Access-Based Localization with Bypassing

Hakjoo Oh and Kwangkeun Yi

Seoul National University

Abstract. We present an extension of access-based localization technique to mitigate a substantial inefficiency in handling procedure calls. Recently, access-based localization was proposed as an effective way of tightly localizing abstract memories. However, it 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. The weakness is especially exacerbated in the presence of recursive call cycles, which is common in analysis of realistic programs. In this paper, we present a technique, called bypassing, that mitigates the problem. Our technique localizes input memory states only with memory locations that the procedure directly accesses. Those parts not involved in analysis of the procedure are bypassed to transitively called procedures. In experiments with an industrial-strength global C static analyzer, the technique reduces the average analysis time by 42%. In particular, the technique is especially effective for programs that extensively use recursion: it saves analysis time by 77% on average.

1 Introduction

Memory localization is vital for reducing global analysis cost [12,14,3,23,22]. The performance problem of flow-sensitive global analysis is that code blocks such as procedure bodies are repeatedly analyzed (often needlessly) with different input memory states. Localization of input abstract memories, which removes the irrelevant memory entries that will not be used inside called procedure bodies, alleviates the problem by increasing the chance of reusing 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 input states (localization), the analysis result for the first call can be reused at the second call without re-analyzing the procedure body.

Access-based technique provides an effective way of realizing memory localization [14]. Because localization must be done before analyzing a procedure, it is impossible to exactly compute to-be-used parts of input memory. Thus, some approximation must be involved, so that the localized state can contain some spurious entries that will not be actually used by the procedure. The conventional approximation methods are reachability-based techniques: from input memory, only the abstract locations reachable from actual parameters and global locations are collected. However, the technique is too conservative in practice because only few reachable locations are actually accessed [14]. Access-based technique, on the other hand, trims input memory states more aggressively: locations that are reachable but may not be accessed are additionally removed. The access information is computed by a conservative pre-analysis. Thus, access-based localization more effectively reduces global analysis cost than the reachability-based technique does [14].

However, the localization has a source of inefficiency in handling procedure calls. In access-based localization[1], 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 body).

Such inefficiency is especially exacerbated with recursive call cycles. Consider a recursive call cycle $f \rightarrow g \rightarrow h \rightarrow f \rightarrow \cdots$. 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 help any more inside call cycles. Moreover, recursive cycles (even large ones) are common in real C programs. For example, in GNU open source code, we noticed that a number of programs have large recursive cycles and a single cycle sometimes contains more than 40 procedures. This is the main performance bottleneck of access-based localization in practice (Section 4.2).

In this paper, we extend access-based localization technique so that the aforementioned inefficiency can be relieved. With our technique, localized states for a procedure contains only the abstract locations that are accessed by the procedure and does 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 procedure involves only the memory parts that the procedure directly accesses (even inside recursive cycles), which results in more tight localization and hence reduces analysis cost more than access-based localization does. The following example illustrates how our technique saves cost.

Example 1. Consider the following code.

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

[1] In fact, any localization techniques suffers from similar problems. In this paper, we discuss the problem in the context of access-based localization.

Procedure `main` calls `f`, and `f` calls `g`. Procedures `f` and `g` update the value of `a` and `b`, respectively. Procedure `main` calls `f` two times with the value of `b` 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 locations `a` and `b` because both are (directly/indirectly) accessed while analyzing `f`. The localized state at the second call (line 6) contains the same locations. Because the value of `b` is changed, `f` (as well as `g`) is re-analyzed at the second call.
– With our technique: `f` is analyzed only once (though `g` is analyzed twice). Localized memories for procedure `f` contain only the location that `f` directly accesses, i.e., `a`. The value of `a` is not changed and the body of `f` is not re-analyzed at the second call. However, procedure `g` is re-analyzed because we propagate the changed value of `b` to the entry of `g`.

