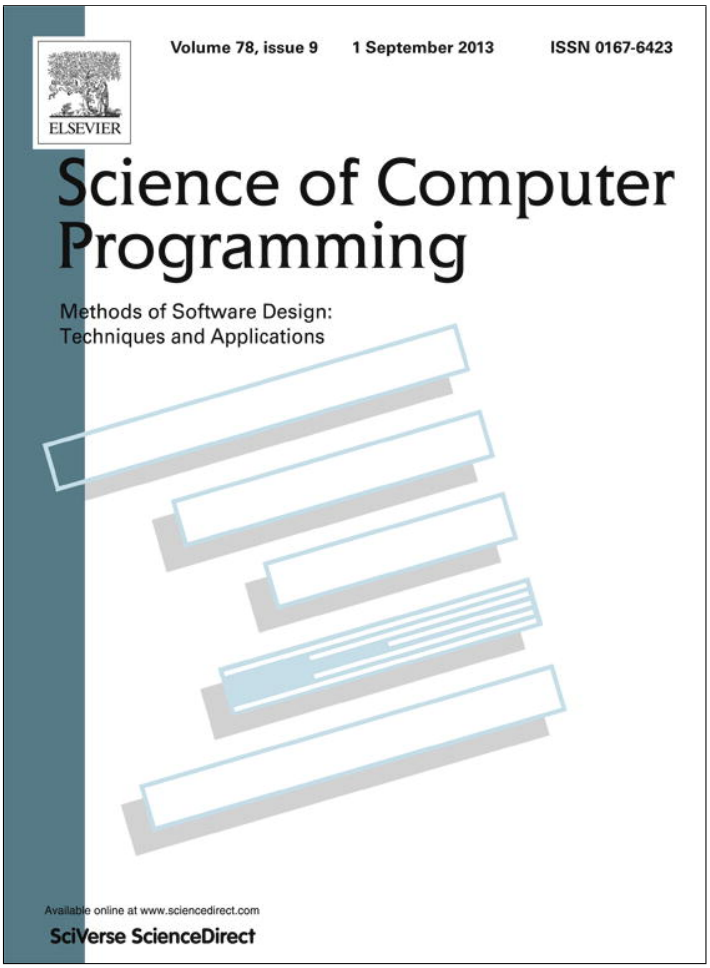


Science of Computer Programming 78 (2013) 1701–1727

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Access-based abstract memory localization in static analysis☆

Hakjoo Oh*, Kwangkeun Yi

Seoul National University, Republic of Korea

HIGHLIGHTS

- The conventional reachability-based localization is too conservative in practice.
- We propose access-based localization technique and show that it is much more effective than the reachability-based localization.
- Access-based localization is effectively realizable by employing a conservative pre-analysis.
- Access-based localization is effectively applicable to arbitrary code blocks rather than procedures.

ARTICLE INFO

Article history:
Received 4 August 2011
Received in revised form 14 March 2013
Accepted 15 April 2013
Available online 29 April 2013

Keywords:
Static analysis
Abstract interpretation
Localization

ABSTRACT

On-the-fly localization of abstract memory states is vital for economical abstract interpretation of imperative programs. Such localization is sometimes called "abstract garbage collection" or "framing". In this article we present a new memory localization technique that is more effective than the conventional reachability-based approach. Our technique is based on a key observation that collecting the reachable memory parts is too conservative and the accessed parts are usually tiny subsets of the reachable part. Our technique first estimates, by an efficient pre-analysis, which parts of input states will be accessed during the analysis of each code block. Then the main analysis uses the access-set results to trim the memory entries before analyzing code blocks. In experiments with an industrial-strength global C static analyzer, the technique is applied right before analyzing each procedure's body and reduces the average analysis time and memory by 92.1% and 71.2%, respectively, without sacrificing the analysis precision.

In addition, we present three extensions of access-based localization: (1) we generalize the idea and apply the localization more frequently such as at loop bodies and basic blocks as well as procedure bodies, additionally reducing analysis time by an average of 31.8%; (2) we present a technique to mitigate a performance problem of localization in handling recursive procedures, and show that this extension improves the average analysis time by 42%; (3) we show how to incorporate the access-based localization into relational numeric analyses.

1. Introduction

In global abstract interpretation of imperative programs, memory localization (sometimes called "abstract garbage collection" or "framing") is vital for reducing analysis cost [8,15,22,24,37] (we confine our claim to a family of static analyses described in Section 2). Localization, when analyzing a code block, attempts to remove the irrelevant parts of input memory

☆ This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/National Research Foundation of Korea (NRF) (Grant 2012-0000468) and Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2012.

* Corresponding author. Tel.: +82 1091719547.
E-mail addresses: pronto@ropas.snu.ac.kr, hakjoo.oh@gmail.com (H. Oh).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.04.002

Table 1
Reachability-based approach is too conservative.

Program	LOC	accessed abstract memory / reachable abstract memory			
spell-1.0	2,213	5	/	453	(1.1%)
httptunnel-3.3	6,174	10	/	673	(1.5%)
gzip-1.2.4a	7,327	22	/	1002	(2.2%)
jwhois-3.0.1	9,344	28	/	830	(3.4%)
parser	10,900	75	/	1787	(4.2%)
bc-1.06	13,093	24	/	824	(2.9%)
less-290	18,449	86	/	1546	(5.6%)

states, which will not be used during the analysis of the block. Not to mention the immediate benefit of the reduced memory footprint, localization has another important impact on cost reduction. In flow-sensitive global abstract interpretation, code blocks such as procedure bodies are repeatedly analyzed (often needlessly) with different input memory states. Localization makes input memory states smaller, which results in more general summaries for the blocks. More general summaries reduce re-analysis of blocks by increasing the chance of reusing the previously computed analysis results. For example, consider a code `x=0;f();x=1;f();` and assume that `x` is not used inside `f`. Without localization, `f` is analyzed twice because the input state to `f` is changed at the second call. If `x` is removed from the input state (localization), the analysis result of `f` for the first call can be reused for the second call without re-analyzing the procedure.

Realization of effective localization is not trivial. Suppose we analyze a procedure with an input memory state. Localization tries to remove, from the input state, all the parts that will not be used during the analysis of the procedure. The problem is that it is not always possible to know in advance the to-be-used parts of the input state unless we actually analyze the procedure. For example, we cannot exactly determine the access information for indirect reference `*p` before starting the analysis. Hence, any localization technique estimates the usable memory parts conservatively: the localized state is guaranteed to contain all the parts that will be used but may contain spurious entities (that will not actually be used) as well.

The conventional estimation methods are reachability-based [8,36,37,15,32,31,22,24]. They collect memory parts that are reachable (by pointer chains) from the current environment, and the memory parts that can no longer be reached (hence, obviously cannot be used anymore) are removed from the current states. When applied to procedure bodies, the technique allows a procedure to be analyzed with only memory portions that are reachable from actual parameters or global variables. Because the reachable is often smaller than the entire state, the method is popular in various kinds of program analysis: for example, in shape analysis [36,15,32,22,9] and higher-order flow analysis [24,8]. In this article, we assume the sorts of analyses with allocation-site heap abstraction, where all the memory locations allocated at an allocation site are collectively represented by a single abstract location.

However, reachability is just a crude approximation and sometimes too conservative in practice. This is mainly because large parts of the reachable portion of input states are not actually accessed, i.e. the values are neither read nor written during the analysis. For example, Table 1 shows, given a reachability-based localized input state to a procedure, how much is actually accessed inside the (directly or transitively) called procedures. For each a/b (r%), a is the average number of abstract locations accessed in the called procedures, b is the average size of the reachable input state, and r is their ratio.[1] The results show that only few reachable memory entries were actually accessed: procedures accessed only 1.1%–5.6% of reachable memory states. Nonetheless, the reachability-based approach propagates all the reachable parts to procedures. It is therefore possible for a procedure body to be needlessly recomputed for input memory states whose only differences lie in the reachable-but-non-accessed portions. This means that the reachability-based approach can be too conservative for real C programs and hence is inefficient in both time and memory cost. This observation was made while investigating the reasons for the inefficiency of an industrial-strength static analyzer [19–21,26–29] that uses the reachability-based localization.

In this article, we present a localization technique that is more aggressive than reachability-based approach. In addition to excluding unreachable memory entries from the localized state, we also exclude some memory entries that are reachable but will not be accessed. We attack the problem of localization by staging: (1) the set of abstract locations that will be used during the analysis of a code block is conservatively estimated by a pre-analysis; (2) then, the actual analysis uses the information and trims input memories before analyzing each block. The pre-analysis is derived by applying conservative abstractions to the abstract semantics of the original analysis and quickly finds an over-approximation of resources that the actual analysis requires. By reducing the sizes of localized memory states, our technique saves more analysis time and memory than the reachability-based approach does.

The time savings by our new localization method are significant: when applied to each procedure's body, our access-based localization reduces the analysis time by on average 92.1% over reachability-based localization. We implemented our approach inside an industrial-strength interval-domain-based abstract interpreter [19–21,26–29]. In experiments, the

[1] The reachable- and accessed-memory ratio is an average over all procedures. We ran the reachability-based analysis and recorded, for every analysis of procedures, the sizes of localized memory and its accessed portion. We averaged the size ratio over the total number of analyses of procedures.

```
1: struct S { int a; int b; }
2: int g = 0;
3: void f (S* p) { p->a = 0; }
4: void main() {
5:     S *s = (S*)malloc(sizeof S);
6:     s->a = 0;
7:     s->b = 0; f(s);       // first call to f
8:     s->b = 1; f(s); }     // second call to f
```

$$
\begin{array}{rcl}
s & \mapsto & \langle l_5, \{a, b\}\rangle \\
p & \mapsto & \langle l_5, \{a, b\}\rangle \\
\langle l_5, a\rangle & \mapsto & [0, 0] \\
\langle l_5, b\rangle & \mapsto & [0, 0] \\
g & \mapsto & [0, 0]
\end{array}
$$

(a) Non-localized memory.

$$
\begin{array}{rcl}
p & \mapsto & \langle l_5, \{a, b\}\rangle \\
\langle l_5, a\rangle & \mapsto & [0, 0] \\
\langle l_5, b\rangle & \mapsto & [0, 0] \\
g & \mapsto & [0, 0]
\end{array}
$$

(b) Reachability-based localization.

$$
\begin{array}{rcl}
p & \mapsto & \langle l_5, \{a, b\}\rangle \\
\langle l_5, a\rangle & \mapsto & [0, 0]
\end{array}
$$

(c) Access-based localization.

Fig. 1. Example code and abstract memories right before calling procedure `f` at line 7.

technique reduces analysis time by 78.5%–98.5%, on average 92.1%, and peak memory consumption by 33.0%–81.2%, on average 71.2%, over the reachability-based approach for a variety of open-source C benchmarks (2K–105KLOC). Moreover, our technique enables the largest four programs of our benchmarks to be analyzed, which could not be analyzed with the reachability-based approach because of the analysis running out of memory.

In addition, we generalize the idea of access-based localization in three ways.

- We apply the technique even to arbitrary code blocks other than procedure bodies. When applying localization to such smaller code blocks, we have to carefully select localization targets because localizing operations introduce performance overhead. We present a block selection strategy that is flexible to balance actual cost reduction against the overhead. The generalized localization reduces the analysis time by 8.5%–53.7%, on average 31.8%, on top of the procedure-level localization.
- We extend the access-based localization to support efficient handling of large recursive call cycles. We show that real C programs often have large recursive call cycles and localization is ineffective for such programs. Our extension, called bypassing, aims to alleviate the problem and shows 9%–79%, on average 42%, improved performance in analyzing recursive procedures.
- We adapt access-based localization to some relational numeric analyses. Though localizing relational analysis in general is complicated, we show that the packed relational analysis [25], a popular form of practical relational analysis, naturally fits to our technique. We define a simple localizing relational analysis and show that access-based localization improves the analysis performance by more than 80%.

Contributions. This paper, which is an extended version of [27,29], makes the following contributions.

- We introduce a new approach to localization in global static analysis. Compared to the conventional reachability-based approach, our access-based approach considers only the memory locations that will actually be accessed during the analysis. As far as we know, published program analyzers do not perform access-based localization: previous analyses use pure reachability-based techniques (e.g., [24,15,31]) or their variants (e.g., [8,23]).
- We report a real problem and solution. In fact, access-based localization is theoretically a simple extension of the reachability-based localization. However, we show, for the first time, that such simple extension has a significant impact in practice. The necessity of access-based approach has been mostly ignored in the literature.
- We evaluate our localization technique on top of a realistic C analyzer, Airac. Airac is an industrial-strength static analyzer [19–21,26–29] using intervals- and allocation-site-based abstractions of numeric and pointer values.
- We present three extensions of access-based localization: the block-level localization, access-based localization with bypassing, and localizing relational numeric analysis. These extensions of localization are all new and have not been addressed in the literature.

Example. We illustrate our localization technique and compare it with the reachability-based localization approach. Consider the C code in Fig. 1 and an interval analysis of the code. The analysis begins with an empty memory state ($\lambda x.\bot$). The abstract memory state right before calling `f` at line 7 (after the parameter is bound) is represented by Fig. 1(a). Here, s denotes a structure with fields $\{a, b\}$ allocated at line 5. The abstract locations of each field are represented by $\langle l_5, a\rangle$ and $\langle l_5, b\rangle$, which initially have bottom values. p is a parameter of `f` and g is a global variable.

Reachability-based localization collects all reachable memory entries: global variable g, parameter p, and structure fields $\langle l_5, a\rangle$ and $\langle l_5, b\rangle$ that are reachable by dereferencing p. Fig. 1(b) shows the resulting localized memory.

Our approach additionally filters the memory entries for $\langle l_5, b\rangle$ and g. Our pre-analysis infers that only the abstract locations $\{p, \langle l_5, a\rangle\}$ could be accessed during actual analysis of `f`. The actual analysis uses the results and trims memory entries, resulting in the memory state shown in Fig. 1(c). Note that, because the localized memory (Fig. 1(c)) does not contain

$\langle l_5, b\rangle$, the update to location $\langle l_5, b\rangle$ at line 8 does not cause `f` to be re-analyzed at the subsequent call to `f` (line 8). On the other hand, with reachability-based localization, `f` will be analyzed again at the second call.

Outline. Section 2 presents the baseline analyzer on top of which we will develop our localization techniques. Section 3 defines the conventional reachability-based localization. Section 4 develops our access-based localization. Section 5 extends the localization for arbitrary code blocks. Section 6 extends the localization to efficiently handle recursive call cycles. Section 7 extends the localization to relational analysis. Section 8 evaluates the proposed techniques. Section 9 presents related work and discussion. Section 10 concludes the paper.

Notation. Throughout this paper, we use the following notations. Given a set X, $\mathcal{P}(X)$ denotes the powerset of X. Given a function $f : A \to B$ and a set $X \subseteq A$, $f|_X$ denotes function restriction: $f|_X(x) = f(x)$ if $x \in X$, otherwise $f|_X(x) = \bot$. We abuse the notation $f|_a$ for the domain restrictions on singleton set $\{a\}$. We write $f[a \mapsto b]$ to mean the function we get from function f by changing the value for a to b. For all of the domains, we assume an implicit and appropriate $\sqsubseteq$, $\top$, and $\bot$ for domains that need them. We write $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We write $f[\{a_1, \ldots, a_n\} \overset{w}{\mapsto} b]$ for $f[a_1 \mapsto f(a_1) \sqcup b, \ldots, a_n \mapsto f(a_n) \sqcup b]$ (weak update). When X is a tuple, $X.n$ indicates the nth component of the tuple. Finally, $\mathbb{N}$ represents the set of natural numbers $\{0, 1, 2, \ldots\}$.

2. A parametric non-relational static analysis for C-like languages

In this section, we define a parametric static analysis for imperative programs, on top of which we develop our localization technique in the next section. The analysis is parametric in that its abstract numeric domain is parameterized so we can instantiate the analysis to obtain any non-relational analysis for imperative programs. We present the program representation (Section 2.1), collecting semantics (Section 2.2), and abstract semantics (Section 2.3).

In this article, we assume the sorts of static analyses with the following assumptions. We suppose that the analysis is context-insensitive and the set of concrete memory locations are abstracted by a finite set of abstract locations. Specifically, the analysis uses allocation site-based heap abstraction where all the memory locations allocated at an allocation site are collectively represented by a single abstract location.

