLOW appears to be a relatively small benchmark

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Benchmark	**Analysis time (seconds)**							**Average contexts**				**Memory (GB)**		
	B_c	**Pre**	**Post**	**OM**	**PM**	**PMP**	**2obj1h**	**B_c**	**OM**	**PMP**	**2obj1h**	**B_c**	**PMP**	**2obj1h**
avrora	1322	1.0	0.5	231	71	55	592	9.5	2.6	1.2	13.0	54	11	29
batik	-	2.2	2.4	-	1033	946	-	-	-	1.3	-	-	64	-
eclipse	-	2.7	6.1	-	1312	988	-	-	-	1.4	-	-	49	-
luindex	1175	1.1	0.7	507	60	46	485	10.6	2.9	1.2	14.6	58	11	23
lusearch	1215	1.0	0.9	561	58	57	611	10.5	3.6	1.2	15.0	54	11	29
pmd	5769	1.3	0.7	1243	130	108	1112	11.9	4.5	1.2	15.1	127	13	45
sunflow	-	2.1	2.2	-	692	684	-	-	-	1.2	-	-	53	-
moldyn	929	0.9	0.6	222	54	53	412	9.5	2.5	1.3	13.1	29	11	22
montecarlo	925	0.9	0.3	238	60	53	413	9.4	2.6	1.2	13.3	29	9	23
raytracer	940	0.9	0.3	211	61	53	415	9.4	2.6	1.2	13.2	29	10	23
geomean	1364	1.3	0.9	351	151	130	542	10.1	3.0	1.2	13.9	47	18	27

Figure 8: Evaluation results for control-flow analysis. Abbreviations: B_c: Base$_c$; OM: Only Main; PM: Pre and Main; PMP: Pre, Main and Post. A '-' implies that the analysis did not terminate in 3 hours; the geomean was computed by excluding non-terminating benchmarks.

(see column 3, Figure 6), the corresponding analysis time is high, as SUNFLOW references a large number of the JDK library classes (see column 4, Figure 6).

In order to individually estimate the effects of the pre and the post analyses (that is, the insights presented in Section 3.1) on the scalability, we studied the analysis times of the OM and the PM versions (see column 5 in Figure 7 and columns 5-6 in Figure 8). For escape analysis, the OM version did not terminate in 3 hours for any benchmark (hence not shown), while for control-flow analysis, it did not terminate for three large DaCapo benchmarks (BATIK, ECLIPSE and SUNFLOW). This indicates that only level-wise summarization (done in OM version, *Insight 2a*) is not sufficient to scale all the analyses under consideration. We can see that though the time savings due to the pre-analysis alone (PM version, *Insights 1+2a*) are significant (except for the escape analysis of ECLIPSE, where it did not terminate), the post-analysis improves it further. The PMP version (*Insights 1+2a+2b*) runs faster than the PM version by about 56% and 14% for the escape and the control-flow analyses, respectively. Thus, by spending just ~2 seconds for the pre and the post analyses over the OM version, the PMP version successfully scales both the analyses.

Overall, we see that the combination of the pre, the main, and the post analyses helps us perform previously non-terminating analyses in less than 40 minutes, for all the benchmarks under consideration.

6.2 Number of Contexts

Columns 7 in Figure 7 and 9-11 in Figure 8 show the average number of contexts created during escape analysis and control-flow analysis, respectively, over all the methods. The numbers for the PM version are not shown separately; they match the ones for the PMP version, as the deferred methods in PMP are later analyzed in all the deferred contexts. For escape analysis, on average, PMP creates 1.4 contexts per method (Base$_e$ and OM did not terminate, and hence average contexts not reported). For control-flow analysis, for the cases where Base$_c$ terminated, we can see that the average number of contexts created per method is about 10.1. On the other hand, the number is around 3.0 for the OM version (for control-flow analysis), and just 1.2 for the PMP version. This implies that our proposed techniques significantly reduce the number of times a method is analyzed (~8×, on average).

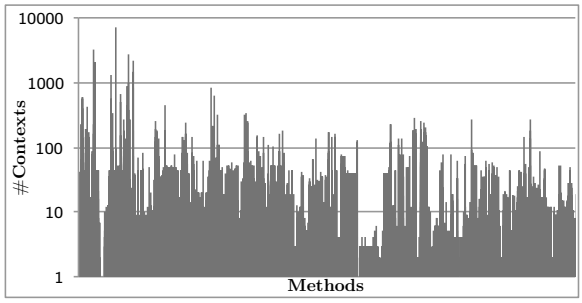
To further visualize the difference in the number of contexts created, we have plotted two histograms for the benchmark PMD for control-flow analysis; see Figure 9. The charts show the number of contexts created per method in Base$_c$ (Figure 9a) and PMP (Figure 9b) versions, respectively. The methods (in the x-axis) are arranged in alphabetical order, and the number of contexts (y-axis) is shown growing logarithmically. The high density of the bars for Base$_c$ clearly shows the large number of contexts created for a large number of methods. For instance, the maximum number of contexts created in the Base$_c$ version was 7324 for the method *java.lang.Object: void <init>*, for which there was only one context created in the PMP version. We observed a similar trend for all the benchmarks.