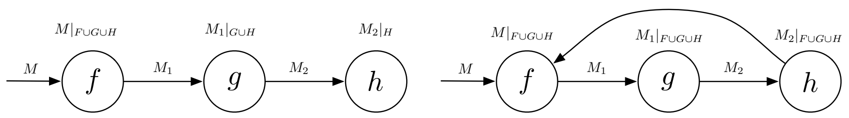
In experiments with an industrialized abstract interpretation-based static analyzer, our technique saved 9–79%, on average 42%, in analysis time in comparison with the access-based localization technique for a variety of open-source C benchmarks (2K–100K). In particular, for those benchmark programs that extensively use recursion and have large recursive call cycles, our technique is more effective: it reduces the analysis time for those programs by 77% on average. The technique does not compromise the analysis precision.

Contributions. This paper makes the following contributions.

– We report on a substantial performance degradation of localization and present a technique to mitigate the problem. Our technique is meaningful because real C programs often have complex procedural relationships such as large recursive cycles that significantly exacerbate the problem. Though we focus on access-based localization, any localization schemes (including reachability-based ones) suffer from similar (basically the same) problems. To the best of our knowledge, these aspects of localization techniques have not been adequately addressed in the literature.
– We prove the effectiveness of our technique by experiments with an industrial-strength C static analyzer [8,9,10,13,14,15].

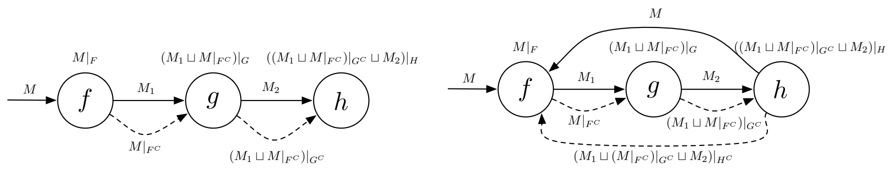
Overview. We illustrate how our technique works with examples. Fig. 1 shows example call graphs. There are three procedures: f, g and h. Suppose F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. We describe how the problem occurs 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. 1(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 with 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 from the fact that not the entire localized memory is accessed



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

Fig. 1. 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.


$M|_F$
$(M_1 \sqcup M|_{F^C})|_G$
$((M_1 \sqcup M|_{F^C})|_{G^C} \sqcup M_2)|_H$
M
M_1
M_2
$M|_{F^C}$
$(M_1 \sqcup M|_{F^C})|_{G^C}$
$(M_1 \sqcup (M|_{F^C})|_{G^C} \sqcup M_2)|_{H^C}$

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

Fig. 2. 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.

by each procedure. For example, abstract locations $G \cup H$ are not necessary in analyzing the body of f.

The problem becomes severe when analyzing recursive call cycles. Consider Fig. 1(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. 2 illustrates how our technique works. We first consider non-recursive call case (Fig. 2(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. 2(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. 2(b), the localized memory for f (resp., g and h) only contains locations F (resp., G and H).

2 Setting: Analysis Framework

We describe our analysis framework. The analysis basically performs flow-sensitive and context-insensitive global analysis, and an abstract memory state is represented by a map from abstract locations to abstract values. In Section 3, we present our technique on top of this framework. Section 2.1 shows the intermediate representation of programs, and Section 2.2 defines the analysis in terms of abstract domain and semantics.

2.1 Graph Representation of Programs

We assume that a program is represented by a supergraph [17]. A supergraph consists of control flow graphs of procedures with interprocedural edges connecting each call-site to its callee and callees to the corresponding return-sites. Each node $n \in \mathit{Node}$ in the control flow graph has one of the four types :

$$\mathit{entry} \mid \mathit{exit} \mid \mathit{call}(f_x, e) \mid \mathit{return} \mid \mathit{set}(lv, e)$$

Each control flow graph has *entry* and *exit* nodes. A call-site in a program is represented by a call node and its corresponding return node. A call node $\mathit{call}(f_x, e)$ indicates that it invokes a procedure f, its formal parameter is x, and the actual parameter is e. For simplicity, we assume that there are no function pointers in the program and consider only one parameter. Node type *return* indicates a return node of a call node. $\mathit{set}(lv, e)$ represents an assignment statement that assigns the value of e into the location that l-value expression lv denotes. In this paper, we do not restrict expression (e) and l-value expression (lv) to specific ones. We assume that edges in flow graphs are assembled by function $\mathsf{succof} \in \mathit{Node} \rightarrow 2^{\mathit{Node}}$, which maps each node to its successors.