2.1. Programs

We define the program and language our static analyzer considers. We assume that a program is represented by a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ where $\mathbb{C}$ is a finite set of control points (or program points) and $\hookrightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is the control flow relation of the program; $c' \hookrightarrow c$ indicates that c is a successor control point of c'. Each control point c is associated with command $\mathsf{cmd}(c)$. Command c has one of the following five types:

$$\mathsf{assign}(lv, e) \mid \mathsf{alloc}(lv, a) \mid \mathsf{assume}(x < n) \mid \mathsf{call}(f_x, e) \mid \mathsf{return}_f$$

where expression e, l-value expression lv, and allocation expression a are defined as follows:

$$\begin{array}{lrl} \text{expression} & e & \to\ n \mid e + e \mid lv \mid \&lv \\ \text{l-value} & lv & \to\ x \mid *e \mid e[e] \mid e.x \\ \text{allocation} & a & \to\ [e]_l \mid \{x\}_l \end{array}$$

An expression may be a constant integer (n), a binary operation ($e + e$), an l-value expression (lv), or an address-of expression ($\&lv$). An l-value may be a variable (x), a pointer dereference ($*e$), an array access ($e[e]$), or a field access ($e.x$). Expressions and l-value expressions have no side-effects. (We assume that all memory accesses are valid within expressions and l-value expressions). All program variables, including formal parameters, have unique names. Command $\mathsf{assign}(lv, e)$ assigns the value of e into the location of lv. Command $\mathsf{alloc}(lv, a)$ allocates an array $[e]_l$ or a structure $\{x\}_l$, where e is the size of the array, x is the field name, and the subscript l is the label for the allocation site. Multidimensional arrays are represented by nested allocation commands: in order to allocate a two dimensional array $a[e_1][e_2]$, we first allocate a with size e_1 and allocate each $a[i](0 \le i < e_1)$ with arrays of size e_2. An assume command $\mathsf{assume}(x < n)$ makes the program continue only when the condition evaluates to true. For simplicity, we consider structures with one field only but generalizing it to multiple fields is immediate (our abstract domain covers the general case). Each call-site for a procedure is represented by two control points: a call-point and its corresponding return-point. A call-point is associated with command $\mathsf{call}(f_x, e)$, which indicates that procedure f, whose formal parameter is x, is called with actual parameter e. For simplicity, we assume that there are no function pointers[2] and consider procedures with one parameter only. Command return_f denotes the return statement of procedure f.

2.2. Collecting semantics

The collecting semantics of program P is an invariant $[\![P]\!] \in \mathbb{C} \to \mathcal{P}(\mathbb{S})$ that represents the set of reachable states at each control point, where the concrete domain of states, $\mathbb{S}$, is defined as $\mathbb{S} = \mathbb{L} \to \mathbb{V}$: concrete state $s \in \mathbb{S}$ is a map

[2] In implementation, we resolve all function pointers a priori with a pre-analysis.

from locations ($\mathbb{L}$) to values ($\mathbb{V}$). The collecting semantics is characterized by the least fixpoint of the semantic function $F \in (\mathbb{C} \to \mathcal{P}(\mathbb{S})) \to (\mathbb{C} \to \mathcal{P}(\mathbb{S}))$ such that,

$$F(X) = \lambda c \in \mathbb{C}. \bigcup_{c' \hookrightarrow c} f_{c'}(X(c'))$$

where $f_c \in \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ is a semantic function at control point c, which transfers the input state of c into its output state depending on the commands associated with c. We leave out the standard definition of the concrete semantic function.

2.3. Abstract semantics

Abstract domain. We abstract the collecting semantics of program P by the following Galois connection:

$$\mathbb{C} \to \mathcal{P}(\mathbb{S}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{C} \to \hat{\mathbb{S}}$$

where α and γ are pointwise liftings of abstract and concretization function $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$ (such that $\mathcal{P}(\mathbb{S}) \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$), respectively. That is, we abstract the set of reachable states by a single abstract state. An abstract memory state

$$\hat{\mathbb{S}} = \hat{\mathbb{L}} \to \hat{\mathbb{V}}$$

denotes a finite map from abstract locations ($\hat{\mathbb{L}}$) to abstract values ($\hat{\mathbb{V}}$).

$$\hat{\mathbb{L}} = \mathit{Var} + \mathit{AllocSite} + \mathit{AllocSite} \times \mathit{FieldName}$$
$$\hat{\mathbb{V}} = \hat{\mathbb{Z}} \times \mathcal{P}(\hat{\mathbb{L}}) \times \mathcal{P}(\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}) \times \mathcal{P}(\mathit{AllocSite} \times \mathcal{P}(\mathit{FieldName}))$$

An abstract location may be a program variable (*Var*), an allocation site (*AllocSite*), or a structure field (*AllocSite* × *FieldName*). All the elements of an array allocated at allocation site l are collectively represented by l. The abstract location for field x of a structure allocated at l is represented by $\langle l, x \rangle$. An abstract value is a quadruple. Numeric values are tracked by the first component, abstract numerical value $\hat{\mathbb{Z}}$. The analysis is parametric so we can choose any non-relational numeric domains for $\hat{\mathbb{Z}}$, such as the lattice of intervals ($\{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \leq u\} \cup \{\bot\}$) or the constant propagation lattice ($\mathbb{Z} \cup \{\bot, \top\}$). Points-to information is kept by the second component ($\mathcal{P}(\hat{\mathbb{L}})$): it indicates pointer targets an abstract location may point to. Allocated arrays of memory locations are represented by array blocks ($\mathcal{P}(\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}})$): an array block $\langle l, o, s \rangle$ consists of an abstract base address (l), offset (o), and size (s). A structure block $\langle l, \{x\} \rangle \in \mathcal{P}(\mathit{AllocSite} \times \mathcal{P}(\mathit{FieldName}))$ abstracts structure values that are allocated at l and have a set of fields $\{x\}$.[3] Note that all the domains (such as $\hat{\mathbb{L}}$) are finite except that the abstract numerics ($\hat{\mathbb{Z}}$) could be infinite. When $\hat{\mathbb{Z}}$ is infinite, we apply a widening operation [14] to ensure the termination of the analysis.

Abstract semantic function. Abstract semantics is characterized by the least fixpoint of abstract semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$ defined as,

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}. \bigsqcup_{c' \hookrightarrow c} \hat{f}_{c'}(\hat{X}(c')) \tag{1}$$

where $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is a semantic function at control point c:

$$\hat{f}_c(\hat{s}) = \begin{cases} \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = \mathsf{assign}(lv, e) \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \{\langle l, \alpha_{\hat{\mathbb{Z}}}(0), \hat{\mathcal{V}}(e)(\hat{s}).1 \rangle\}, \bot \rangle] & \mathsf{cmd}(c) = \mathsf{alloc}(lv, [e]_l) \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \bot, \{\langle l, \{x\} \rangle\} \rangle] & \mathsf{cmd}(c) = \mathsf{alloc}(lv, \{x\}_l) \\ \hat{s}[x \mapsto \langle \hat{s}(x).1 \sqcap_{\hat{\mathbb{Z}}} \alpha_{\hat{\mathbb{Z}}}(\{z \in \mathbb{Z} \mid z < n\}), \hat{s}(x).2, \hat{s}(x).3, \hat{s}(x).4 \rangle] & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \\ \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = \mathsf{call}(f_x, e) \\ \hat{s} & \mathsf{cmd}(c) = \mathsf{return}_f \end{cases}$$

Auxiliary functions $\hat{\mathcal{V}}(e)(\hat{s})$ and $\hat{\mathcal{L}}(lv)(\hat{s})$ compute abstract values for e and abstract locations for lv, respectively, under $\hat{s}$. The effect of node $\mathsf{assign}(lv, e)$ is to (weakly) update the abstract value of e into abstract locations $\hat{\mathcal{L}}(lv)(\hat{s})$.[4] The array allocation command $\mathsf{alloc}(lv, [e]_l)$ creates a new array block with offset 0 and size e. The structure block command $\mathsf{alloc}(lv, \{x\}_l)$ creates a new structure block. In both cases, we use the allocation site l as the base address (that is, we use the allocation-site abstraction), by which many (possibly unbounded/infinite) concrete locations are summarized by finite abstract locations.

[3] We could also design a structure block by $\mathcal{P}(\mathit{AllocSite} \times \mathit{FieldName})$, which has the same expressive power as $\mathcal{P}(\mathit{AllocSite} \times \mathcal{P}(\mathit{FieldName}))$. However, we have chosen the latter to remove redundancies in representing base addresses. For example, suppose a structure with two fields x and y is allocated at allocation-site l. We represent the structure by $\{\langle l, \{x, y\} \rangle\}$, not by $\{\langle l, x \rangle, \langle l, y \rangle\}$.

[4] For brevity, we consider only weak updates. Applying strong updates is orthogonal to our localization techniques.

Our abstract semantics sets the initial contents of the newly allocated location l to be bottom, which will be changed to actual initial values by the programs.[5] Command $\mathsf{assume}(x < n)$ confines the value of x so that the resulting memory state satisfies the condition. The call command $\mathsf{call}(f_x, e)$ binds the formal parameter x to the value of actual parameter e. Note that the output of the call node is the memory state that flows into the body of the called procedure, not the memory state returned from the call. The abstract semantics for procedure calls show that our analysis is context-insensitive: it ignores the calling context in which procedures are invoked.

We now define $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, which compute abstract values and locations, respectively. Given expression e and abstract state $\hat{s}$, $\hat{\mathcal{V}}(e)(\hat{s})$ evaluates the abstract value of e under $\hat{s}$.

$$\begin{aligned}
\hat{\mathcal{V}}(e) &\in \hat{\mathbb{S}} \to \hat{\mathbb{V}} \\
\hat{\mathcal{V}}(n)(\hat{s}) &= \langle \alpha_{\hat{\mathbb{Z}}}(n), \bot, \bot, \bot \rangle \\
\hat{\mathcal{V}}(e_1 + e_2)(\hat{s}) &= \hat{\mathcal{V}}(e_1)(\hat{s}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{s}) \\
\hat{\mathcal{V}}(lv)(\hat{s}) &= \bigsqcup \{\hat{s}(l) \mid l \in \hat{\mathcal{L}}(lv)(\hat{s})\} \\
\hat{\mathcal{V}}(\&lv)(\hat{s}) &= \langle \bot, \hat{\mathcal{L}}(lv)(\hat{s}), \bot, \bot \rangle
\end{aligned}$$

$\hat{\mathcal{V}}$ is inductively defined for each type of expression. Integer n evaluates to its corresponding interval value $[n, n]$. Expressions involving binary operators are inductively evaluated. For l-values lv, we first find the abstract locations that lv denotes and then look up the abstract values associated with the locations. $\&lv$ evaluates to the abstract locations that lv denotes. Note that the analysis is parameterized by abstract binary ($\hat{+}_{\hat{\mathbb{Z}}}$), join ($\sqcup_{\hat{\mathbb{Z}}}$), and meet ($\sqcap_{\hat{\mathbb{Z}}}$) operations for abstract numeric domain $\hat{\mathbb{Z}}$, from which those operators for abstract values are defined in a standard way.

Similarly, given l-value expression lv and abstract memory state $\hat{s}$, $\hat{\mathcal{L}}(lv)(\hat{s})$ evaluates the set of abstract locations that lv denotes under $\hat{s}$.

$$\begin{aligned}
\hat{\mathcal{L}}(lv) &\in \hat{\mathbb{S}} \to \mathcal{P}(\hat{\mathbb{L}}) \\
\hat{\mathcal{L}}(x)(\hat{s}) &= \{x\} \\
\hat{\mathcal{L}}(*e)(\hat{s}) &= \hat{\mathcal{V}}(e)(\hat{s}).2 \cup \{l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).3\} \cup \{\langle l, x \rangle \mid \langle l, X \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).4 \wedge x \in X\} \\
\hat{\mathcal{L}}(e_1\,[e_2])(\hat{s}) &= \{l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e_1)(\hat{s}).3\} \\
\hat{\mathcal{L}}(e\,.\,x)(\hat{s}) &= \{\langle l, x \rangle \mid \langle l, X \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).4 \wedge x \in X\}
\end{aligned}$$

The abstract location for variable x is represented by x. When $*e$ is used as an l-value, it denotes all the abstract locations that e evaluates to, including arrays and structure fields. Array access $e_1\,[e_2]$ refers to the location of arrays e_1 denotes. In our analysis, all of the array elements are smashed into a single element, and hence, the definition of $\hat{\mathcal{L}}(e_1\,[e_2])$ does not involve e_2. As expected, the abstract location for structure field $e\,.\,x$ is represented by a pair of allocation site and field name.

Our abstract domain and semantics are rather conventional, similar to those used in other abstract interpretations for C programs or executable codes (e.g., [2,3]). Our abstract semantics estimate numeric and pointer values within a monolithic abstract interpretation. A benefit of combining the two analyses is that information about numeric values can improve the pointer analysis, and pointers information can improve the numeric analysis, as studied in [30].

Notes on soundness. In practice, it is difficult to design an abstract semantics that is sound with respect to the C language because, for example, there is no formal definition of C semantics and C allows random memory accesses via arbitrary pointer values. Thus, static analyzers for C are usually designed on top of their own intermediate language (IL) and the soundness is guaranteed with respect to the IL. We can also claim that our abstract semantics is sound with respect to the semantics of our intermediate language (a variant of our soundness proof is available at [33]) but not with respect to the C language. In practice, we try to close the gap between the semantics of C and IL when translating C into IL. For instance, in our intermediate language, we assume that there is no expressions like `*&*&x`. So, during the translation, we simplify `*&*&x` into `x`. When we cannot translate it correctly, we ignore the expression and replace it with the top value.

2.4. Fixpoint algorithm

Worklist algorithm. We compute the least fixpoint of abstract semantic function given in (1) by a worklist-based fixpoint algorithm (Fig. 2). Unlike the naive fixpoint algorithm, which would directly iterate $\hat{F}^i$, the worklist algorithm evaluates only the program points whose abstract states has changed. The worklist, W, consists of those control points whose abstract states are not yet stabilized. For each iteration of the loop, a control point is selected (`choose`) and evaluated. When the new output memory state $\hat{s}$ is changed, next control points of c are added to the worklist and the updated information is stored. We use weak topological ordering for worklist management [6].

[5] In programs, structures are allocated with their fields immediately initialized. Otherwise, we translate the program so that uninitialized locations are explicitly initialized to "top" values in our intermediate language.

$W \in \mathit{Worklist} = \mathcal{P}(\mathbb{C})$
$\hat{X} \in \mathbb{C} \to \hat{\mathbb{S}}$
$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$
$W := \mathbb{C}$
$\hat{X} := \lambda c.\bot$
repeat
  $c := \mathsf{choose}(W)$
  $W := W - \{c\}$
  $\hat{s}_{in} := \bigsqcup_{c' \hookrightarrow c} \hat{f}_{c'}(\hat{X}(c'))$
  if $\hat{s}_{in} \not\sqsubseteq \hat{X}(c)$
    if c is a head of a flow cycle
      $\hat{s}_{in} := \hat{X}(c) \triangledown \hat{s}_{in}$
    $\hat{X}(c) := \hat{s}_{in}$
    $W := W \cup \{c' \mid c \hookrightarrow c'\}$
until $W = \emptyset$

Fig. 2. The worklist-based fixpoint computation algorithm. The algorithm uses the widening operation ($\triangledown$), which is necessary for analysis' termination.

The performance of the worklist algorithm is sensitive to the iteration strategy (that is, the actual implementation of choose). There is a large body of related works on worklist ordering strategy, and there is no best solution to the problem. In experiments, we used the so-called nested reverse topological ordering [28]. In this ordering, the order between control points is defined as a reverse topological order between procedures on the call graph: a control point n of a procedure f precedes a control point m of a procedure g if f precedes g in the reverse topological order in the call graph. If f and g are the same procedure, the order between the nodes is defined by the reverse topological order of the control flow graph of that procedure. Strongly connected components are considered as single entities. Note that there can be two or more nodes that have the highest order, for example of each branch of conditional statements. In this case, the algorithm arbitrarily chooses a node among them. The ordering consistently shows better performance than naive worklist management scheme (BFS/DFS) or simple "wait-at-join" techniques (e.g., [20]).

Widening. When the abstract domain is of infinite height, we need to apply a widening operator during the fixpoint computation. Widening [14] is a speed-up technique designed to safely approximate least fixpoints of semantic function. In abstract-interpretation-based static analysis, program invariants are characterized as least fixpoints of (abstract) semantic functions over an abstract domain. For finite height domains, the fixpoints can be computed (in finite time) by using a classical iterative algorithm. But the iterative algorithm does not terminate or has unacceptable costs for domains with infinite height or very large height. For infinite or very large height domains such as the lattice of intervals, the widening technique [14] is used to guarantee and accelerate the analysis' termination. With widening, the iterative algorithm does not necessarily compute least fixpoints but finds a safe (upper) approximation of the least fixpoint. In our abstract domain, only the numerical domain ($\hat{\mathbb{Z}}$) requires widening (if it has an infinite height); the other domains are all finite. For example, our static analyzer Airac uses the lattice of intervals and we use the following widening operation $\triangledown$ for intervals [12]:

$$\begin{aligned} \bot \triangledown X &= X \\ X \triangledown \bot &= X \\ [l_0, u_0] \triangledown [l_1, u_1] &= [0 \le l_1 < l_0 \,?\, 0 : (l_1 < l_0 \,?\, -\infty : l_0), \\ &\quad\; 0 \ge u_1 > u_0 \,?\, 0 : (u_0 < u_1 \,?\, +\infty : u_0)] \end{aligned}$$

In Fig. 2, we abuse the notation $\triangledown$ for the widening operations between abstract states, which is a pointwise lifting of the interval-widening operation. In the analysis, the widening is applied only to loop heads of flow graphs represented by $\hookrightarrow$.