Overall, we see that our proposed techniques were able to significantly reduce the number of contexts created per method, which in turn led to a significant reduction in the resources spent in analyzing those methods, thus making the analyses scalable.

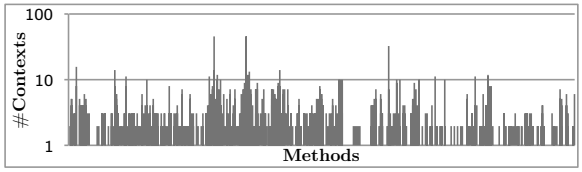
6.3 Peak Memory Usage

Figure 7 and 8 also show the peak memory consumption of the Base and the PMP versions for the escape and the control-flow analyses, respectively. As Base$_e$ did not terminate within the cutoff of 3 hours, the corresponding statistic is not shown. However, as a point of indication, the usage at the end of 3 hours for the smallest benchmark MOLDYN was about 373 GB, which indicates that for larger benchmarks the analysis may run out of memory, if run for a longer time. The high memory usage is due to the large number of contexts created during the analysis and the resultant flow-sensitive points-to graphs maintained therein.

In comparison, the memory usage in the PMP version was several magnitudes lesser, and was just 13 GB and 18 GB respectively, on

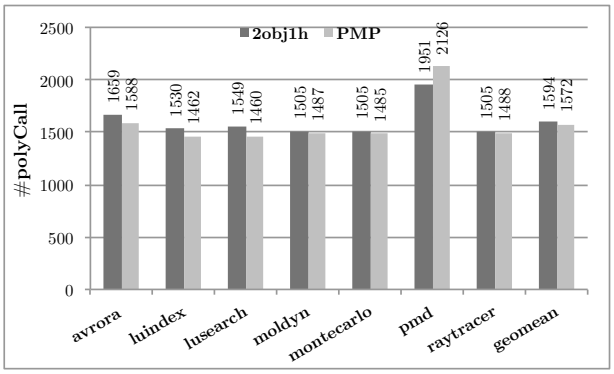


(a) Base$_c$ version.

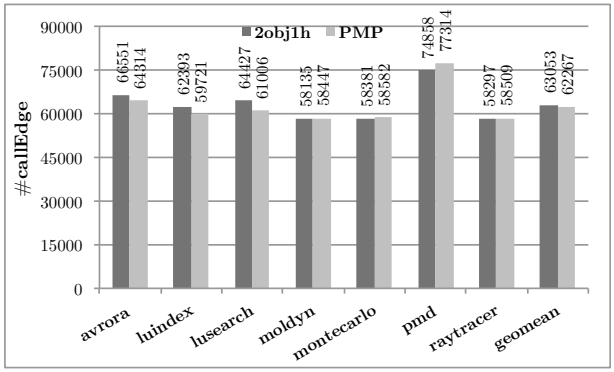


(b) PMP version of LSRV-contexts.

Figure 9: Number of contexts created per method during the control-flow analysis of the benchmark PMD.



(a) Number of polymorphic calls.



(b) Number of edges in the call-graph.

Figure 10: Precision of LSRV-contexts versus 2obj1h (lower values are better).

average, for the two analyses. We argue that given the range and size of the benchmarks under consideration, this is quite reasonable. We also observe that the memory usage varied mostly with the size of the benchmark: lower for the JGF benchmarks (smaller) and higher for BATIK, ECLIPSE and SUNFLOW (larger), for both escape and control-flow analysis.

Overall, we note that our proposed techniques allow performing precise whole-program heap analyses, which earlier did not scale even with a large amount of memory (~512 GB), now on systems with much less memory (~32-64 GB).

6.4 Comparison with Object-sensitive Analyses

It is well understood [15, 17] that as the contexts created in the call-site-sensitive and object-sensitive approaches are different, these approaches are in-principle incomparable. However, there have been works that approximate a comparison by averaging some precision-indicating metrics (for example, the number of calls that could not be resolved to a single method) over all the contexts created in each approach [15]. We next use a similar approach to compare LSRV-contexts with the state-of-the-art two-level object-sensitive control-flow analysis, with one level of heap-cloning (abbreviated as 2obj1h). Our implementation of 2obj1h (in Soot) is based on *full-object-sensitivity* as defined by Smaragdakis et al. [27].