2.2 Static Analysis

We consider static analyses, in which the set of (possibly infinite) concrete memory states are represented by an abstract memory state:

$$\hat{\mathit{Mem}} = \hat{\mathit{Addr}} \rightarrow \hat{\mathit{Val}}$$

That is, $\hat{\mathit{Mem}}$ is a map from abstract locations ($\hat{\mathit{Addr}}$) to the abstract values ($\hat{\mathit{Val}}$). We assume that $\hat{\mathit{Addr}}$ is a finite set and $\hat{\mathit{Val}}$ is an arbitrary cpo (complete partial order). We assume further that abstract values and locations are computed by two functions $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, respectively. Given an expression e and an abstract memory state $\hat{m}$, $\hat{\mathcal{V}}(\in e \rightarrow \hat{\mathit{Mem}} \rightarrow \hat{\mathit{Val}})$ evaluates the abstract value that e denotes under $\hat{m}$. Similarly, $\hat{\mathcal{L}}(\in lv \rightarrow \hat{\mathit{Mem}} \rightarrow 2^{\hat{\mathit{Addr}}})$ evaluates the set of abstract locations of lv under $\hat{m}$.

With $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, we define semantic function $\hat{f} : \mathit{Node} \rightarrow \hat{\mathit{Mem}} \rightarrow \hat{\mathit{Mem}}$. Given a node n and an input memory state m, $\hat{f}(n)(m)$ computes the effect of the command in node n on the input state :

$$\hat{f}(n)(\hat{m}) = \begin{cases} \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!/ \hat{\mathcal{L}}(x)(\hat{m})\} & \text{if } n = call(f_x, e) \\ \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!/ \hat{\mathcal{L}}(lv)(\hat{m})\} & \text{if } n = set(lv, e) \\ \hat{m} & \text{otherwise} \end{cases}$$

where, $\hat{m}\{v /\!/ \{l_1, \ldots, l_k\}\}$ means $\hat{m}\{l_1 \mapsto (\hat{m}(l_1) \sqcup v)\} \cdots \{l_k \mapsto (\hat{m}(l_k) \sqcup v)\}$. The effect of node $set(lv, e)$ is just to (weakly) assign the abstract value of e into the locations in $\hat{\mathcal{L}}(lv)(\hat{m})$.[2] The call node command $call(f_x, e)$ binds the formal parameter x to the value of actual parameter e. Please 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.

Then, the analysis is to compute a fixpoint table $\mathcal{T} \in Node \to \hat{Mem}$ that maps each node in the program to its (input) abstract memory state. The map is defined by the least fixpoint of the following function $\hat{F}$:

$$\hat{F} : (Node \to \hat{Mem}) \to (Node \to \hat{Mem})$$
$$\hat{F}(T) = \lambda n. \bigsqcup_{p \in \mathsf{predof}(n)} \hat{f}(n)(\mathcal{T}(p))$$

The fixpoint is computed by a worklist algorithm. The worklist consists of nodes of the control flow graph of the program whose abstract state has to be recomputed. When a computed memory state for n is changed, we add successors of n into the worklist. The algorithm uses widening operation [2] to guarantee termination. Fig. 4(a) shows the algorithm.

3 Access-Based Localization with Bypassing

In this section, we describe our technique on top of the analysis framework (Section 2). Our technique is an extension of the access-based localization. In Section 3.1, we describe the access-based localization. Then, we extends the localization technique to derive our bypassing technique.

3.1 Access-Based Localization: Previous Approach

In access-based localization [14], the entire analysis is staged into two phases: (1) a pre-analysis conservatively estimates the set of abstract locations that will be accessed during actual analysis for each procedure; (2) then, the actual analysis uses the access-set results and, right before analyzing each procedure, filters out memory entries that will not be accessed inside the procedure's body.