3. Reachability-based localization

In this section, we define the reachability-based localization of abstract memories. We formalize the technique on top of the parametric static analyzer defined in the previous section. The reachability-based technique will be extended to the access-based technique in the next section.

3.1. Conventional reachability-based localization

Given an input abstract memory to a procedure, unless we actually analyze the procedure with the input memory, it is generally impossible to infer the exact set of abstract locations that will be accessed during the analysis of the procedure. Instead, the conventional localization technique uses a reachability heuristic, which is based on the simple intuition

that unreachable abstract locations cannot be accessed in the future of the analysis. With reachability-based localization, procedures are analyzed as follows:

1. Before entering the procedure, the analysis stops and searches the input memory to collect abstract locations that are reachable from abstract locations for global variables or actual parameters.
2. Then, the input memory is restricted to the collected, reachable abstract locations.
3. The called procedure is analyzed with only the reachable portion of the input memory, not the entire input memory.

Formally, given a call $\mathsf{call}(f_x, e)$ and its input abstract memory state $\hat{s}$ (parameter-bound), $\mathsf{Airac}_{\mathsf{Reach}}$ computes the following set of abstract locations (let $\mathsf{Globals}$ be the set of abstract locations that represent global variables in the program):

$$\mathcal{R}(f_x, \hat{s}) = \mathsf{Reach}(\mathsf{Globals}, \hat{s}) \cup \mathsf{Reach}(\{x\}, \hat{s})$$

which is the union of abstract locations that are reachable from global variables ($\mathsf{Globals}$) and those that are reachable from the parameter (x). We use $\mathsf{Reach}(X, \hat{s})$ to denote the set of abstract locations in $\hat{s}$ that are (directly or transitively) reachable from the location set X.

$$\mathsf{Reach}(X, \hat{s}) = \mathbf{lfp}(\lambda Y.X \cup \mathsf{OneHop}(Y, \hat{s}))$$

$\mathsf{OneHop}(X, \hat{s})$ is the set of locations that are directly reachable from X:

$$\mathsf{OneHop}(X, \hat{s}) = \bigcup_{x \in X} \hat{s}(x).2 \cup \{l \mid \langle l, o, s\rangle \in \hat{s}(x).3\} \cup \{\langle l, f\rangle \mid \langle l, F\rangle \in \hat{s}(x).4 \wedge f \in F\}$$

Given an input memory $\hat{s}$ to a call-point c (such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$), the definition of the semantic function $\hat{f}_c$ is changed as follows: (Note that the output from the semantic function is the memory that flows into the body of the called procedure, not the memory that is returned from the body.)

$$\hat{f}_c(\hat{s}) = \hat{s}'|_{\mathcal{R}(f_x, \hat{s}')} \quad \text{where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

That is, when a procedure is being analyzed, its input abstract memory is restricted to reachable abstract locations.

We also have to consider procedure returns. Because the analysis is localized, the abstract memory state returned from the exit of the callee procedure does not have enough information to continue the rest of caller procedure. Thus, at return point, we combine (via join operation for abstract memories) the non-localized memory parts at the corresponding call point with the returned memory from the callee. This join operation is sound because every object is represented by a fixed abstract location in our abstract memories.

4. Access-based localization

This section describes our access-based localization technique. Our technique more tightly localizes abstract memories than the reachability-based technique does: whereas the reachability-based approach only strips away unreachable abstract locations, our method additionally filters out the abstract locations that are reachable but definitely not to-be-accessed during the analysis.

In order to compute such tighter information, we take a "static" approach. That is, with our technique, all the abstract locations that will be accessed during the analysis are already prepared from the beginning of the analysis. On the other hand, the reachability-based localization is "dynamic" in a sense that the estimation process, which collects possibly accessed abstract locations, is simultaneously performed during the analysis. We separate the entire analysis into two phases:

1. **Pre-analysis**: the set of abstract locations that are accessed during the actual analysis of each procedure are conservatively estimated.
2. **Actual analysis**: the actual analysis uses the access-information and filters out memory entries that will definitely not be accessed by called procedures.

A careful design of pre-analysis is important for both safety and efficiency of our approach. To be safe, the pre-computed access information should be conservative with respect to the abstract locations that would be accessed during the actual analysis. To be useful, it should be efficient enough to compensate for the extra burden of running the pre-analysis.

Before discussing the analysis, we need to clarify the meaning of 'accessed'. In our analysis, the abstract entities that are accessed during the analysis are abstract locations. We say an abstract location $a \in \hat{\mathbb{L}}$ is *accessed* if an abstract value is read by referencing the abstract location or an abstract value is written to a.

Example 1. Consider a semantic function $\hat{f} = \lambda\hat{s}.\hat{s}[x \mapsto \hat{s}(y)]$, where $\hat{s}$ represents the input memory state and $\hat{f}(\hat{s})$ computes its output state. When evaluating $\hat{f}(\hat{s})$, the accessed locations are x and y: location x is written, and y is dereferenced.

In Section 4.1, we design a pre-analysis and prove its safety and Section 4.2 describes the actual analysis.

4.1. Pre-analysis

The pre-analysis aims to compute a map $\mathsf{access} \in \mathbb{C} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$ that, for each control point, conservatively estimates a set of abstract locations that are possibly accessed during the actual analysis of the control point. In order to find such a map, we use the following strategy:

- We define the 'access function' that computes access information for each command in the program. The access function is a summarized version of semantics function $\hat{f}_c$, focusing only on the read/write behaviors instead of describing full semantics of $\hat{f}_c$.
- We define a conservative abstraction of the original analysis, whose analysis results over-approximate input memory states that occur in the analysis.
- By using the access function and over-approximated analysis results, the accessed abstract locations for each control point are conservatively estimated.

Access function. We define the access function, $\mathcal{A} \in \mathbb{C} \to \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$, that computes which abstract locations are accessed during the analysis of a control point on an input memory state. The intuition is that, given control point c and its input abstract memory $\hat{s}$, $\mathcal{A}(c)(\hat{s})$ computes the set of abstract locations that are accessed during the evaluation of $\hat{f}_c(\hat{s})$. Note that an abstract location a is accessed during the evaluation of $\hat{f}_c(\hat{s})$ if a is referenced (i.e., $\hat{s}(a)$ appears in the definition of $\hat{f}_c$) or a value v is written to a (i.e., $\hat{s}[a \mapsto v]$ appears in the definition) during the evaluation.

In order to define function $\mathcal{A}$, we first define two sub-functions $\hat{\mathcal{AV}} \in e \to \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$ and $\hat{\mathcal{AL}} \in lv \to \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$. Given expression e and memory state $\hat{s}$, $\hat{\mathcal{AV}}(e)(\hat{s})$ computes abstract locations that are accessed during the evaluation of $\hat{\mathcal{V}}(e)(\hat{s})$. Similarly, $\hat{\mathcal{AL}}(lv)(\hat{s})$ computes abstract locations that are accessed during the evaluation of $\hat{\mathcal{L}}(lv)(\hat{s})$. $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$ are defined as follows. (Note that the base cases in the following definitions are not always empty; $\hat{\mathcal{AV}}(lv)(\hat{s})$ is defined to contain $\hat{\mathcal{L}}(lv)(\hat{s})$ that is mostly not empty.)

$$
\begin{array}{rcl@{\qquad}rcl}
\hat{\mathcal{AV}}(e) & \in & \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}}) & \hat{\mathcal{AL}}(lv) & \in & \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}}) \\
\hat{\mathcal{AV}}(n)(\hat{s}) & = & \emptyset & \hat{\mathcal{AL}}(x)(\hat{s}) & = & \emptyset \\
\hat{\mathcal{AV}}(e_1 + e_2)(\hat{s}) & = & \hat{\mathcal{AV}}(e_1)(\hat{s}) \cup \hat{\mathcal{AV}}(e_2)(\hat{s}) & \hat{\mathcal{AL}}(*e)(\hat{s}) & = & \hat{\mathcal{AV}}(e)(\hat{s}) \\
\hat{\mathcal{AV}}(lv)(\hat{s}) & = & \hat{\mathcal{AL}}(lv)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \hat{\mathcal{AL}}(e_1\,[e_2])(\hat{s}) & = & \hat{\mathcal{AV}}(e_1)(\hat{s}) \\
\hat{\mathcal{AV}}(\&lv)(\hat{s}) & = & \hat{\mathcal{AL}}(lv)(\hat{s}) & \hat{\mathcal{AL}}(e.x)(\hat{s}) & = & \hat{\mathcal{AV}}(e)(\hat{s})
\end{array}
$$

These definitions are naturally derived from the definitions of semantic functions $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$. Consider $\hat{\mathcal{AV}}$ (defined in the left column) first. When $e = n$, we see that the definition of $\hat{\mathcal{V}}$ (in Section 2.3) does not read (nor write to) any location, and hence there are no accessed locations ($\hat{\mathcal{AV}}(n)(\hat{s}) = \emptyset$). When $e = e_1 + e_2$, accessed locations are collected recursively. When an l-value lv is used as an r-value (the third case), from the definition of $\hat{\mathcal{V}}(lv)(\hat{s})$, we see that abstract locations of lv and abstract locations that are accessed during the evaluation of $\hat{\mathcal{L}}(lv)(\hat{s})$ are accessed, which are collected by $\hat{\mathcal{L}}(lv)(\hat{s})$ and $\hat{\mathcal{AL}}(lv)(\hat{s})$, respectively. When an l-value lv is used as an address-of expression (the fourth case), from the definition of $\hat{\mathcal{V}}(\&lv)(\hat{s})$, we see that abstract locations that lv denotes ($\hat{\mathcal{L}}(lv)(\hat{s})$) are not accessed during the evaluation of $\hat{\mathcal{V}}(\&lv)(\hat{s})$ and hence the fourth case only includes $\hat{\mathcal{AL}}(lv)(\hat{s})$. Similarly, the definition of $\hat{\mathcal{AL}}$ (defined in the right column) is derived from the definition of $\hat{\mathcal{L}}$. $\hat{\mathcal{AL}}(x)(\hat{s})$ is $\emptyset$ because $\hat{\mathcal{L}}(x)(\hat{s})$ just produces a location x but does not read (resp., write) any value from (resp., to) x. The second case holds because computing $\hat{\mathcal{L}}(*e)(\hat{s})$ only accesses locations that are accessed during the evaluation of e. Likewise, the fourth case holds. Lastly, the third case holds: we collect only the locations accessed during the evaluation of e_1 because $\hat{\mathcal{L}}(e_1\,[e_2]\,)(\hat{s})$ does not involve the evaluation of e_2 (we smash all the array elements into one memory cell).

Example 2. Consider expression $*x$ and suppose the current abstract memory is $\hat{s} = [x \mapsto p, p \mapsto v]$. The locations that are accessed during the evaluation of $\hat{\mathcal{V}}(*x)(\hat{s})$ are $\{x, p\}$, which is derived as follows from the definition of $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$:

$$
\begin{array}{rcl}
\hat{\mathcal{AV}}(*x)(\hat{s}) & = & \hat{\mathcal{AL}}(*x)(\hat{s}) \cup \hat{\mathcal{L}}(*x)(\hat{s}) \\
& = & \hat{\mathcal{AV}}(x)(\hat{s}) \cup \hat{\mathcal{L}}(*x)(\hat{s}) \\
& = & (\hat{\mathcal{AL}}(x)(\hat{s}) \cup \hat{\mathcal{L}}(x)(\hat{s})) \cup \hat{\mathcal{L}}(*x)(\hat{s}) \\
& = & (\emptyset \cup \hat{\mathcal{L}}(x)(\hat{s})) \cup \hat{\mathcal{L}}(*x)(\hat{s}) \\
& = & \{x\} \cup \{p\} = \{x, p\}
\end{array}
$$

As another example, consider expression $\&p[1]$ and abstract memory $\hat{s} = [p \mapsto \langle l, o, s\rangle, l \mapsto v]$. The locations that are accessed during the evaluation of $\hat{\mathcal{V}}(\&p[1])(\hat{s})$ are $\{p\}$, which is derived as follows:

$$
\begin{array}{rcl}
\hat{\mathcal{AV}}(\&p[1])(\hat{s}) & = & \hat{\mathcal{AL}}(p[1])(\hat{s}) \\
& = & \hat{\mathcal{AV}}(p)(\hat{s}) \\
& = & \hat{\mathcal{AL}}(p)(\hat{s}) \cup \hat{\mathcal{L}}(p)(\hat{s}) \\
& = & \emptyset \cup \hat{\mathcal{L}}(p)(\hat{s}) \\
& = & \{p\}
\end{array}
$$

Formally, the following lemmas show that $\hat{\mathcal{A}}\hat{\mathcal{V}}$ and $\hat{\mathcal{A}}\hat{\mathcal{L}}$ are correct and monotone.

Lemma 1. *For all expressions e, lv, and input memory $\hat{s}$,*

$$a \in \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \Leftrightarrow a \text{ is accessed during the evaluation of } \hat{\mathcal{V}}(e)(\hat{s})$$
$$a \in \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \Leftrightarrow a \text{ is accessed during the evaluation of } \hat{\mathcal{L}}(lv)(\hat{s})$$

Proof. By structural induction on *e* and *lv*. For example, consider the case for $e = e_1 + e_2$:

$$\begin{array}{lll} & a \in \hat{\mathcal{A}}\hat{\mathcal{V}}(e_1 + e_2)(\hat{s}) & \\ \Leftrightarrow & a \in \hat{\mathcal{A}}\hat{\mathcal{V}}(e_1)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e_2)(\hat{s}) & \cdots \text{ def. of } \hat{\mathcal{A}}\hat{\mathcal{V}} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{V}}(e_1)(\hat{s}) \text{ or } \hat{\mathcal{V}}(e_2)(\hat{s}) & \cdots \text{ induction hypothesis} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{V}}(e_1 + e_2)(\hat{s}) & \cdots \text{ def. of } \hat{\mathcal{V}}(e_1 + e_2)(\hat{s}) \end{array}$$

Other cases are proved similarly. □

Lemma 2. *$\hat{\mathcal{A}}\hat{\mathcal{V}}$ and $\hat{\mathcal{A}}\hat{\mathcal{L}}$ are monotone. That is, for all expression e, l-value lv and abstract memory states s and s′,*

$$s \sqsubseteq s' \implies \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(s) \subseteq \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(s')$$
$$s \sqsubseteq s' \implies \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(s) \subseteq \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(s')$$

Proof. By structural inductions on *e* and *lv*. □

Now we define the access function $\mathcal{A}$ that computes abstract locations that are accessed during the analysis of each command:

$$\mathcal{A}(c)(\hat{s}) = \begin{cases} \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = \mathsf{assign}(lv, e) \\ \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = \mathsf{alloc}(lv, [e]_l) \\ \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = \mathsf{alloc}(lv, \{x\}_l) \\ \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \{x\} & \mathsf{cmd}(c) = \mathsf{assume}(x < e) \\ \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \{x\} & \mathsf{cmd}(c) = \mathsf{call}(f_x, e) \\ \emptyset & \mathsf{cmd}(c) = \mathsf{return}_f \end{cases}$$

Similar to the derivation of $\hat{\mathcal{A}}\hat{\mathcal{V}}$ and $\hat{\mathcal{A}}\hat{\mathcal{L}}$, $\mathcal{A}$ has a semantic relationship with $\hat{f}_c$ and is thus naturally derived from the definition of $\hat{f}_c$. When $\mathsf{cmd}(c)$ is $\mathsf{assign}(lv, e)$ or $\mathsf{alloc}(lv, [e]_l)$, the definition of $\hat{f}_c(\hat{s}) = \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \hat{\mathcal{V}}(e)(\hat{s})]$ tells us that abstract locations $\hat{\mathcal{L}}(lv)(\hat{s})$ are written and read (because of the weak update), and those that are accessed during the evaluation of *lv* and *e* are also accessed ($\hat{\mathcal{A}}\hat{\mathcal{L}}(lv)$ and $\hat{\mathcal{A}}\hat{\mathcal{V}}(e)$, respectively). Thus, $\mathcal{A}(c)(\hat{s})$ is defined to be the union of these sets. When $\mathsf{cmd}(c) = \mathsf{alloc}(lv, \{x\}_l)$, $\hat{\mathcal{L}}(lv)(\hat{s})$ is accessed and locations accessed inside *lv* is inductively collected. When $\mathsf{cmd}(c)$ is $\mathsf{assume}(x < e)$ or $\mathsf{call}(f_x, e)$, note that *x* is accessed in addition to the locations that are accessed inside *e*. When $\mathsf{cmd}(c) = \mathsf{return}_f$, $\hat{f}_c$ does not access anything. Note that we could have factored the above definition of $\mathcal{A}$ using the equivalence $\hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) = \hat{\mathcal{A}}\hat{\mathcal{V}}(lv)(\hat{s})$, but we did not do this in order to emphasize the semantic relationship between $\mathcal{A}$ and $\hat{f}_c$. The following lemmas shows that $\mathcal{A}$ is indeed correct and monotone.