For the comparison, we use two popular [15, 27] clients: (i) *#polyCall*: the number of call-sites that could not be resolved to a single method (Figure 10a); and (ii) *#callEdge*: the number of edges in the on-the-fly call-graph (Figure 10b). For the benchmarks BATIK, ECLIPSE and SUNFLOW, 2obj1h did not terminate in 3 hours (hence not shown in Figure 10). We can see that LSRV-contexts were able to resolve up to 1.38% more call-sites as "monomorphic" over 2obj1h, on average. That is, the methods called at these many call-sites can be inlined and/or statically linked, possibly enabling several interprocedural optimizations. Similarly, the number of edges in the on-the-fly call-graph created while using LSRV-contexts was 1.25% lesser than using 2obj1h, across the benchmarks on which 2obj1h terminated. Note that the trends are not uniform over the benchmarks, which further elucidates the theoretical incomparability of the precision of the call-site- and the object-sensitive approaches.

Columns 8, 12 and 15 in Figure 8 respectively show the analysis time, the average number of contexts created, and the peak memory usage of 2obj1h, per benchmark. As observed by prior works [16, 27], we too observe that 2obj1h does not scale to large benchmarks (BATIK, ECLIPSE and SUNFLOW, in our case). For the rest of the benchmarks, we can see that the analysis-cost of LSRV-contexts is much less compared to 2obj1h: 89.2% lesser time and 59.4% lesser memory. Akin to the improvement over traditional value-contexts, we can correlate the scalability of LSRV-contexts over 2obj1h by observing a significant dip in the average number of contexts created (and analyzed) per method: 13.9 for 2obj1h, but only 1.2 for LSRV-contexts. For completeness, we also compared the scalability of LSRV-contexts with a less precise but faster one-level object-sensitive (without heap-cloning) control-flow analysis (say 1obj). We found that though 1obj terminated for all the benchmarks, it still took 71.5% more time than LSRV-contexts (detailed statistics skipped).

Overall, we see that compared to the state-of-the-art object-sensitive analyses, LSRV-contexts scale much better, and generate more opportunities, for the considered clients. To the best of our knowledge, this is the first time that the state-of-the-art object- and call-site-sensitive analyses could even be compared for large benchmarks (made possible by the scalability added to the latter by LSRV-contexts).

7 RELATED WORK

There have been several works that propose ways to perform context-sensitive analyses in a scalable manner; we can broadly classify them into two categories: (i) efficiently maintaining the identified contexts [31, 33, 35]; and (ii) defining new context abstractions [10, 17, 27, 30].

Whaley and Lam [33] proposed the use of binary decision diagrams (BDDs) to represent equivalent contexts with lesser memory. Xu et al. [35] explore the non-scalability of BDDs for context-sensitive analyses involving heap-cloning, and merge the calling contexts with similar points-to relationships. However, their presented analysis is imprecise as it lacks flow-sensitivity. Thiessen and Lhoták [31] present a novel way to scale k-length call-string based analyses by representing points-to information in terms of the input-output values of different contexts, which allows merging of equivalent call-strings. In contrast, in this paper, we have proposed ways to scale the value-contexts based approach by compacting (level-summarizing) and improving the precision of the process of finding equivalent contexts.

Milanova et al. [17] propose a new context abstraction called k-object-sensitivity, which distinguishes the contexts based on the allocation site of the receiver. Smaragdakis et al. [27] clarify the definition of object-sensitivity for $k > 1$, and propose type-sensitivity as another close sibling. In recent attempts to scale these abstractions, Tan et al. [30] merge type-consistent objects for type-dependent analyses, and Li et al. [16] select among the different variants of object- and type-sensitivity for each method (with a small dip in the precision). On the contrary, we proposed LSRV-contexts: a way to scale call-site-sensitivity based analyses (while maintaining their precision), which, though incomparable, generate similar number of optimization opportunities as object-sensitive analyses. Similar to prior works [17, 27], we plan to use heap-cloning to further improve the precision of client-specific call-site-sensitive analyses (while keeping scalability), in future.

There have been works that scale the main-analysis using a pre-analysis. Oh et al. [19] perform a pre-analysis that estimates the impact of context-sensitivity on different methods for a given set of queries (for C programs), and then reduce the precision of the main-analysis on methods that might not benefit from the enhanced precision. Tan et al. [29] use a pre-analysis to identify and eliminate redundant objects from object-sensitive analyses (for Java programs) to reduce the number of effective contexts. Recently, Karkare [9] first uses a fast analysis to mark variables whose shape cannot be refined, and skips them in a following precise (slow) pass. Prior works [26, 28] use a pre-analysis to identify code portions that do not affect the analysis results or may degrade scalability, and analyze them conservatively. In contrast, our proposed pre-analysis identifies the relevant portions of the caller's heap, whose results are then used to scale whole-program context-sensitive analyses for Java programs. We also use the pre-analysis to identify caller-ignorable methods that are deferred by the main-analysis and analyzed as a post-pass without any loss of precision.