The pre-analysis is a further abstraction of the original analysis. The pre-analysis must be safe in that the estimated access information should be conservative with respect to the actual access set that would be used during the actual analysis. Moreover, to be useful, the estimation should be efficient enough to compensate for the extra burden of running pre-analysis once more. In [14],

[2] For brevity, we consider only weak updates. Applying strong update is orthogonal to our technique we present in this paper.

such a pre-analysis is obtained by applying conservative abstractions (such as ignoring statement orders, i.e., flow-insensitivity) to the abstract semantics of original analysis. During the pre-analysis, the access-sets for each program point are collected. Let $\mathcal{A} \in Node \to 2^{\hat{Addr}}$ be the access information. That is, all the abstract locations that are accessed during the analysis at node n are represented by $\mathcal{A}(n)$.

The actual analysis is the same as the original analysis except for localizing operation. Because actual analysis additionally performs localizations using the access-set information, the abstract semantics for call node is changed. Thus, in actual analysis, non-accessed memory locations are excluded from the input memories of procedures: given an input memory state $\hat{m}$ to a call node $call(f_x, e)$, the semantic function $\hat{f}$ for the call statement $call(f_x, e)$ is changed as follows:

$$\hat{f}\ call(f_x, e)\ \hat{m} \quad = \quad \hat{m}'|_{\mathsf{access}(f)} \text{ where } \hat{m}' = \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \{x\}\}$$

That is, after parameter binding ($\hat{m}'$) the memory is restricted on $\mathsf{access}(f)$, where $\mathsf{access}(f)$ is defined as follows: ($\mathsf{callees}(f)$ denotes the set of procedures, including f, that are reachable from f via the call-graph and $\mathsf{nodesof}(f)$ the set of nodes in procedure f.)

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} (\bigcup_{n \in \mathsf{nodesof}(g)} \mathcal{A}(n))$$

$\mathsf{access} \in ProcId \to 2^{\hat{Addr}}$ maps each procedure to a set of abstract locations that are possibly accessed during the analysis of callee procedures. Note that we consider all the transitively called procedures as well, instead of just considering the directly called procedure.

3.2 Access-Based Localization with Bypassing: Our Approach

Definition 1 (directly/indirectly(transitively) called procedure). *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.*

Example 2. 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.

Definition 2 (directly/indirectly accessed locations). *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 3. 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).

Our technique is an extension of access-based localization. Thus, we also separate the entire analysis into two phases: pre-analysis, and actual analysis.

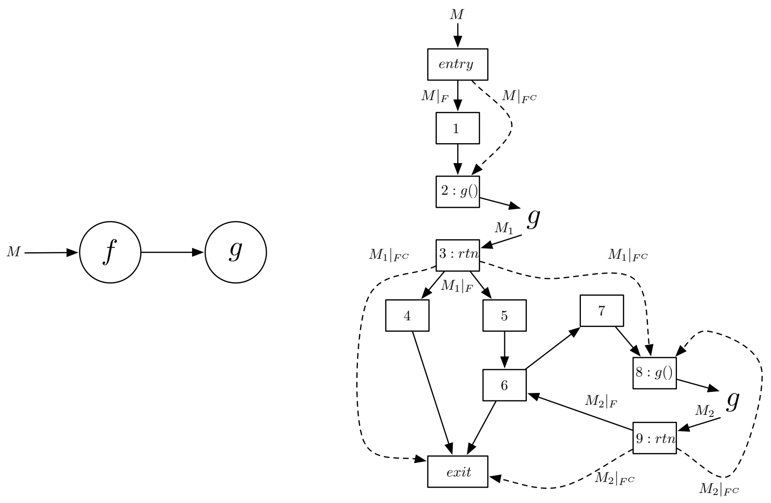
Pre-analysis is slightly changed. Pre-analysis is exactly the same with the one that would be used in access-based localization, except that we use its result in a different way. In access-based localization, we compute $\mathsf{access}(f)$, which includes abstract locations directly accessed by f as well as locations indirectly accessed by f. Instead, our technique computes $\mathsf{direct} \in \mathit{ProcId} \rightarrow 2^{\hat{Addr}}$ that maps each procedure to a set of abstract locations that are directly accessed by the procedure, excluding indirectly accessed locations. Given $\mathcal{A} : \mathit{Node} \rightarrow 2^{\hat{Addr}}$ from pre-analysis, the set $\mathsf{direct}(f)$ is defined as follows:

$$\mathsf{direct}(f) = \bigcup_{n \in \mathsf{nodesof}(g)} \mathcal{A}(n)$$

Major changes are in actual analysis. With access-based localization, actual analysis performs 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 nodes where memory states flow into the nodes from other procedures. These nodes include entry and return nodes: 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 node. Hence, the memory states at entry and return nodes of a procedure are localized so that the procedure is analyzed with the directly accessed locations. We call such nodes, where localization occurs, bypassing sources.

  Example 4. Consider the Fig. 3. Fig. 3(a) shows a call-graph, where procedure f calls g, and Fig. 3(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*, node 3, and node 9. Nodes 3 and 9 are return nodes. At *entry*, the input memory M is restricted on F. Hence, node 1 is analyzed with the localized memory $M|_F$. At node 3 and 9, the memory returned from procedure g, M_1 and M_2 are restricted on 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 strictly bigger than $M|_F$.



(a) call-graph (b) control flow graph of f

Fig. 3. 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

– **Bypassing:** Bypassing happens between bypassing sources and targets. The non-localized parts at bypassing sources (entry or return nodes) should be delivered to nodes where memory states flow into other procedures. These nodes include procedure exit and call nodes: at procedure exit, the output memory state of the procedure is propagated to the caller, and at call nodes, 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 nodes that are reachable without passing through other call nodes. We call such call and exit nodes as bypassing targets.

Example 5. Consider the Fig. 3(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 node 2. Another call node 8 or exit node are not bypassing target for *entry* because they are not reachable from *entry* without passing through the call node 2. And, bypassing targets for node 3 are 8 and *exit*. Similarly, bypassing targets for node 9 are 8 and *exit*. At entry node, the non-localized memory parts ($M|_{F^C}$) are bypassed to entry's bypassing target, node 2. Similarly, at nodes 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. 4(b) shows our technique integrated in the worklist-based analysis algorithm. In order to transform access-based localization into our technique, only shaded lines are inserted; other parts remain the same. Predicate bypass_source $\in$

```
(01) : $\mathcal{W} \in \mathit{Worklist} = 2^{\mathit{Node}}$
(02) : $\mathcal{T} \in \text{Table} = \mathit{Node} \to \mathit{Mem}$
(03) : $\hat{f} \in \mathit{Node} \to \mathit{Mem} \to \mathit{Mem}$

(04) : $\mathit{FixpointIterate}\ (\mathcal{W}, \mathcal{T}) =$
(05) : repeat
(06) :     $n := \mathsf{choose}(\mathcal{W})$
(07) :     $m := \hat{f}(n)(\mathcal{T}(n))$
(15) :     for all $n' \in \mathsf{succof}(n)$ do
(16) :         if $m \not\sqsubseteq \mathcal{T}(n')$
(17) :             $\mathcal{W} := \mathcal{W} \cup \{n'\}$
(18) :             $\mathcal{T}(n') := \mathcal{T}(n') \sqcup m$
(19) : until $\mathcal{W} = \emptyset$
```

(a) The worklist-based algorithm

```
(01) : $\mathcal{W} \in \mathit{Worklist} = 2^{\mathit{Node}}$
(02) : $\mathcal{T} \in \text{Table} = \mathit{Node} \to \mathit{Mem}$
(03) : $\hat{f} \in \mathit{Node} \to \mathit{Mem} \to \mathit{Mem}$

(04) : $\mathit{FixpointIterate}\ (\mathcal{W}, \mathcal{T}) =$
(05) : repeat
(06) :     $n := \mathsf{choose}(\mathcal{W})$
(07) :     $m := \hat{f}(n)(\mathcal{T}(n))$
(08) :     if $\mathsf{bypass\_source}(n)$ then
(09) :         $(m_l, m_b) := \mathsf{project}(m, \mathsf{procof}(n))$
(10) :         for all $t \in \mathsf{bypass\_target}(n)$ do
(11) :             if $m_b \not\sqsubseteq \mathcal{T}(t)$
(12) :                 $\mathcal{T}(t) := \mathcal{T}(t) \sqcup m_b$
(13) :                 $\mathcal{W} := \mathcal{W} \cup \{t\}$
(14) :         $m := m_l$
(15) :     for all $n' \in \mathsf{succof}(n)$ do
(16) :         if $m \not\sqsubseteq \mathcal{T}(n')$
(17) :             $\mathcal{W} := \mathcal{W} \cup \{n'\}$
(18) :             $\mathcal{T}(n') := \mathcal{T}(n') \sqcup m$
(19) : until $\mathcal{W} = \emptyset$
```