Lemma 3. *For all $c \in \mathbb{C}$, $\hat{s} \in \hat{\mathbb{S}}$, $a \in \hat{\mathbb{L}}$,*

$$a \in \mathcal{A}(c)(\hat{s}) \Leftrightarrow a \text{ is accessed during the evaluation of } \hat{f}_c(\hat{s})$$

Proof. By case analysis on the type of $\mathsf{cmd}(c)$. For example, consider the case for $\mathsf{cmd}(c) = \mathsf{assign}(lv, e)$:

$$\begin{array}{lll} & a \in \mathcal{A}(c)(\hat{s}) & \\ \Leftrightarrow & a \in \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \cdots \text{ def. of } \mathcal{A} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{L}}(lv)(\hat{s}) \text{ or } \hat{\mathcal{V}}(e)(\hat{s}) \text{ or } a \in \hat{\mathcal{L}}(lv)(\hat{s}) & \cdots \text{ Lemma 1} \\ \Leftrightarrow & a \text{ is accessed in } \hat{f}_c(\hat{s}) & \cdots \text{ def. of } \hat{f} \end{array}$$

Other cases are proved similarly. □

Lemma 4. *$\mathcal{A}$ is monotone, i.e., $\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}}$ and $\forall c \in \mathbb{C}$,*

$$\hat{s} \sqsubseteq \hat{s}' \implies \mathcal{A}(c)(\hat{s}) \subseteq \mathcal{A}(c)(\hat{s}')$$

Proof. By case analysis on type of $\mathsf{cmd}(c)$ and Lemma 2. □

$old, new \in \hat{\mathbb{S}}$
Preliminary $(old, new) =$
$new := \bot_{\hat{\mathbb{S}}}$
repeat
  $old := new$
  for all $c \in \mathbb{C}$ **do**
    $new := new \triangledown \hat{f}_c(new)$
until $new \sqsubseteq old$
return new

Fig. 3. The fixpoint algorithm for our pre-analysis. The analysis uses the widening operation in order to ensure the analysis' termination.

Deriving a further abstraction. Using the access function $\mathcal{A}$, we can estimate accessed locations. Suppose $\hat{X}$ is the analysis results for the original analysis, i.e., $\hat{X} = \mathbf{lfp}(\hat{F})$. Then, because $\mathcal{A}$ is monotone, all the abstract locations that are accessed at c throughout the analysis are captured by $\mathcal{A}(c)(\hat{s})$, where $\hat{s} = \bigsqcup_{c' \hookrightarrow c} \hat{X}(c')$ is the input abstract memory at fixpoint. However, because $\hat{X}$ itself is computed from the original analysis ($\mathbf{lfp}(\hat{F})$), the accessed-locations-estimation phase would take at least the same time as the actual analysis. We have to find the accessed locations in a more efficient way. We do this by computing $\hat{X}'$ that is more approximate than $\hat{X}$, i.e., $\hat{X} \sqsubseteq \hat{X}'$.

We define a pre-analysis that computes such a $\hat{X}'(\sqsupseteq \hat{X})$. To this end, we apply a conservative abstraction to the original analysis. The abstract domain $\mathbb{C} \to \hat{\mathbb{S}}$ and semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$ for the original (actual) analysis was defined as follows (the following is just a repetition, for convenience, of the definition in Section 2.3) :

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}. \bigsqcup_{c' \hookrightarrow c} \hat{f}_{c'}(\hat{X}(c')).$$

We apply a simple abstraction that ignores the order of program statements (flow-insensitivity). The abstract domain is obtained by defining a Galois connection:

$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\mathbb{S}}$$

such that,

$$\alpha = \lambda \hat{X}. \bigsqcup_{c \in \mathbb{C}} \hat{X}(c)$$
$$\gamma = \lambda \hat{s}. \lambda c \in \mathbb{C}. \hat{s}$$

The semantic function $\hat{F}_p : \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is defined as follows:

$$\hat{F}_p = \lambda \hat{s}. \left(\bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s}) \right)$$

The following lemma shows that the pre-analysis is a conservative approximation of the original analysis.

Lemma 5. $\mathbf{lfp}(\hat{F}) \sqsubseteq \gamma(\mathbf{lfp}(\hat{F}_p))$

Proof. See Appendix. □

In general, the pre-analysis can be derived using any conservative abstraction of the original analysis. However, we chose the above abstraction because it is efficient enough in practice and it is precise enough to track reachability among the dynamically allocated locations and structure fields. In our experiments, filtering out not only unused variables but also unused allocated locations and fields was found to be vital for the performance of our localization technique.

Fig. 3 shows the fixpoint algorithm for the pre-analysis. It is based on a flow-insensitive fixpoint computation. The analysis starts with a bottom memory state ($\bot_{\hat{\mathbb{S}}}$). The state is iteratively updated by flow functions for all control points in the program until the resulting state is subsumed by the state of the previous iteration. Let $\hat{s}_{pre}$ be the analysis results from the pre-analysis.

4.2. Actual analysis

The actual analysis is the same as Airac except that now we use the access information ($\mathcal{A}$) and localize memory states. Given an input memory state $\hat{s}$ to a call-point c (such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$), the semantic function $\hat{f}_c$ for the call statement is changed as follows:

$$\hat{f}_c(\hat{s}) = \hat{s}'|_{\mathsf{access}(f)} \quad \text{where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

After the parameter is bound ($\hat{s}'$), the memory state is restricted to the set of accessed locations $\mathsf{access}(f)$ that represents the set of abstract locations that are accessed by procedure f:

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} \left(\bigcup_{c \in \mathsf{control}(g)} \mathcal{A}(c)(\hat{s}_{pre}) \right)$$

where $\mathsf{callees}(f)$ denotes the set of procedures, including f, that are reachable from f via the call-graph and $\mathsf{control}(f)$ the set of control points in procedure f, and $\hat{s}_{pre}$ is the analysis result from the pre-analysis. The following theorem ensures the safety of the localization. (By correctness, we mean that the precomputed access information is conservative. In this article, we assume the sorts of static analyses (designed in Section 2) where abstract locations are fixed and abstract values between different locations are independent of each other. With this particular family of abstractions, from the fact that all the abstract locations that will be accessed in the actual analysis are included in the pre-analysis results, the soundness of localizing analysis is immediate up to the accessed locations. For other styles of analyses, the soundness of localizing analyses was studied in [31,24]).

Theorem 1 (*Safety of Access-based Localization*)**.** *For all procedures f,* $\mathsf{access}(f)$ *conservatively estimates abstract locations that are accessed during the original (non-localized) analysis of f.*

Proof. Abstract location a is accessed inside procedure f if and only if it is accessed either in the body of f or in the bodies of procedures that are called by (reachable via call-graph from) f, which is the definition of access. Moreover, because $\hat{s}_{pre}$ conservatively approximates the abstract memories of all program points (Lemma 5) and $\mathcal{A}$ is monotone (Lemma 3), $\mathcal{A}(n)(\hat{s}_{pre})$ contains all the abstract locations that would be accessed in actual analysis. Thus, access is a safe estimation of accessed locations. □

Access-based localization can be used in combination with the reachability-based approach to localize memory states more aggressively. Given an input memory state $\hat{s}$ to a call point c such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$, reachable locations $\mathcal{R}(f_x, \hat{s})$, and accessed locations $\mathsf{access}(f)$, the semantic function $\hat{f}$ for the call statement $\mathsf{call}(f_x, e)$ is changed as follows:

$$\hat{f}_c(\hat{s}) = (\hat{s}'|_{\mathcal{R}(f_x, \hat{s}')})|_{\mathsf{access}(f)} \quad \text{where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

After parameter binding ($\hat{s}'$) the memory is first restricted to the reachable locations ($\mathcal{R}(f_x, \hat{s}')$) and then the resulting memory is restricted to $\mathsf{access}(f)$. The reason why we restrict the memory to $\mathcal{R}(f_x, \hat{s}') \cap \mathsf{access}(f)$ is that $\mathsf{access}(f)$ may have locations that are unreachable, i.e., not contained in $\mathcal{R}(f_x, \hat{s}')$, because $\hat{s}_{pre}$ is computed by the less precise pre-analysis but $\mathcal{R}(f_x, \hat{s}')$ is computed during the more precise actual analysis. Hence, the memory states localized by the combination of reachability- and access-based approach are never larger than those localized by the reachability-based approach.

Remark. Note that missing entries in the localized memories do not cause problems in practice. For example, suppose there are two global variables g_1 and g_2, and function f uses g_1 only in its body. In the original analysis without localization, the local memory of f involves both values of g_1 and g_2. On the other hand, the analysis with access-based localization does not put the value of g_2 in the local memory of f. Thus, it seems that the localizing analysis cannot make any conclusions regarding g_2 from the analysis results. However, we believe this limitation is not a real problem in practice because, if necessary, the original analysis results can be easily reconstructed from the localized results by looking for the most recent definitions of the values. For example, the missing value for g_2 can be obtained from the program points where g_2 was lastly defined. All we need to do is to compute def-use chains, which is well-known in the literature and is even freely available if the program is in the SSA form.

5. Extension 1: localization for arbitrary code blocks

Generalizing the idea, we can apply the access-based localization technique for code blocks smaller than procedures. In the literature, localizations were conventionally performed only at procedure-level [31,37,15]. It is probably because devising a reachability-based localization technique for arbitrary code blocks is not as simple as the technique for procedures. For procedures, the initial locations for computing reachability are parameters and globals, but, for arbitrary code blocks, the initial locations are not explicitly given in the program text. By contrast, applying the access-based localization for arbitrary code blocks is easier than the reachability-based localization.

Given a code block, it is straightforward to collect accessed locations for the block because our pre-analysis provides access information ($\mathcal{A}$) for each control point in the program. We localize the input memories to the block according to the access information for the block, and analyze the block with the localized memory state, which avoids re-analyses of blocks and speeds up memory operations. We select localization target blocks before starting the actual analysis.

For effectiveness, we have to carefully select blocks to apply localization. Localization improves the analysis performance, but at the same time, introduces a performance overhead. At the entry of a selected block, additional set-operations to localize the input memory state have to be performed and at the exit of the block, non-localized memory portions of the input memory have to be merged with the output of the block. In order to balance against the localization overhead, we

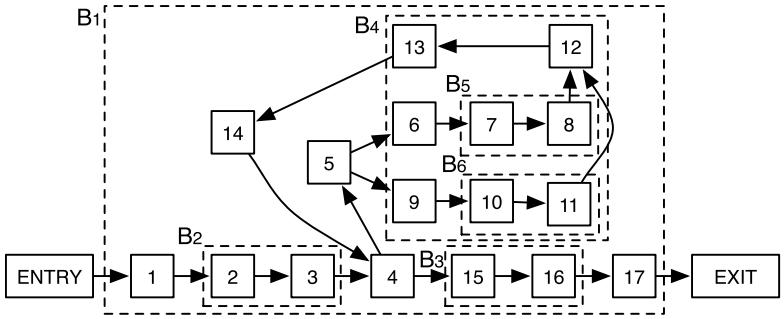


Fig. 4. An example for access-based localization for arbitrary code blocks.

select code blocks $\langle entry, exit, B\rangle$ that consists of one *entry* node, one *exit* node, and a selected block B that satisfies the following properties:

- the *entry* (respectively, the *exit*) node strictly dominates (respectively, post-dominates) all nodes in B, and B contains all nodes that are strictly dominated and post-dominated by the *entry* and *exit*, respectively
- code block size $|B| \geq k$ for parameter k

Using the parameter k, we are able to find a balance between actual reduction and overhead introduced by localizing operations. The above selection strategy is applied recursively: a block satisfying the requirements can be selected inside another selected block.

Example 3. Consider the control flow graph of a procedure in Fig. 4 . The dashed nested boxes are the blocks selected by our algorithm when $k = 2$. First, the entire body of the procedure is selected (B_1). Inside B_1, the algorithm selects B_2, B_3, B_4, B_5 and B_6 recursively. As an example, consider block B_4 whose entry and exit are node 5 and 14, respectively. We localize B_4's input memory (the output memory of node 5) according to the set of abstract locations accessed by B_4 (the set of nodes 6, 7, 8, 9, 10, 11, 12, 13). And the non-localized memory portions at the entry (node 5) are merged with output memory of B_4 at the exit (node 14).

6. Extension 2: access-based localization with bypassing

In this section, we extend the access-based localization to mitigate a substantial inefficiency in handling procedure calls. Though access-based localization is effective in tightly localizing abstract memories, the technique has a limitation in handling procedure calls: the localized input memory for a procedure contains not only memory locations accessed by the procedure but also those accessed by transitively called procedures. This weakness is especially aggravated in the presence of recursive call cycles, which is common in analysis of realistic programs. In this section, we present a technique, called bypassing, to alleviate the problem.

6.1. Overview

Any localization technique, including both access-based and reachability-based techniques, has a source of inefficiency in handling procedure calls. In access-based localization, the localized input state for a procedure involves not only the abstract locations that are accessed by the called procedure but also those locations that are accessed by transitively called procedures. For instance, when procedure f calls g, the localized state for f contains abstract locations that are accessed by g as well as abstract locations accessed by f. Those locations that are exclusively accessed by g are, however, irrelevant to the analysis of f because they are not used in analyzing f. Even so, those locations are involved in the localized state for f, which sometimes leads to unnecessary computational cost (due to re-analyses of procedure bodies).[6]

The inefficiency is exacerbated with recursive call cycles. Consider a recursive call cycle $f \to g \to h \to f \to \cdots$. Because of the cyclic dependence among procedures, every procedure receives input memories that contain all abstract locations accessed by the whole cycle. That is, access-based localization does not work any more inside call cycles. Moreover, recursive cycles (even large ones) are common in real C programs. For example, in GNU open source programs, we noticed that a number of programs have large recursive cycles and a single cycle sometimes contains more than 40 procedures.[7] This was found to be the main performance bottleneck of access-based localization in practice.

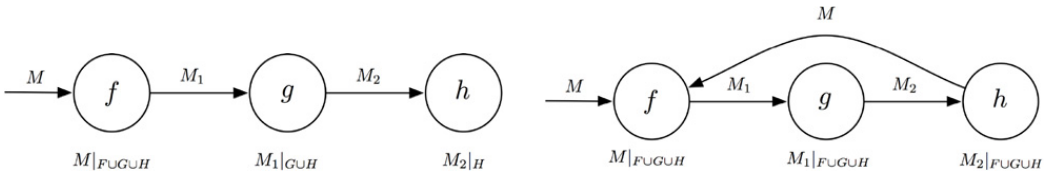
In this section, we extend access-based localization technique to mitigate the aforementioned inefficiency. With our technique, localized states for a procedure contain only the abstract locations that are accessed by the procedure and do not contain other locations that are exclusively accessed by transitively called procedures. Those excluded abstract locations are "bypassed" to the transitively called procedures, instead of passing through the called procedure. In this way, analysis of a

6 In fact, any localization techniques suffers from similar problems. Here, we discuss the problem in the context of access-based localization.

7 One might wonder whether the large recursive call cycles are actual call cycles or spurious ones due to an imprecise analysis of function pointers. For the benchmark programs in Table 5, we have manually checked and found that the call cycles are actual cycles, not spurious ones added by the analysis.

```
int a=0;
void g() { a++; }
void f() { g(); }
int main () {
   a=1; f();       // first call to f
   a=2; f();       // second call to f
}
```

Fig. 5. Example code for illustration of bypassing technique.



(a) Non-recursive call chain. (b) Recursive call cycle.

Fig. 6. Problem of localization. F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. $M|_F$ denotes the memory state M with projected on abstract locations F.

procedure involves only the memory parts that the procedure directly accesses (even inside recursive cycles), which results in tighter localization and hence reduces analysis cost more than access-based localization does. The following example illustrates how our technique saves cost.

Example 4. Consider the code in Fig. 5. Procedure `main` calls `f`, and `f` calls g. Procedure g updates the value of `a`. Procedure `main` calls `f` two times with the value of `a` changed.

- With access-based localization: Both `f` and g are analyzed two times. The localized input memory for `f` at the first call (line 5) contains location `a` because `a` is (indirectly) accessed in `f`. The localized state at the second call (line 6) contains the same location. Because the value of `a` is changed, `f` (as well as g) is re-analyzed at the second call.
- With access-based localization with bypassing: `f` is analyzed only once (though g is analyzed twice). There is no locations that are directly accessed by `f` and hence is not re-analyzed at the second call. However, procedure g is re-analyzed because we propagate the changed value of `a` to the entry of g.

Next, we illustrate how our technique works with examples. Fig. 6 shows example call graphs. There are three procedures: f, g and h. Suppose F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. We describe how the problem occurs and then how to overcome the problem.

Access-based localization has inefficient aspects in analyzing procedure calls. We first consider the case for non-recursive call chains (Fig. 6(a)). With the localization, the input memory M to f is localized so that the procedure f is analyzed only with a subpart $M|_{F\cup G\cup H}$ (M projected on locations set $F \cup G \cup H$) rather than the entire input memory. Similarly, the input memory M_1 to g is localized to $M|_{G\cup H}$, and h's input memory M_2 is localized to $M_2|_H$. The inefficiency comes because the entire localized memory is not accessed by each procedure. For example, abstract locations contained in $G \cup H$ are not necessary in analyzing the body of f.