Whole-program escape analysis and control-flow analysis are two very important heap analyses with wide applicability, and there have been works to improve their scalability on large programs. Kotzmann and Mössenböck [11] propose a fast but imprecise unification-based escape analysis for the HotSpot client compiler [12]. Padhye and Khedker [20] present a value-contexts based precise control-flow analysis; however it does not scale well on large programs. Our proposed approach helps perform value-contexts based escape and control-flow analyses that are not only precise, but also scale very well to large programs. To the best of our knowledge, this is the first work that scales these heap analyses while realizing the precision of unbounded call-strings, especially using the practical value-contexts approach.

8 CONCLUSION AND FUTURE WORK

In this paper, we proposed a three-stage analysis approach to scale complex whole-program value-contexts based heap analyses for large programs, without losing precision. Our approach was based on the novel idea of LSRV-contexts, which take into account an important observation that we do not need to compare the complete value-contexts at each call-site. LSRV-contexts helped reduce the comparison performed for determining context-equality and classify more value-contexts as equivalent. We evaluated our approach on two nontrivial heap analyses. The results showed that our approach not only reduced the analysis time and memory consumption significantly, but also helped analyze previously unanalyzable large programs in a reasonable time.

Future work. Besides using LSRV-contexts to scale other analyses, we also plan to incorporate the benefits of object-sensitivity and heap-cloning to LSRV-contexts, thereby improving the precision of value-contexts further.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their suggestions for improvements. We also thank Rohan Padhye (PhD student, UC Berkeley) and Uday Khedker (Professor, IIT Bombay) for insightful discussions on value-contexts.

REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190.

[2] Bruno Blanchet. 2003. Escape Analysis for JavaTM: Theory and Practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. https://doi.org/10.1145/945885.945886

[3] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. https://github.com/secure-software-engineering/tamiflex. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

[4] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *Proceedings of the 14th ACM*

SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99). ACM, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[5] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

[6] Charles Daly, Jane Horgan, James Power, and John Waldron. 2001. Platform Independent Dynamic Java Virtual Machine Analysis: The Java Grande Forum Benchmark Suite. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)*. ACM, New York, NY, USA, 106–115. https://doi.org/10.1145/376656.376826

[7] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local Heaps for Java. In *Proceedings of the 3rd International Symposium on Memory Management (ISMM '02)*. ACM, New York, NY, USA, 76–87. https://doi.org/10.1145/512429.512439

[8] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (Nov. 2001), 685–746. https://doi.org/10.1145/506315.506316

[9] Amey Karkare. 2018. TwAS: Two-stage Shape Analysis for Speed and Precision. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. ACM, New York, NY, USA, 1857–1864. https://doi.org/10.1145/3167132.3167330

[10] Uday P. Khedker and Bageshri Karkare. 2008. Efficiency, Precision, Simplicity, and Generality in Interprocedural Data Flow Analysis: Resurrecting the Classical Call Strings Method. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 213–228. http://dl.acm.org/citation.cfm?id=1788374.1788394

[11] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1064979.1064996

[12] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[13] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169. http://dl.acm.org/citation.cfm?id=1765931.1765948

[14] Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. https://doi.org/10.1007/11688839_5

[15] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987

[16] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first Pointer Analysis with Self-tuning Context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[17] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

[18] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[19] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

[20] 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 (SOAP '13)*. ACM, New York, NY, USA, 31–36. https://doi.org/10.1145/2487568.2487569

[21] Jens Palsberg and Michael I. Schwartzbach. 1991. Object-oriented Type Inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. ACM, New York, NY, USA, 146–161. https://doi.org/10.1145/117954.117965

[22] Erik Ruf. 2000. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 208–218. https://doi.org/10.1145/349299.349327

[23] Marc Shapiro and Susan Horwitz. 1997. The effects of the precision of pointer analysis. In *Static Analysis*, Pascal Van Hentenryck (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–34.

[24] M Sharir and A Pnueli. 1978. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY. https://cds.cern.ch/record/120118

[25] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[26] 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)*. ACM, New York, NY, USA, 253–270. https://doi.org/10.1145/2509136.2509524

[27] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

[28] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

[29] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis*, Xavier Rival (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510.

[30] 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)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

[31] Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 263–277. https://doi.org/10.1145/3062341.3062359

[32] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–23. http://dl.acm.org/citation.cfm?id=781995.782008

[33] 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 (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[34] John Whaley and Martin Rinard. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*. ACM, New York, NY, USA, 187–206.

[35] Guoqing Xu and Atanas Rountev. 2008. Merging Equivalent Contexts for Scalable Heap-cloning-based Context-sensitive Points-to Analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/1390630.1390658