(b) The algorithm with bypassing

Fig. 4. Comparison of the normal analysis algorithm and our bypassing algorithm: our technique is a simple addition of the traditional algorithm

Node $\to$ *bool* checks whether a node is a bypass source or not. Function procof $\in$ *Node* $\to$ *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 bypass_target $\in$ *Node* $\to 2^{\mathit{Node}}$ maps each bypass source to its bypass targets. If the current node n is a bypass source (line 8), the memory state m is divided into a local memory m_l and the rest part m_b (line 9). The local memory m_l is propagated to the successors of n as in the case of the normal algorithm (line 14). The non-localized memory (m_b) is updated to the input memories of bypassing targets of n (line 10–13).

3.3 Delivery Points Optimization

Bypassing operation induces additional join operations, one of the most expensive operation in semantic-based static analyses [1,9]. 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. 3. At node 2, two input memories, one propagated from node 1 and another bypassed from *entry*, are joined. Similarly, at the other bypassing targets (node 8 and *exit*), additional join operations take place.

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

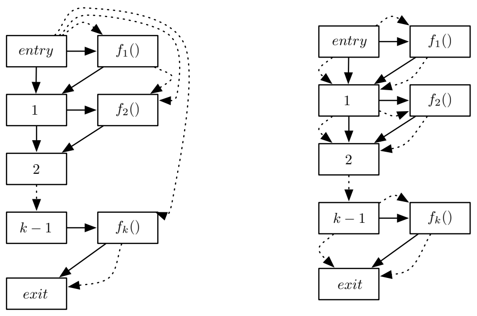


Fig. 5. Example of common code patterns that increases bypassing overhead

We noticed that the number of additional joins is sometimes unbearable. For example, Fig. 5 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,`$\cdots$,`fk` are sequentially called after respective condition checks (`cond1, cond2,`$\cdots$`, condk`). For this code, bypassing happens as follows (as dashed lines in Fig. 5 show):

– From *entry* to $f_1, f_2, f_3, \ldots, f_k, exit$
– From f_1 to $f_2, f_2, f_3, \ldots, f_k, exit$
– $\cdots$
– 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 nodes that reduces the total number of bypassing edges. These nodes, we call them "delivery points", include some join points and loop heads. For example, in Fig. 5, we use nodes $\{1, 2, \cdots, k-1\}$ as delivery points and let bypassing drop by those nodes. As a result, bypassing happens as shown in the rightmost graph in Fig. 5. Bypassing from *entry* to $call_1$ takes place as in before, but Instead of bypassing from *entry* to $\{f_2, \cdots, f_k, exit\}$, we pass through node $1, 2, \cdots, k-1$, 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.

4 Experiments

We check the performance of our technique by experiments with Airac, a global abstract interpretation engine in an industrialized bug-finding analyzer Sparrow [8,9,10,13,14,15].

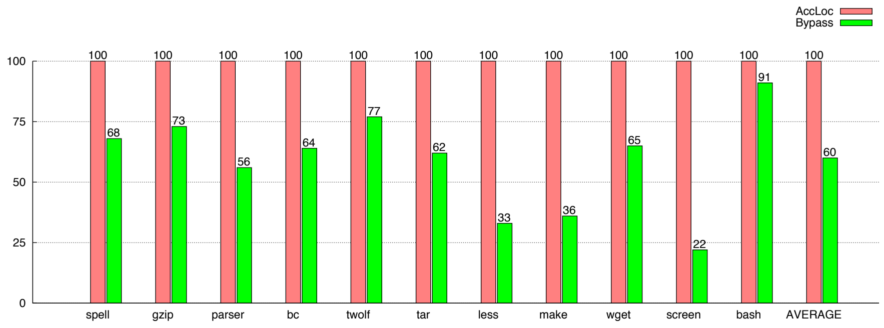


Fig. 6. Comparison of analysis time between access-based localization and bypassing