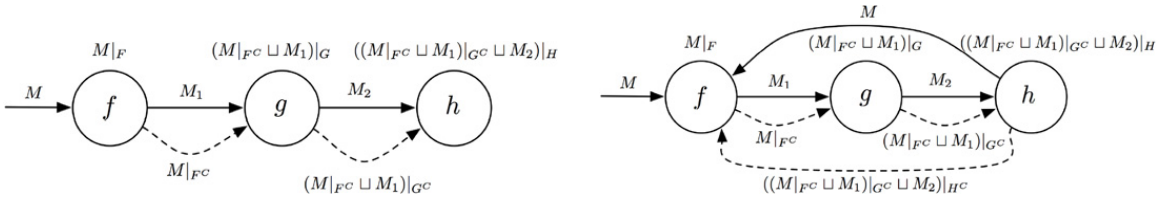
The problem becomes severe when analyzing recursive call cycles. Consider Fig. 6(b). As in the previous case, the input memory M to f is localized to $M|_{F\cup G\cup H}$. However, in this case, the input memory M_1 to g is also projected on $F \cup G \cup H$, not on $G \cup H$, because f can be called from g through the recursive cycle. Similarly, input memory M_2 to h is localized to $M|_{F\cup G\cup H}$. In summary, localization does not work any more inside the cycle.

Fig. 7 illustrates how our technique works. We first consider non-recursive call case (Fig. 7(a)). Instead of restricting f's input memory to $F \cup G \cup H$, we localize it with respect to only the directly accessed locations, i.e., F. Thus, f is analyzed with $M|_F$. The non-localized memory part ($M|_{F^C}$) is directly bypassed (dashed line) to g. Then, the output memory M_1 from f and the bypassed memory $M|_{F^C}$ are joined to prepare input memory $M_1 \sqcup M|_{F^C}$ for procedure g. The input memory is localized to $(M_1 \sqcup M|_{F^C})|_G$ and g is analyzed with the localized memory. Again, the non-localized parts $(M_{F^C} \sqcup M_1)|_{G^C}$ are bypassed to the subsequent procedure h. In this way, each procedure is analyzed only with abstract locations that the procedure directly accesses.

The technique is naturally applicable to recursive cycles (Fig. 7(b)). With our technique, even procedures inside recursive call cycles are analyzed with memory parts that are directly accessed by each procedure. Hence, in Fig. 7(b), the localized memory for f (resp., g and h) only contains locations F (resp., G and H).

6.2. Incorporating bypassing into access-based localization

Before discussion, we introduce some notations. When a procedure f is called from a call-site, we say that f is a directly called procedure from the call-site, and procedures that are reachable from f via the call-graph are indirectly (or transitively) called procedures.



(a) Non-recursive call chain. (b) Recursive call cycle.

Fig. 7. Illustration of our technique. With our technique, each procedure is analyzed with its respective directly accessed locations, and others are bypassed (dashed line) to the subsequent procedure.

Example 5. Consider a call chain $f \to g \to h$. When f is called from a call-site, f is the directly called procedure, and g and h are indirectly called procedures.

When a procedure f is called from a call-site, we say that a location is directly accessed by procedure f if the location is accessed inside the body of f. We say that the location is indirectly accessed by f if the location is not accessed inside f's body but accessed by indirectly called procedures.

Example 6. Consider a call chain $f \to g$, and assume that locations l_1 is accessed inside the body of f and l_2 is accessed inside the body of g. We say l_1 is directly accessed by f and l_2 is indirectly accessed by f (l_2 is directly accessed by g).

We need to modify both pre-analysis and actual analysis. Pre-analysis is slightly changed: pre-analysis is exactly the same as the one used in access-based localization, except that we use its result in a different way. In access-based localization, we compute $\mathsf{access}(f)$, which includes abstract locations directly accessed by f as well as locations indirectly accessed by f. Instead, here, we computes $\mathsf{direct} \in \mathit{ProcId} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$ that maps each procedure to a set of abstract locations that are directly accessed by the procedure, excluding indirectly accessed locations. Given $\mathcal{A} : \mathbb{C} \to \hat{\mathbb{S}} \to \hat{\mathcal{P}}(\hat{\mathbb{L}})$ and the pre-analysis result $\hat{s}_{pre}$, the set $\mathsf{direct}(f)$ is defined as follows:

$$\mathsf{direct}(f) = \bigcup_{c \in \mathsf{control}(f)} \mathcal{A}(c)(\hat{s}_{pre})$$

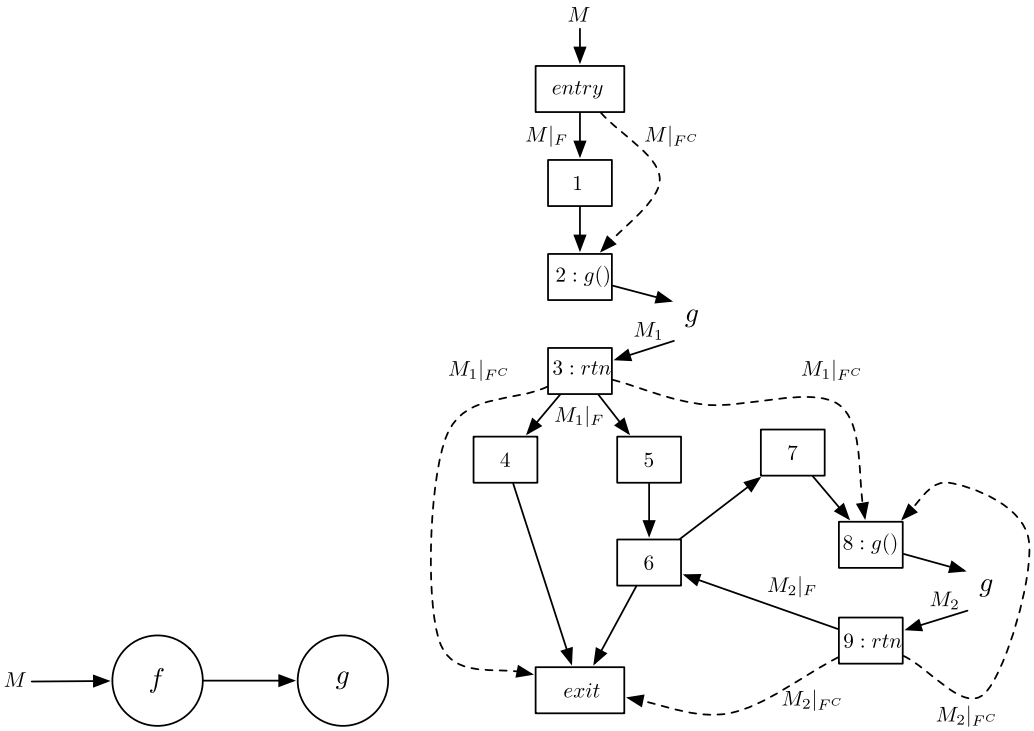
The main changes are in actual analysis. In access-based localization, actual analysis performed localization using the access information from pre-analysis. Now, the actual analysis is changed in two ways: the analysis performs the localization in a different way, and it additionally performs another technique, called bypassing. When analyzing a procedure, we localize the input memory state so that only the abstract locations directly accessed by the procedure are passed to the current procedure. The non-localized parts, which contains indirectly accessed locations, are not passed to the directly called procedure but bypassed to indirectly called procedures. In this way, every procedure is analyzed with input memory state that is more tightly localized than access-based localization. In terms of analysis on control flow graphs, these operations work as follows:

- **Localization**: Localization is performed at control points where memory states come from other procedures. These control points include entry and return points: when a procedure is called from a call-site, the input memory from the call-site flows into entry of the called procedure, and when a procedure returns, the memory state returned from the procedure flows into its caller via a return point. Hence, the memory states at entry and return points of a procedure are localized so that the procedure is analyzed with the directly accessed locations. We call such points, where localization occurs, bypassing sources.

  Example 7. Consider the Fig. 8. Fig. 8(a) shows a call-graph, where procedure f calls g, and Fig. 8(b) shows the control flow graph for f. Let F and G be the set of abstract locations that are directly accessed by procedure f and g, respectively. There are three bypassing sources: entry, 3, and 9. Control points 3 and 9 are return points. At *entry*, the input memory M is restricted to F. Hence, control point 1 is analyzed with the localized memory $M|_F$. At 3 and 9, the memory returned from procedure g, M_1 and M_2 are restricted to the location set F, and hence, the body of procedure f is always analyzed with the local memory $M|_F$. By contrast, with access-based localization, f is analyzed with the localized memory $M|_{F \cup G}$, which is equal or bigger than $M|_F$.

- **Bypassing**: Bypassing happens between bypassing sources and targets. The non-localized parts at bypassing sources (entry or return points) should flow into control points where memory states flow into other procedures. These nodes include procedure exit and calls: at procedure exit, the output memory state of the procedure is propagated to the caller, and at call-sites, memory states flow into called procedures. Thus, after performing localization at a bypassing source, the non-localized parts are bypassed to "immediate" call or exit points that are reachable without passing through other call-sites. We call such calls and exits bypassing targets.

  Example 8. Consider the Fig. 8(b) again. The solid lines represent control flow graphs of procedure f and dashed lines shows how bypassing happens. There are three bypassing sources: entry, 3, and 9. The bypassing target for entry is the call-site 2. Another call-site 8 or exit are not bypassing target for entry because they are not reachable from entry without



(a) Call-graph. (b) Control flow graph of f.

Fig. 8. Example: (a) a call-graph, where f is called with input memory state M and g is called from f (b) inside view (control flow graph) of f, where solid lines represent control flow edges and dashed lines represent bypassing edges.

passing through the call-site 2. The bypassing targets for 3 are 8 and exit. Similarly, bypassing targets for 9 are 8 and exit. At entry, the non-localized memory parts ($M|_{F^C}$) are bypassed to entry's bypassing target, control point 2. Similarly, at 3 and 9, the non-localized memory $M_1|_{F^C}$ and $M_2|_{F^C}$ are bypassed to their bypassing targets, node 8 and exit.

Fig. 9(b) shows our technique integrated in the worklist-based analysis algorithm. In order to incorporate bypassing into access-based localization, only shaded lines are inserted; other parts remain the same. Predicate $\mathsf{bypass\_source} \in \mathbb{C} \to \mathit{bool}$ checks whether a node is a bypass source or not:

$$\mathsf{bypass\_source}(c) = \begin{cases} \mathit{true} & \cdots\ c \text{ is either entry or return} \\ \mathit{false} & \cdots \text{ otherwise} \end{cases}$$

Function $\mathsf{procof} \in \mathbb{C} \to \mathit{ProcId}$ gives name of the procedure that encloses the given node. Function project takes a memory state and a procedure and partitions the input memory into directly accessed and indirectly accessed parts:

$$\mathsf{project}(m, f) = (m|_{\mathsf{direct}(f)}, m|_{\mathsf{access}(f)\setminus\mathsf{direct}(f)})$$

Function $\mathsf{bypass\_target} \in \mathbb{C} \to \mathcal{P}(\mathbb{C})$ maps each bypass source to its bypass targets:

$$\mathsf{bypass\_target}(c) = \{t \mid c \hookrightarrow c_1 \hookrightarrow \cdots \hookrightarrow c_k \hookrightarrow t \ \wedge\ c_i \text{ is not a call} \ \wedge\ t \text{ is either call or exit}\}$$

If the current node n is a bypass source (line 9), the memory state m is divided into a local memory m_l and the rest part m_b (line 10). The local memory m_l is propagated to the successors of n as in the case of the normal algorithm (line 15). The non-localized memory (m_b) is updated in the input memories of bypassing targets of n (line 11–14).

6.3. Delivery points optimization

Bypassing operation induces additional join operations, one of the most expensive operation in semantic-based static analyses [5,20]. At bypassing targets, the bypassed memory from the bypassing source should be joined with the memory propagated along usual control flows. For example, consider Fig. 8. At control point 2, two input memories, one propagated from 1 and another bypassed from *entry*, are joined. Similarly, at the other bypassing targets (point 8 and *exit*), additional join operations take place.

We noticed that the number of additional joins is sometimes unbearable. For example, Fig. 10 shows a common programming pattern: the left-hand shows the code pattern, and the middle shows its control flow graph with bypassing edges (dashed lines). Procedures `f1, f2,..., fk` are sequentially called after respective condition checks (`cond1, cond2,..., condk`). For this code, bypassing happens as follows (as dashed lines in Fig. 10 show):

```
(01) : W ∈ Worklist = 𝒫(ℂ)
(02) : X̂ ∈ ℂ → 𝕊̂
(03) : f̂_c ∈ 𝕊̂ → 𝕊̂

(04) : W := ℂ; X̂ := λc.⊥
(05) : repeat
(06) :   c := choose(W)
(07) :   W := W − {c}
(08) :   ŝ := ⨆_{c'↪c} f̂_{c'}(X̂(c'))
(16) :   if ŝ ⋤ X̂(c)
(17) :     if c is a head of a flow cycle
(18) :       ŝ := X̂(c)▽ŝ
(19) :     X̂(c') := ŝ
(20) :     W := W ∪ {c' | c ↪ c'}
(21) : until W = ∅
```

(a) The worklist-based algorithm.

```
(01) : W ∈ Worklist = 𝒫(ℂ)
(02) : X̂ ∈ ℂ → 𝕊̂
(03) : f̂_c ∈ 𝕊̂ → 𝕊̂

(04) : W := ℂ; X̂ := λc.⊥
(05) : repeat
(06) :   c := choose(W)
(07) :   W := W − {c}
(08) :   ŝ := ⨆_{c'↪c} f̂_{c'}(X̂(c'))
(09) :   if bypass_source(c) then
(10) :     (ŝ_l, ŝ_b) := project(f̂_c(ŝ), procof(c))
(11) :     for all t ∈ bypass_target(c) do
(12) :       if ŝ_b ⋤ X̂(t)
(13) :         X̂(t) := X̂(t) ⊔ ŝ_b
(14) :         W := W ∪ {t}
(15) :     ŝ := ŝ_l
(16) :   if ŝ ⋤ X̂(c)
(17) :     if c is a head of a flow cycle
(18) :       ŝ := X̂(c)▽ŝ
(19) :     X̂(c') := ŝ
(20) :     W := W ∪ {c' | c ↪ c'}
(21) : until W = ∅
```

(b) The algorithm with bypassing.

Fig. 9. Comparison of the normal analysis algorithm and our bypassing algorithm: bypassing is simply incorporated to the traditional algorithm.

- From *entry* to $f_1, f_2, f_3, \ldots, f_k$, *exit*
- From f_1 to $f_2, f_3, \ldots, f_k$, *exit*
- $\cdots$
- From f_k to *exit*

Thus, the total number of bypassing edges for this code fragment is $(k+1)(k+2)/2$ when k is the number of branches.

We mitigate the overhead by making bypassing pass through some particular points that reduce the total number of bypassing edges. These points, we call them "delivery points", include some join points and loop heads. For example, in Fig. 10, we use $\{1, 2, \ldots, k-1\}$ as delivery points. As a result, bypassing happens as shown in the rightmost graph in Fig. 10. Bypassing from *entry* to $call_1$ takes place as before, but instead of bypassing from *entry* to $\{f_2, \ldots, f_k, exit\}$, we pass through points $1, 2, \ldots, k-1$, which reduces the total number of bypassing edges from $(k+1)(k+2)/2$ to $3k$. In order to select such delivery points, we use a simple heuristic that uses join points or loop heads as delivery points when the selection actually reduces the total number of bypassing edges. This optimization does not degrade the analysis precision because the set of abstract locations on which join operations are performed are not greater than that in the original method.

7. Extension 3: localizing relational analysis

So far, we described the access-based localization for non-relational analyses. In this section, we show how to apply the technique to some relational analyses as well.

Relational analyses are more precise than non-relational analyses. While non-relational analysis independently tracks values of each variable, relational analysis additionally considers relations among the variables. For example, suppose we analyze statement `x:=y` and `y` has value $[1, 2]$ in intervals before the statement. After analyzing the statement non-relationally, the only information we know is that `x` also has interval value $[1, 2]$; thus, we cannot be sure whether the value of `x` is always the same as `y`. By contrast, if we analyze the statement relationally, we get additional constraints that `x` has the same value as `y` after the statement.

Practical relational analyses exploit *packed* relationality [13,25,34,5], and access-based localization is applicable in this case. It is well-known that global relational analyses, which keep the relations of all variables, has unacceptable costs in practice [13,25]. For example, the octagon domain [25], the most popular relational domain, has time complexity $O(n^3)$, where n is the number of program variables. Thus, instead of considering the global relations, practical relational analyses divide variables into *packs* and keep each pack's relations separately. We denote a pack of variables $x_1, \ldots, x_n$ as $\langle\langle x_1, \ldots, x_n \rangle\rangle$.

Example 9. Suppose $\{a, b, c\}$ are program variables and we want to keep track of relations between variables a and b; we are not interested in other relations. In this case, we can divide the variable set into the set of variable packs $\{\langle\langle a, b\rangle\rangle, \langle\langle c\rangle\rangle\}$. Thus, in packed relational analysis using these packs, variables a and b are related together but they are not related with variable c.

7.1. Localizing packed relational analysis

We now define a simple relational analysis and show how to apply the access-based localization to the analysis. The distinguishing feature of localization for packed relational analysis is that the entities that are accessed are defined in terms of variable packs. For example, at a simple statement $x := 1$, all the variable packs that contain x may be accessed.

Language. As in Section 2.1, a program is a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ and commands are associated with control points. We consider the following commands.

$$\begin{array}{rcl} cmd & \to & \mathsf{assign}(x, e) \mid \mathsf{assume}(x < n) \mid \mathsf{call}f \\ e & \to & n \mid x \mid e + e \end{array}$$

The language is simple, pointer-free, and argument-free; however, extending the language to include other language features does not require novelty but verbosity. In this section, we focus only on the key differences between non-relational and relational access-based localizations.

Abstract domain. As before, the abstract domain is a function domain $\mathbb{C} \to \hat{\mathbb{S}}$, mapping abstract states to each control point. In packed relational analyses, the abstract state is defined as follows:

$$\hat{\mathbb{S}} = \mathit{Packs} \to \hat{\mathbb{R}}$$

Abstract states ($\hat{\mathbb{S}}$) maps variable packs ($\mathit{Packs} \subseteq \mathcal{P}(\mathit{Var})$ such that $\bigcup \mathit{Packs} = \mathit{Var}$) to a relational domain ($\hat{\mathbb{R}}$). We assume Packs is given by the user or a pre-analysis [25]. The relational domain ($\hat{\mathbb{R}}$) expresses numerical constraints among variables in the corresponding pack. Examples of numerical constraints are the domain of octagons [25] or polyhedrons [10].

Abstract semantics. Because packed relational analysis does not track all the possible relations between variables, we sometimes need to know actual values (such as ranges) of variables. For example, suppose we analyze $a := b$ with $\mathit{Packs} = \{\langle\langle a, c\rangle\rangle, \langle\langle b, c\rangle\rangle\}$. Analyzing the statement, we try to update the abstract value for pack $\langle\langle a, c\rangle\rangle$. However, because variable b is not contained in the pack, we must obtain the value of b from the other pack $\langle\langle b, c\rangle\rangle$. The value for b is obtained by projecting the relational domain elements for $\langle\langle b, c\rangle\rangle$ into a non-relational domain, such as intervals. To this end, we transform the original program into an internal form with such variables replaced by their actual values. Suppose the actual value of b in terms of intervals is $[1, 2]$ then code $\mathsf{assign}(a, b)$ is transformed into internal code $a := [1, 2]$, where variable b is replaced with its interval value. Formally, we assume abstract semantic function $\mathcal{R} \in cmd^r \to \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ for relational domain $\hat{\mathbb{R}}$ is defined over the following internal language:

$$\begin{array}{rcl} cmd^r & \to & \mathsf{assign}(x, e^r) \mid \mathsf{assume}(x < \hat{\mathbb{Z}}) \mid \mathsf{call}f \\ e^r & \to & \hat{\mathbb{Z}} \mid x \mid e^r + e^r \end{array}$$

where $\hat{\mathbb{Z}}$ is a (non-relational) abstract domain for integers ($2^{\mathbb{Z}} \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \hat{\mathbb{Z}}$) such as intervals.

We now define the semantics of the packed relational analysis. The abstract semantics is defined by the least fixpoint of semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$:

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}. \bigsqcup_{c' \hookrightarrow c} \hat{f}_{c'}(\hat{X}(c')).$$

The abstract semantic function $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is defined as follows:

$$\hat{f}_c(\hat{s}) = \begin{cases} \hat{s} & \cdots \ \mathsf{cmd}(c) = \mathsf{call}f \\ \hat{s}[p_1 \mapsto \mathcal{R}(cmd_1)(\hat{s}(p_1)), \ldots, p_k \mapsto \mathcal{R}(cmd_k)(\hat{s}(p_k))] & \ldots \ \text{otherwise} \end{cases}$$

where

$$\begin{array}{rcl} \{p_1, \ldots, p_k\} & = & \mathsf{pack}(x) \\ \mathsf{pack}(x) & = & \{p \in \mathit{Packs} \mid x \in p\} \\ cmd_i & = & \mathscr{T}(p_i)(\hat{s})(\mathsf{cmd}(c)) \end{array}$$

For both $\mathsf{assign}(x, e)$ and $\mathsf{assume}(x < n)$, we update only the packs that include x. $\mathscr{T}$ transforms cmd into cmd^r: given a variable pack p, state $\hat{s}$, and command cmd, $\mathscr{T}(p)(\hat{s}) \in cmd \to cmd^r$ returns the transformed command.

$$\begin{array}{rcl} \mathscr{T}(p)(\hat{s})(\mathsf{assign}(x, e)) & = & \mathsf{assign}(x, \mathscr{T}_e(p)(\hat{s})(e)) \\ \mathscr{T}(p)(\hat{s})(\mathsf{assume}(x < n)) & = & \mathsf{assume}(x < \mathscr{T}_e(p)(\hat{s})(n)) \end{array}$$

```
if (cond1) f1 ();
if (cond2) f2 ();
...
if (condk) fk ();
```

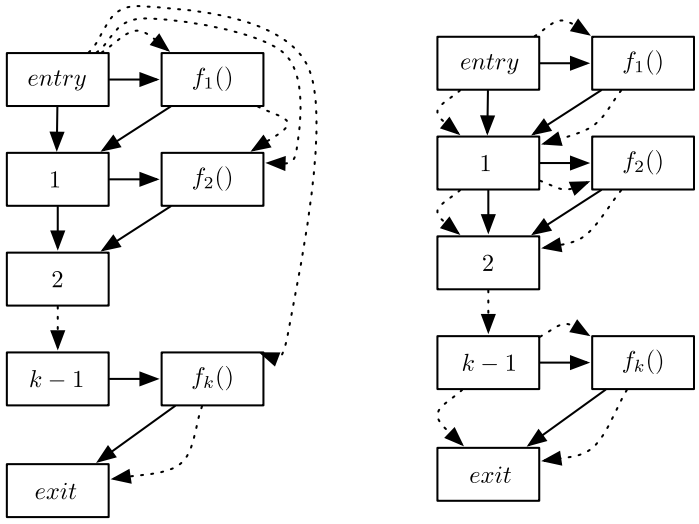


Fig. 10. Example of common code patterns that increases bypassing overhead. In this figure, for simplicity, we merged the call and return nodes into one block.

where $\mathscr{T}_e(p)(\hat{s}) \in e \to e^r$ transforms expressions:

$$\begin{aligned}\mathscr{T}_e(p)(\hat{s})(n) &= \alpha_{\mathbb{Z}}(\{n\})\\ \mathscr{T}_e(p)(\hat{s})(x) &= \begin{cases} x & \text{if } x \in p\\ \pi_x(\hat{s}) & \text{otherwise}\end{cases}\\ \mathscr{T}_e(p)(\hat{s})(e_1 + e_2) &= \mathscr{T}_e(p)(\hat{s})(e_1) + \mathscr{T}_e(p)(\hat{s})(e_2)\end{aligned}$$

where $\pi_x \in \hat{\mathbb{S}} \to \hat{\mathbb{Z}}$ is a user-defined function that projects a relational domain element onto variable x to obtain its abstract integer value. To be safe, π_x should satisfy the following soundness condition:

$$\forall \hat{s} \in \hat{\mathbb{S}}.\pi_x(\hat{s}) \sqsupseteq \alpha_{\hat{\mathbb{Z}}}\left(\bigcap_{p \in \mathsf{pack}(x)} \{r(x) \mid r \in \gamma_{\hat{\mathbb{R}}}(\hat{s}(p))\}\right)$$

Abstract semantics with access-based localization. Note that the abstract semantic function $\hat{f}_c$ given in Section 7.1 propagates the input state for function calls to the called procedures as it is; currently, $\hat{f}_c$ does not localize abstract states. We modify $\hat{f}_c$ to perform access-based localization at function calls.

We first define access function $\mathcal{A} \in \mathbb{C} \to \mathcal{P}(\mathit{Packs})$. As in non-relational analysis (Section 4.1), $\mathcal{A}$ is also naturally derived from the abstract semantic function $\hat{f}_c$:

$$\mathcal{A}(c) = \begin{cases} \emptyset & \cdots \ \mathsf{cmd}(c) = \mathsf{call}f\\ \mathsf{pack}(x) \cup \bigcup_{l \in \mathcal{V}(e)} \mathsf{pack}(l) & \ldots \ \text{otherwise}\end{cases}$$

where $\mathcal{V}(e)$ denotes the set of variables that appear inside expression e. When the command is a call statement, we see that no locations are accessed from the definition of $\hat{f}_c$. When the command is an assignment or an assert statement, variable packs that contain x and those appearing inside expression e are accessed. Unlike the previous access function presented in Section 4.1, the access function for our relational analysis does not depend on states, because the language does not contain pointers. With pointers, the access function takes states and the access information is computed from a pre-analysis, as described before in Section 4.1.

Now, we can incorporate the localization into $\hat{f}_c$. When calling a procedure, the semantic function is changed as follows:

$$\hat{f}_c(\hat{s}) = \hat{s}|_{\mathsf{access}(f)}$$

where $\mathsf{access}(f)$ represents the set of abstract locations that are accessed by procedure f:

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} \left(\bigcup_{c \in \mathsf{control}(g)} \mathcal{A}(c)\right)$$

Note that the localization is very similar to the non-relational case (presented in Section 4.2). The only difference is that the accessed entities are now variable packs rather individual abstract locations.

Table 2
Properties of the benchmarks and analysis results for Airac. Lines of code (**LOC**) are given before preprocessing. The number of procedures (**Proc**), and basic blocks in the program (**Blocks**) are given after preprocessing. Entries with ∞ mean missing data because the analysis runs out of memory.

Program	**LOC**	**Proc**	**Blocks**	Airac (w.o. localization)	
				time (s)	*MB*
spell-1.0	2,213	31	782	46.1	29
barcode-0.96	4,460	57	2,634	105.7	291
httptunnel-3.3	6,174	110	2,757	2808.9	284
gzip-1.2.4a	7,327	135	6,271	12,756.2	886
jwhois-3.0.1	9,344	73	5,147	3,424.5	633
parser	10,900	325	9,298	196,318.8	2,917
bc-1.06	13,093	134	4,924	87,988.5	767
twolf	19,700	222	14,610	∞	∞
tar-1.13	20,258	222	10,800	157,545.0	2916
less-382	23,822	382	10,056	148,015.7	2445
make-3.76.1	27,304	191	11,061	126,908.8	2757
wget-1.9	35,018	434	16,544	∞	∞
screen-4.0.2	44,734	589	31,792	∞	∞
bison-2.4	56,361	1203	20,781	∞	∞
bash-2.05a	105,174	959	28,675	∞	∞

8. Experiments

In this section, we evaluate the performance of the techniques presented in this paper. In Section 8.1, we compare the performance of the procedure-level access-based localization (presented in Section 4) against the reachability-based localization (presented in Section 3). In Sections 8.2 and 8.3, we evaluate the block-level localization and bypassing as well. In Section 8.4, the access-based localization is evaluated for a relational analysis.

We implemented those localization techniques on top of Airac, an interval domain-based abstract interpretation engine in an industrialized bug-finding analyzer [20,19,26,28,27,29].

8.1. Performance of procedure-level access-based localization

From our baseline analyzer Airac, which does not use any localization technique, we have made two analyzers $\text{Airac}_{\text{Reach}}$ and $\text{Airac}_{\text{ProcAcc}}$ that respectively use reachability-based and access-based localization techniques. Those analyzers differ from Airac only in their respective localization schemes. Hence, performance differences, if any, are solely attributed to the different localization methods. The analyzers are written in OCaml.

We have analyzed 15 software packages. Table 2 shows our benchmark programs. `parser` and `twolf` are from SPEC2000 benchmarks, and the others from GNU open-source programs. The entire program is analyzed starting from the `main` procedure. All experiments were done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory. We use two performance measures: (1) *time* is the CPU time spent in seconds; (2) *MB* is the peak memory consumption in megabytes.

We compare the performance of Airac, $\text{Airac}_{\text{Reach}}$, and $\text{Airac}_{\text{ProcAcc}}$. The analysis results for Airac is shown in Table 2. The results for $\text{Airac}_{\text{Reach}}$ and $\text{Airac}_{\text{ProcAcc}}$ are shown in Table 3.

Airac vs. $\text{Airac}_{\text{Reach}}$. The results show that the reachability-based localization reduces the analysis time and memory for most programs. $\text{Airac}_{\text{Reach}}$ consistently reduces the analysis time by 36.7% on average. The effectiveness is clear from the fact that $\text{Airac}_{\text{Reach}}$ reduces analysis time of Airac more than 50% for programs `httptunnel`, `gzip`, `jwhois`, `parser` and `bc`. And the peak memory consumption is reduced by on average 46.5%, enabling $\text{Airac}_{\text{Reach}}$ to analyze `twolf` that cannot be analyzed by Airac because of memory cost.

However, the results also show some limitations of the reachability-based localization. First, for two programs (`spell` and `make`), $\text{Airac}_{\text{Reach}}$ took more time than Airac. This is mainly because of the overhead of localizing operations at procedure calls. Second, it cannot analyze the largest four programs (`wget`, `screen`, `bison`, `bash`) because the analysis runs out of memory. Yang et al. [37,36] report similar observations that shape analysis with reachability-based localization is insufficient for practical performance, and another technique (in their case, a new join operator) is required for more scalability.

$\text{Airac}_{\text{Reach}}$ vs. $\text{Airac}_{\text{ProcAcc}}$. Overall, $\text{Airac}_{\text{ProcAcc}}$ saved 78.5%–98.5%, on average 92.1%, of the analysis time of $\text{Airac}_{\text{Reach}}$. For example, for `bc`, $\text{Airac}_{\text{Reach}}$ took 13879.2 s but $\text{Airac}_{\text{ProcAcc}}$ took 730.9 s, reducing the analysis time by 94.7%. The analysis time of $\text{Airac}_{\text{ProcAcc}}$ includes pre-analysis time. The pre-analysis takes just a small portion of the total analysis time. For example, for `bc`, the total analysis took 730.9 s and pre-analysis just took 8.5 s.

The big time savings are thanks to the synergy between reduction in the number of fixpoint iterations and improved speed of memory operations. First, in our experiments, $\text{Airac}_{\text{ProcAcc}}$ reduced the number of fixpoint iterations by on average 74.3%: more general (weaker) procedural summaries computed by $\text{Airac}_{\text{ProcAcc}}$ let the analyzer avoid re-computation of procedure bodies at different call-sites than $\text{Airac}_{\text{Reach}}$. Second, $\text{Airac}_{\text{ProcAcc}}$ improves speed (#iteration/time) by on average 4.0 times: each procedure is analyzed with smaller memory states.

During the experiments, we observed that the improved performance is quite sensitive to the localization ratio (the size of accessed memory entries/the size of the reachable memory). For example, in Table 3, we see that, analyzing `screen`, $\text{Airac}_{\text{ProcAcc}}$ takes much more time than other programs. This is because `screen` has a bigger localization ratio (on average 38.5%) than other programs (e.g., the ratio for `bash` was 4.6%). So, we believe that localizing the input memory as much as we can is important for saving analysis time. This is why we designed a pre-analysis that considers dynamically allocated locations and structure fields as well as global variables (Section 4).

Moreover, our technique noticeably saves peak memory consumption by on average 71.2%. As an example, for `bc`, $\text{Airac}_{\text{Reach}}$ required 335 MB of peak memory but $\text{Airac}_{\text{ProcAcc}}$ required 106 MB. The reduction enabled $\text{Airac}_{\text{ProcAcc}}$ to analyze the largest four programs (`wget`, `screen`, `bison`, `bash`) that cannot be analyzed by $\text{Airac}_{\text{Reach}}$.

Discussion on analysis precision. $\text{Airac}_{\text{ProcAcc}}$ is at least as precise as $\text{Airac}_{\text{Reach}}$. In principle, more aggressive localization improves the precision of our analysis because unnecessary memory entries are not passed to procedures, avoiding needless value propagations. In order to simply compare the precision, we first joined all the memories, ignoring flow-sensitivity, associated with each program point and then counted the number of constant intervals (*#const*), finite intervals (*#finite*), intervals with one infinity (*#open*), and intervals with two infinities (*#top*) contained in the joined memory. The constant interval and top interval indicate the most precise and imprecise values, respectively. The following table shows the results for some programs. (This result is just to show that our localization technique does not sacrifice (in fact, it increases) analysis precision, *not* to show how accurate our analyzer is. For simplicity of the comparison, we turned off many precision-improving techniques: flow-sensitivity, context pruning, and narrowing.)

Program	Analysis	*#const*	*#finite*	*#open*	*#top*
spell-1.0	$\text{Airac}_{\text{Reach}}$	665	101	33	142
	$\text{Airac}_{\text{ProcAcc}}$	665	102	32	142
barcode-0.96	$\text{Airac}_{\text{Reach}}$	2343	595	221	515
	$\text{Airac}_{\text{ProcAcc}}$	2347	597	218	512

Why does the precision improve? Consider analyzing the following code with interval domain and context-insensitive handling of procedure calls.

```
1: int g = 0;
2: void f () { ... } // assume g is not used inside f
3: void main() {
4:  g = 0; f();  // first call to f
5:  g = 1; f();  // second call to f
6:}
```

Without localization or even with reachability-based localization, the value of g at line 6 is [0, 1] in interval. This is because the analysis is context-insensitive and the values of g from two call-sites are merged at the entry of procedure `f` and the merged value is propagated back to both call-sites.

With access-based localization, the value of g at line 6 is [1, 1] in interval. Because global variable g is not used inside procedure `f`, the input memory states for procedure `f` does not contain the value for g. Hence, the value [0, 0] from the first call is not propagated to line 6 via the interprocedurally invalid path.

8.2. Performance of block-level access-based localization

We made $\text{Airac}_{\text{GenAcc}}$ from $\text{Airac}_{\text{ProcAcc}}$, which further applies the access-based localization to arbitrary code blocks inside procedures. Note that $\text{Airac}_{\text{GenAcc}}$ subsumes $\text{Airac}_{\text{ProcAcc}}$: $\text{Airac}_{\text{GenAcc}}$ implements procedure-level localization as well as block-level localization. We set the minimum block size k to 6 for $\text{Airac}_{\text{GenAcc}}$, which was shown to be most efficient in our setting.

Table 4 and Fig. 11 presents the performance of localizing arbitrary code blocks. In the experiments, the block-level localization ($\text{Airac}_{\text{GenAcc}}$) additionally saved 8.5%–53.7%, on average 31.8%, of the analysis time of the procedure-level localization ($\text{Airac}_{\text{ProcAcc}}$). For example, for `bash`, $\text{Airac}_{\text{ProcAcc}}$ took 2011.3 s but $\text{Airac}_{\text{GenAcc}}$ took 1142.7 reducing the analysis time by 43.2%. Memory costs between them is nearly the same: $\text{Airac}_{\text{GenAcc}}$ reduced peak memory consumption by on average 1.9%. Memory costs for $\text{Airac}_{\text{GenAcc}}$ sometimes increase (e.g., `parser`), because we cache accessed-locations sets for each localization block. The pre-analysis time for $\text{Airac}_{\text{GenAcc}}$ increases because of additional computation selecting localization blocks.

Table 3
Comparison between $\text{Airac}_{\text{Reach}}$ and $\text{Airac}_{\text{ProcAcc}}$. Analysis time for $\text{Airac}_{\text{ProcAcc}}$ includes pre-analysis time. The pre-analysis time is shown inside parentheses. $Save_1$ shows time savings of $\text{Airac}_{\text{Reach}}$ against Airac. $Save_2$ shows time savings of $\text{Airac}_{\text{ProcAcc}}$ against $\text{Airac}_{\text{Reach}}$. *MB* is peak memory consumption in megabytes. Entries with ∞ mean missing data because the analysis runs out of memory.

Program	**LOC**	$\text{Airac}_{\text{Reach}}$			$\text{Airac}_{\text{ProcAcc}}$			
		time (s)	*MB*	$Save_1$	*time*:total(pre)		*MB*	$Save_2$
spell-1.0	2,213	53.0	23	−15.0%	2.4	(0.2)	5	**95.4%**
barcode-0.96	4,460	92.6	125	12.4%	9.4	(0.6)	25	**89.8%**
httptunnel-3.3	6,174	1383.2	154	50.8%	31.4	(1.3)	36	**97.7%**
gzip-1.2.4a	7,327	2,866.6	333	77.5%	94.8	(1.3)	73	**96.7%**
jwhois-3.0.1	9,344	1,185.4	254	65.4%	94.8	(1.3)	73	**96.7%**
parser	10,900	60,577.8	1048	69.1%	890.0	(3.8)	224	**98.5%**
bc-1.06	13,093	13,879.2	335	84.2%	730.9	(4.1)	106	**94.7%**
twolf	19,700	27,230.3	1199	N/A	1,037.7	(7.5)	332	**96.2%**
tar-1.13	20,258	113,061.4	1797	28.2%	2,524.0	(6.0)	338	**97.8%**
less-382	23,822	137,827.3	1480	6.9%	26,817.6	(40.7)	466	**80.5%**
make-3.76.1	23,304	142,325.6	1954	−12.1%	19,015.2	(39.4)	580	**86.6%**
wget-1.9	35,018	∞	∞	N/A	6,735.8	(20.8)	609	N/A
screen-4.0.2	44,734	∞	∞	N/A	340,849.0	(281.5)	2458	N/A
bison-2.4	56,361	∞	∞	N/A	2,487.3	(13.1)	301	N/A
bash-2.05a	105,174	∞	∞	N/A	2,011.3	(20.2)	439	N/A

Table 4
Performance results for localizing arbitrary code blocks ($\text{Airac}_{\text{GenAcc}}$). Analysis time is the total time that includes pre-analysis time. The pre-analysis time is shown inside parentheses. *Save* shows time savings of $\text{Airac}_{\text{GenAcc}}$ against $\text{Airac}_{\text{ProcAcc}}$. *MB* is peak memory consumption in megabytes. Entries with ∞ mean missing data because the analysis runs out of memory.

Program	**LOC**	$\text{Airac}_{\text{ProcAcc}}$			$\text{Airac}_{\text{GenAcc}}(k = 6)$			
		time:total(pre)		*MB*	*time*:total(pre)		*MB*	*Save*
spell-1.0	2,213	2.4	(0.2)	5	2.1	(0.3)	5	**13.5%**
barcode-0.96	4,460	9.4	(0.6)	25	5.5	(1.4)	21	**41.6%**
httptunnel-3.3	6,174	31.4	(1.3)	36	21.6	(2.4)	24	**31.0%**
gzip-1.2.4a	7,327	94.8	(1.3)	73	54.5	(8.0)	67	**42.5%**
jwhois-3.0.1	9,344	254.8	(1.2)	170	188.1	(18.5)	148	**26.2%**
parser	10,900	890.0	(3.8)	224	617.0	(9.2)	245	**30.7%**
bc-1.06	13,093	730.9	(4.1)	106	542.7	(8.5)	116	**25.8%**
twolf	19,700	1,037.7	(7.5)	332	480.4	(20.0)	260	**53.7%**
tar-1.13	20,258	2,524.0	(6.0)	338	1,638.3	(15.9)	351	**35.1%**
less-382	23,822	26,817.6	(40.7)	466	18,766.7	(95.0)	528	**30.0%**
make-3.76.1	27,304	19,015.2	(39.4)	580	17,405.7	(75.9)	740	**8.5%**
wget-1.9	35,018	6,735.8	(20.8)	609	3,823.3	(48.5)	623	**43.2%**
screen-4.0.2	44,734	340,849.0	(281.5)	2458	274,280.3	(667.1)	2958	**19.5%**
bison-2.4	56,361	2,487.3	(13.1)	301	1,696.6	(37.7)	302	**31.8%**
bash-2.05a	105,174	2,011.3	(20.2)	439	1,142.7	(59.5)	416	**43.2%**

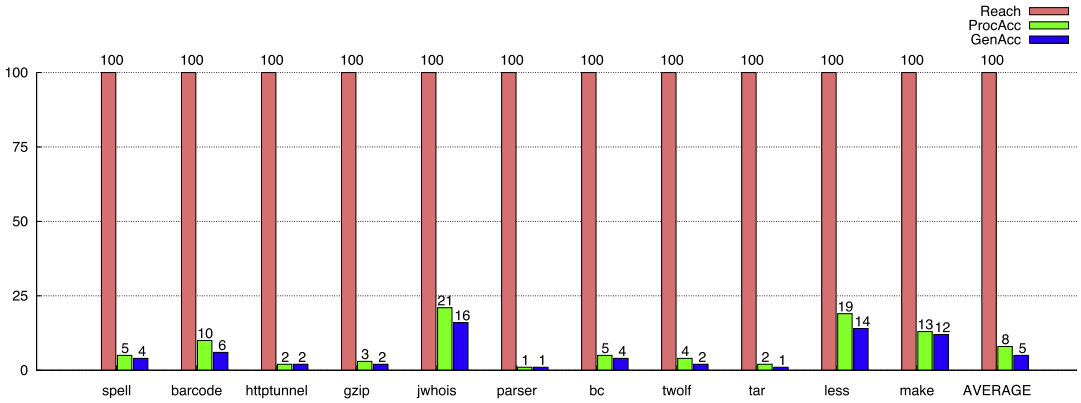


Fig. 11. Performance comparison of reachability-based localization ($\text{Airac}_{\text{Reach}}$), procedure-level access-based localization ($\text{Airac}_{\text{ProcAcc}}$), and block-level access-based localization ($\text{Airac}_{\text{GenAcc}}$). Compared to the reachability-based localization, our access-based procedural localization technique reduces analysis time by 80.5%–98.5%, on average 92.1%. The generalized, access-based localization for arbitrary code blocks further reduces the analysis time by 8.5%–53.7%, on average 31.8%, against the procedure-level access-based localization.

Table 5
Performance of bypassing techniques. Lines of code (**LOC**) are given before preprocessing. The number of procedures (**Proc**) is given after preprocessing. **LRC** represents the size of largest recursive call cycle contained in each program. *time* shows analysis time in seconds. *MB* shows peak memory consumption in megabytes. $\text{Airac}_{\text{ProcAcc}}$ uses access-based localization for procedure calls and $\text{Airac}_{\text{Bypass}}$ uses our technique. *time* for $\text{Airac}_{\text{ProcAcc}}$ and $\text{Airac}_{\text{Bypass}}$ are the total time that includes pre-analysis time. *Save* shows time savings in percentage of $\text{Airac}_{\text{Bypass}}$ against $\text{Airac}_{\text{ProcAcc}}$. The performance numbers of $\text{Airac}_{\text{ProcAcc}}$ in this table are little bit different from those in Tables 3 and 4, because the analyzer had been more tuned (in a way orthogonal to localization) since the time when experiments in Tables 3 and 4 were conducted.

Program	**LOC**	**Proc**	**LRC**	$\text{Airac}_{\text{ProcAcc}}$		$\text{Airac}_{\text{Bypass}}$		*Save*
				time (s)	*MB*	*time* (s)	*MB*	(*time*)
spell-1.0	2,213	31	0	2.4	10	1.6	10	**31.6%**
gzip-1.2.4a	7,327	135	2	51.9	65	37.7	64	**27.4%**
parser	10,900	325	3	571.6	206	319.4	245	**44.1%**
bc-1.06	13,093	134	1	496.9	131	318.4	165	**35.9%**
twolf	19,700	192	1	509.5	212	389.9	212	**23.5%**
tar-1.13	20,258	222	13	2,407.9	294	1,503.2	338	**37.6%**
less-382	23,822	382	46	14,720.8	490	4,906.4	427	**66.7%**
make-3.76.1	27,304	191	61	14,681.9	695	5,248.0	549	**64.3%**
wget-1.9	35,018	434	13	6,717.5	544	4,383.4	552	**34.7%**
screen-4.0.2	44,734	589	77	310,788.0	2228	66,920.6	1875	**78.5%**
bash-2.05a	105,174	959	4	1,637.6	272	1,492.4	265	**8.9%**

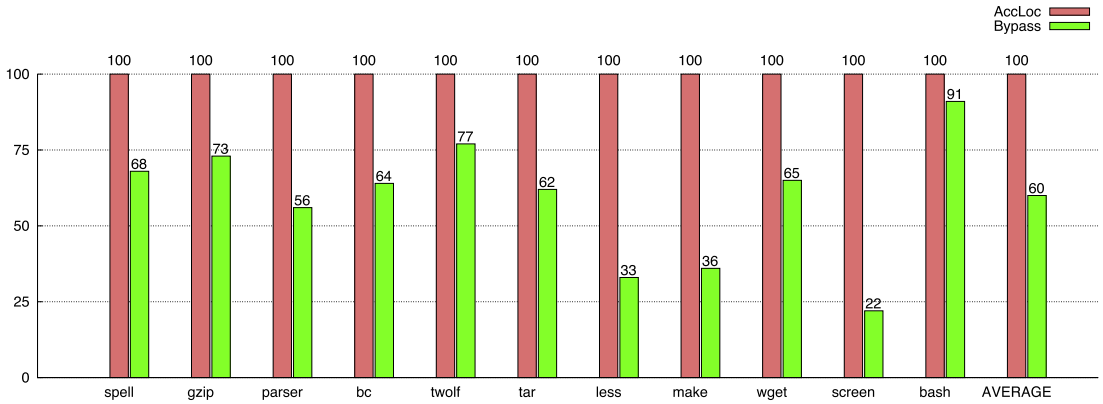


Fig. 12. Performance comparison between access-based localization and bypassing. Extending access-based localization with bypassing saves analysis time by 9%–79%, on average 42%. In particular, bypassing technique is more effective for benchmark programs, such as make and screen, that extensively use recursion and have large recursive call cycles. For those programs, the average savings are 77%.

8.3. Performance of bypassing

From $\text{Airac}_{\text{ProcAcc}}$, we have made $\text{Airac}_{\text{Bypass}}$ that uses the access-based localization with the bypassing technique. $\text{Airac}_{\text{Bypass}}$ is exactly the same as $\text{Airac}_{\text{ProcAcc}}$ except that $\text{Airac}_{\text{Bypass}}$ additionally performs the bypassing operation. Hence, performance differences, if any, between them, are solely attributed to the bypassing technique.

Fig. 12 and Table 5 compares the *time* of $\text{Airac}_{\text{ProcAcc}}$ and $\text{Airac}_{\text{Bypass}}$. Overall, $\text{Airac}_{\text{Bypass}}$ saved 8.9%–78.5%, on average 42.1%, of the analysis time of $\text{Airac}_{\text{ProcAcc}}$. There are some noteworthy points.

- Some programs contain large recursive call cycles. One common belief for C programs is that it does not largely use recursion in practice. However, for the benchmark programs in Table 5, we have found that some programs extensively use recursion and large recursive cycles unexpectedly exist in a number of real C programs. Note that these call cycles are actual cycles (we have manually checked this), not spurious ones caused by an imprecise analysis of function pointers. For example, from Table 5, note that program less, make, and screen have recursive cycles (scc) that contain more than 40 procedures.
- $\text{Airac}_{\text{ProcAcc}}$ is extremely inefficient for those programs. For other programs that have small (or no) recursive cycles, the analysis with access-based localization is quite efficient. For example, analyzing bash (the largest one in our benchmark) takes 1637 s. However, analyzing those programs that have large recursive cycles takes much more time: less and make take more than 10,000 s and screen takes more than 310,000 s to finish the analysis, even though they are not the largest programs.
- $\text{Airac}_{\text{Bypass}}$ is especially effective for those programs. For programs less, make, and screen that contain large recursive cycles, our technique reduces the analysis time by 66.7%, 64.3%, and 78.5%, respectively.

Table 6
Performance of access-based localization for relational analysis. The access-based localization significantly improves the relational analysis (based on the octagon domain [25]) by saving both time and memory costs.

Programs	LOC	OctBase		OctLocal		*Save*
		Time	**Mem**	**Time**	**Mem**	
gzip-1.2.4a	7,327	2078	2832	273	1,072	**87%**
bc-1.06	13,093	9536	6987	1,065	3,230	**89%**
tar-1.13	20,258	∞	∞	9,566	5,963	N/A
less-382	23,822	∞	∞	16,121	8,410	N/A
make-3.76	27,304	∞	∞	17,724	12,771	N/A
wget-1.9	35,018	∞	∞	15,998	9,363	N/A

- $\text{Airac}_{\text{Bypass}}$ is also noticeably effective for other programs. For programs, which have small cycles (consisting of less than 20 procedures), $\text{Airac}_{\text{Bypass}}$ saved 8.9%–44.1% of the analysis time of $\text{Airac}_{\text{ProcAcc}}$. For example, in analyzing parser, $\text{Airac}_{\text{ProcAcc}}$ took 572 s but $\text{Airac}_{\text{Bypass}}$ took 319 s.

The bypassing technique is also likely to reduce peak memory cost. Because the bypassing technique localizes memory states more aggressively than the original access-based localization, the peak memory consumption must be reduced. However, in the experiments, memory cost for analyzing smaller programs (gzip, parser, bc, twolf, tar) slightly increased. This is because $\text{Airac}_{\text{Bypass}}$ additionally keeps bypassing information in memory. But, for larger programs (less, make, wget, screen, bash), the results show that our technique reduces memory costs. For example, $\text{Airac}_{\text{ProcAcc}}$ required 2228 MB in analyzing screen but $\text{Airac}_{\text{Bypass}}$ required 1875 MB.

8.4. Performance of access-based localization for octagons

We implemented relational static analyzers based on the octagon abstract domain [25]: OctBase and OctLocal. They are obtained by replacing interval domains of Airac with octagon domains. Non-numerical values (such as pointers, array, and structures) are handled in the same way as the interval analysis. OctLocal performs the access-based localization in terms of variable packs, as described in the previous section. OctBase is the same as OctLocal except for the localization. To represent octagon domain, we used the Apron library [18].

In all experiments, we used a syntax-directed packing strategy. Our packing heuristic is similar to Miné's approach [25,13], which groups abstract locations that have syntactic locality within code blocks. For example, abstract locations involved in the linear expressions or loops are grouped together. Scope of the locality is limited within each of syntactic C blocks. We also group abstract locations involved in actual and formal parameters, which is necessary to capture relations across procedure boundaries. Large packs whose sizes exceed a threshold (10) were split down into smaller ones.

We have analyzed 6 programs and the Table 6 shows its results. OctBase requires extremely large amount of time and memory. For example, it can only analyze the smallest two programs and for others the analysis immediately hit the cost limit. By contrast, access-based localization makes the analysis much more realistic. For example, OctLocal is able to analyze 35 KLOC within 5 h and 10 GB of memory, which were impossible to analyze without the localization.

9. Related work & discussion

In static program analysis, although localization is a well-known idea for reducing analysis cost, research has been mainly focused on reachability-based approach. For example, in shape analysis, reachability-based localization has been used to improve the scalability of interprocedural analyses [9,31,32,22,15,37,36]. Rinetzky et al. [31,32] define a shape analysis in which called procedures are only passed reachable parts of the heap. Marron et al. [22] reformulate the idea of [31] for graph-based heap models. In separation-logic-based program verification (both by-hand and automatic checking [4,7]), one typically reasons about a command with respect to its footprint (memory cells that the command accesses) in isolation. However, (even) in separation-logic-based program analysis, the framing, which is expressed as accessibility in logic, is conventionally implemented based on reachability [15,37,36]: Gotsman et al. [15] and Yang et al. [37,36] split states based on reachability. Similar reachability-based techniques are also popular in higher-order flow analyses [16,17,24]. Jagannathan et al. [17] use "an abstract form of garbage collection" that removes unreachable bindings.[8] Might et al. [24] formalize the abstract-garbage-collecting control-flow analysis and show that removing unreachable cells significantly improves the analysis performance. However, reachability is, in fact, just one crude but safe approximation method for estimating "live" abstract resources. In this paper, we present a new non-reachability-based approach, access-based localization.

8 In general, abstract garbage collection in control flow analysis is related to strong updates and hence its main role is to improve the analysis precision. However, we limit our discussion on its ability to improve analysis speed [24].

Chen et al. [8] use a mixture of reachability- and access-based localization, but, their approach is more restricted than ours. During reachability-based localization, they try to infer accessed locations by evaluating expressions twice. However, because input states are not at a fixpoint during the course of the analysis, the accessed locations cannot be completely determined. By contrast, our approach makes access-based localization always possible.

Might et al. [23] observe that reachability is overly conservative, and presents a refined localization technique that is orthogonal to our method. From the reachability-based localized state, they additionally exclude some resources that are governed by unsatisfiable conditions. The resulting localized state may contain non-accessed resources that are not governed by such conditions, which could be filtered by our technique. And, since our technique does not consider unsatisfiable conditions, their technique can improve ours.

Our three extensions of access-based localization are new. Previously, localization was applied only for procedures [31,22,15,37]. For the first time, we propose to localize the analysis of arbitrary code blocks instead of whole procedures and show that this fine-grained localization further improves analysis performance in a realistic setting. Our bypassing technique is also new; previous localization techniques all have a common limitation in handling procedure calls, as described in Section 6. In this paper, we show that traditional localization is overly conservative particularly for programs with large cycles of recursive procedures, and provides an analysis algorithm that solves the problem. In addition, we show how to apply the localization technique to relational numeric analysis via variable packings.

Scaling an analysis with an efficient pre-analysis has been used in other contexts. For example, flow-insensitive pre-analysis has been used in dataflow analysis [1] and pointer analysis [35]. Our work is an instance of these lines of research: we use a pre-analysis to localize memory states in actual analysis. In addition, we show some properties that the pre-analysis must satisfy, which is necessary for the safety of the localization.

We do not claim that our access-based localization is cost-effective in general. Our abstract domain is non-relational or packed relational and, consequently, partitioning of the memory states is relatively simple. The partitioning operation will become costly when analysis' abstract semantics involves relational information such as in analysis with (global, i.e., non-packed) relational domain [25] or storeless semantics [31]. In these cases, not only the resources that are directly accessed by a code block but also resources that are indirectly required to keep relational information should be considered. Hence, the localizing operation would get more complicated.

Lastly, one noteworthy point is that designing a correct pre-analysis with a right balance of accuracy and cost was relatively easy in our case because the underlying analysis was designed as an abstract interpretation. Our pre-analysis was simply a further abstraction of the underlying (actual) abstract interpreter.

10. Conclusion

In this article, we have proven that access-based localization is a decisive key to economical global static analysis based on interval domain and allocation-site-based heap abstraction. We have shown that the conventional reachability-based localization is too conservative and proposed a new, access-based technique. We report that the reachability-based technique does not much help in reducing irrelevant entries of abstract memories: in our case, only 1.1%–5.6% of reachable memory entries were actually accessed. Our technique aims to more tightly localize abstract memories so that reachable but not possibly accessed abstract locations are also removed from the localized memories. In a realistic setting, our approach reduced the average analysis time by 92% over the reachability-based approach.

In addition, we showed that three extensions of the access-based localization further enhance the technique.

- We proposed a fine-grained access-based technique to localize the analysis of arbitrary code blocks instead of whole procedures (Section 5). This fine-grained extension turned out to further save the analysis time by 31% on average.
- We presented a technique to mitigate the performance problem of localization in handling procedure calls, particularly for recursive call cycles. This extension was found to be a key to efficient analysis of programs that extensively use recursion. The performance improvement for recursive programs were on average 72%.
- We showed that the access-based localization is effective for some relational analyses (Section 7). For a packed relational analysis [25,5], a variant of popular relational approaches, incorporating access-based localization makes the analysis faster by a factor of eight on average.

As future work, we are interested in applying the access-based localization to other program analysis settings. In this paper, we showed that access-based localization is effective in two settings (non-relational analysis and packed relational numerical analyses for C-like imperative languages). It would be interesting if our technique can be effectively applied to other program analyses and languages, such as "global" relational analysis (without variable packing), higher-order flow analysis [24], and shape analysis [31,32,15,36].

Acknowledgment

We are grateful to the anonymous reviewers of Science of Computer Programming for their very kind and constructed feedbacks on earlier draft of this paper.

Appendix. Proof of Lemma 5

We prove $\mathbf{lfp}(\hat{F}) \sqsubseteq \gamma(\mathbf{lfp}(\hat{F}_p))$. By the help of the fixpoint transfer theorem [11], it is enough to prove the following two properties:

- α and γ are related by Galois connection
- $\alpha \circ \hat{F} \sqsubseteq \hat{F}_p \circ \alpha$

For the first part (Galois connection), we have to show $\forall x \in \mathbb{C} \to \hat{\mathbb{S}}, y \in \hat{\mathbb{S}}.\ \alpha(x) \sqsubseteq y \Leftrightarrow x \sqsubseteq \gamma(y)$.

- $\alpha(x) \sqsubseteq y \Rightarrow x \sqsubseteq \gamma(y)$:

$$\begin{array}{lll} & \alpha(x) \sqsubseteq y & \\ \Rightarrow & \bigsqcup_{c\in\mathbb{C}} x(c) \sqsubseteq y & \cdots \text{ def. of } \alpha \\ \Rightarrow & \forall c \in \mathbb{C}.x(c) \sqsubseteq y & \cdots \forall c \in \mathbb{C}.x(c) \sqsubseteq \bigsqcup_{c'\in\mathbb{C}} x(c') \\ \Rightarrow & x \sqsubseteq \lambda c.y & \cdots \text{ def. of } \sqsubseteq \text{ on } \mathbb{C} \to \hat{\mathbb{S}} \\ \Rightarrow & x \sqsubseteq \gamma(y) & \cdots \text{ def. of } \gamma \end{array}$$

- $\alpha(x) \sqsubseteq y \Leftarrow x \sqsubseteq \gamma(y)$:

$$\begin{array}{lll} & x \sqsubseteq \gamma(y) & \\ \Rightarrow & x \sqsubseteq \lambda c.y & \cdots \text{ def. of } \gamma \\ \Rightarrow & \forall c.x(c) \sqsubseteq y\ (= (\lambda c.y)(c)) & \cdots \text{ def. of } \sqsubseteq \text{ on } \mathbb{C} \to \hat{\mathbb{S}} \\ \Rightarrow & \bigsqcup_{c\in\mathbb{C}} x(c) \sqsubseteq y & \cdots y \text{ is an upper bound of } \{x(c) \mid c \in \mathbb{C}\} \\ \Rightarrow & \alpha(x) \sqsubseteq y & \cdots \text{ def. of } \alpha \end{array}$$

Next, we show $\alpha \circ \hat{F} \sqsubseteq \hat{F}_p \circ \alpha$, i.e., $\forall d \in \mathbb{C} \to \hat{\mathbb{S}}.\alpha(\hat{F}(d)) \sqsubseteq \hat{F}_p(\alpha(d))$.

$$\begin{array}{lll} \alpha(\hat{F}(d)) & = \alpha\left(\lambda c.\ \bigsqcup_{c'\hookrightarrow c} \hat{f}_{c'}(d(c'))\right) & \cdots \text{ def. of } \hat{F} \\ & = \bigsqcup_{c\in\mathbb{C}} \left(\bigsqcup_{c'\hookrightarrow c} \hat{f}_{c'}(d(c'))\right) & \cdots \text{ def. of } \alpha \\ & \sqsubseteq \bigsqcup_{c\in\mathbb{C}} \left(\bigsqcup_{c'\in\mathbb{C}} \hat{f}_{c'}(d(c'))\right) & \cdots \forall c \in \mathbb{C}.\hat{f}_c \text{ is mono. and } \{c' \mid c' \hookrightarrow c\} \subseteq \mathbb{C} \\ & = \bigsqcup_{c\in\mathbb{C}} \hat{f}_c(d(c)) & \cdots \forall \hat{s}.\ \bigsqcup_{c\in\mathbb{C}} \hat{s} = \hat{s}, \text{ renaming } c' \text{ by } c \\ & \sqsubseteq \bigsqcup_{c\in\mathbb{C}} \hat{f}_c\left(\bigsqcup_{c'\in\mathbb{C}} d(c')\right)\Big) & \cdots \forall c \in \mathbb{C}.\hat{f}_c \text{ is mono. and } d(c) \sqsubseteq \bigsqcup_{c'\in\mathbb{C}} d(c') \\ & = \bigsqcup_{c\in\mathbb{C}} \hat{f}_c(\alpha(d)) & \cdots \text{ def. of } \alpha \\ & = \hat{F}_p(\alpha(d)) & \cdots \text{ def. of } \hat{F}_p \quad \square \end{array}$$

References

[1] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, Westley Weimer, Speeding up dataflow analysis using flow-insensitive pointer analysis, in: Proceedings of the International Symposium on Static Analysis, 2002, pp. 230–246.

[2] Xavier Allamigeon, Wenceslas Godard, Charles Hymans, Static analysis of string manipulations in critical embedded C programs, in: Proceedings of the International Symposium on Static Analysis, 2006, pp. 35–51.

[3] Gogul Balakrishnan, Thomas Reps, Analyzing memory accesses in x86 binary executables, in: Proceedings of the International Conference on Compiler Construction, 2004, pp. 5–23.

[4] Josh Berdine, Cristiano Calcagno, Peter W. O'Hearn, Symbolic execution with separation logic, in: Proceedings of the Asian Symposium on Programming Languages and Systems, Springer, 2005, pp. 52–68.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, 2003, pp. 196–207.

[6] Francois Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Proceedings of the International Conference on Formal Methods in Programming and their Applications, 1993, pp. 128–141.

[7] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, Hongseok Yang, Compositional shape analysis by means of bi-abduction, in: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '09, New York, NY, USA, ACM, 2009, pp. 289–300.

[8] Li-Ling Chen, Williams Ludwell Harrison III, An efficient approach to computing fixpoints for complex program analysis, in: Proceedings of the 8th international conference on Supercomputing, 1994, pp. 98–106.

[9] Stephen Chong, Radu Rugina, Static analysis of accessed regions in recursive data structures, in: Proceedings of the International Symposium on Static Analysis, Springer, 2003, pp. 463–482.

[10] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: Proceedings of the Second International Symposium on Programming, Dunod, Paris, France, 1976, pp. 106–130.

[11] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, ACM Press, New York, NY, 1979, pp. 269–282.
[12] P. Cousot, R. Cousot, Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, in: Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92, 1992, pp. 269–295.
[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, Why does astrée scale up? Formal Methods in System Design 35 (3) (2009) 229–264.
[14] Patrick Cousot, Radhia Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[15] Alexey Gotsman, Josh Berdine, Byron Cook, Interprocedural shape analysis with separated heap abstractions, in: Proceedings of the International Symposium on Static Analysis, 2006, pp. 240–260.
[16] Williams L. Harrison III, The interprocedural analysis and automatic parallelization of scheme programs, Ph.D. Thesis, Center for Supercomputing Research and Development, University of Illinois at Urabana-Champaign, February 1989.
[17] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, Andrew Wright, Single and loving it: must-alias analysis for higher-order languages, in: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, pp. 329–341.
[18] B. Jeannet, A. Miné, Apron: a library of numerical abstract domains for static analysis, in: Computer Aided Verification, CAV'2009, 2009, pp. 661–667.
[19] Yongin Jhee, Minsik Jin, Yungbum Jung, Deokhwan Kim, Soonho Kong, Heejong Lee, Hakjoo Oh, Daejun Park, Kwangkeun Yi, Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at the Workshop of the 30 Years of Abstract Interpretation, San Francisco January 2008. ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf.
[20] Yungbum Jung, Jaehwang Kim, Jaeho Shin, Kwangkeun Yi, Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis, in: Proceedings of the International Symposium on Static Analysis, 2005, pp. 203–217.
[21] Yungbum Jung, Kwangkeun Yi, Practical memory leak detector based on parameterized procedural summaries, in: Proceedings of the International Symposium on Memory Management, 2008, pp. 131–140.
[22] Mark Marron, Manuel Hermenegildo, Deepak Kapur, Darko Stefanovic, Efficient context-sensitive shape analysis with graph based heap models, in: Proceedings of the International Conference on Compiler Construction, 2008, pp. 245–259.
[23] Matthew Might, Benjamin Chambers, Olin Shivers, Model checking via ΓCFA, in: Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2007, Nice, France, January 2007, pp. 59–73.
[24] Matthew Might, Olin Shivers, Improving flow analyses via ΓCFA: Abstract garbage collection and counting, in: Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming, 2006, pp. 13–25.
[25] A. Miné, The octagon abstract domain, Higher-Order and Symbolic Computation 19 (1) (2006) 31–100.
[26] Hakjoo Oh, Large spurious cycle in global static analyses and its algorithmic mitigation, in: Proceedings of the Asian Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 5904, Springer-Verlag, December 2009, pp. 14–29.
[27] Hakjoo Oh, Lucas Brutschy, Kwangkeun Yi, Access analysis-based tight localization of abstract memories, in: VMCAI 2011: 12th International Conference on Verification, Model Checking, and Abstract Interpretation, in: Lecture Notes in Computer Science, vol. 6538, Springer, 2011, pp. 356–370.
[28] Hakjoo Oh, Kwangkeun Yi, An algorithmic mitigation of large spurious interprocedural cycles in static analysis, Software: Practice and Experience 40 (8) (2010) 585–603.
[29] Hakjoo Oh, Kwangkeun Yi, Access-based localization with bypassing, in: Proceedings of the Asian Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 7078, Springer-Verlag, December 2011, pp. 50–65.
[30] Anthony Pioli, Michael Hind, Combining interprocedural pointer analysis and conditional constant propagation, Technical Report, IBM T. J. Watson Research Center, 1999.
[31] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, Reinhard Wilhelm, A semantics for procedure local heaps and its abstractions, in: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005, pp. 296–309.
[32] Noam Rinetzky, Mooly Sagiv, Eran Yahav, Interprocedural shape analysis for cutpoint-free programs, in: Proceedings of the International Symposium on Static Analysis, 2005, pp. 284–302.
[33] Jaeho Shin, An abstract interpretation with the interval domain for C-like programs, Technical Memorandum ROPAS-2006-32, Programming Research Laboratory, School of Computer Science & Engineering, Seoul National University, 2006.
[34] Arnaud Venet, Guillaume Brat, Precise and efficient static array bound checking for large embedded C programs, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI '04, New York, NY, USA, ACM, 2004, pp. 231–242.
[35] Guoqing Xu, Atanas Rountev, Manu Sridharan, Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis, in: Proceedings of the European Conference on Object-Oriented Programming, 2009, pp. 98–122.
[36] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Scalable shape analysis for systems code, in: Proceedings of the International Conference on Computer Aided Verification, 2008, pp. 385–398.
[37] Hongseok Yang, Oukseh Lee, Cristiano Calcagno, Dino Distefano, Peter O'Hearn, On scalable shape analysis, Technical Memorandum RR-07-10, Queen Mary University of London, Department of Computer Science, November 2007.