4.1 Setting Up

Airac is an interval-domain-based abstract interpreter. The analyzer performs flow-sensitive and context-insensitive global analysis: it computes a map $\mathcal{T} \in Node \rightarrow \hat{Mem}$ from program points ($Node$) to abstract memories ($\hat{Mem}$). An abstract memory is a map $\hat{Mem} = \hat{Addr} \rightarrow \hat{Val}$ where $\hat{Addr}$ denotes abstract locations that are either program variables or allocation sites, and $\hat{Val}$ denotes abstract values including interval values, addresses, array blocks, and structure blocks. The details of abstract domain and semantics are described in [14].

From our baseline analyzer Airac, we have made two analyzers: $\text{Airac}_{\text{AccLoc}}$ and $\text{Airac}_{\text{Bypass}}$ that respectively use the access-based localization and our technique. $\text{Airac}_{\text{Bypass}}$ is exactly the same as $\text{Airac}_{\text{AccLoc}}$ 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. The analyzers are written in OCaml.

We have analyzed 10 software packages. Fig. 1 shows our benchmark programs. **LOC** indicates the number of lines of code before preprocessing. **Proc** indicates the number of procedures in each program. **LRC** represents the size of largest recursive call cycle contained in each program. For example, the program `screen` have 589 procedures and, among them, 77 procedures belong to a single recursive cycle. We analyzed each program globally: the entire program is analyzed starting from the entry of 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 (in seconds) spent during the analysis; (2) *MB* is the peak memory consumption (in megabytes) during the analysis.

4.2 Results

Fig. 6 compares the *time* of $\text{Airac}_{\text{AccLoc}}$ and $\text{Airac}_{\text{Bypass}}$. Table 1 shows the raw analysis results. Overall, $\text{Airac}_{\text{Bypass}}$ saved 8.9%–78.5%, on average 42.1%, of the analysis time of $\text{Airac}_{\text{AccLoc}}$. There are some noteworthy points.

Table 1. Program properties and analysis results. 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. $\mathsf{Airac}_{\mathsf{AccLoc}}$ uses access-based localization for procedure calls and $\mathsf{Airac}_{\mathsf{Bypass}}$ uses our technique. *time* for $\mathsf{Airac}_{\mathsf{AccLoc}}$ and $\mathsf{Airac}_{\mathsf{Bypass}}$ are the total time that includes pre-analysis time. *Save* shows time savings in percentage of $\mathsf{Airac}_{\mathsf{Bypass}}$ against $\mathsf{Airac}_{\mathsf{AccLoc}}$.

Program	**LOC**	**Proc**	**LRC**	$\mathsf{Airac}_{\mathsf{AccLoc}}$		$\mathsf{Airac}_{\mathsf{Bypass}}$		*Save*
				time(sec)	*MB*	*time*(sec)	*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	2,228	66,920.6	1,875	78.5%
bash-2.05a	105,174	959	4	1,637.6	272	1,492.4	265	8.9%

- Some programs contain large recursive call cycles. One common belief for C programs is that it does not largely use recursion in practice. However, our finding from the benchmark programs is that some programs extensively use recursion and large recursive cycles unexpectedly exist in a number of real C programs. For example, from Table 1, note that program `less`, `make`, and `screen` have recursive cycles (scc) that contain more than 40 procedures.
- $\mathsf{Airac}_{\mathsf{AccLoc}}$ 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 1,637s. However, analyzing those programs that have large recursive cycles takes much more time: `less` and `make` take more than 10,000s and `screen` takes more than 310,000s to finish the analysis, even though they are not the largest programs.
- $\mathsf{Airac}_{\mathsf{Bypass}}$ 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.
- $\mathsf{Airac}_{\mathsf{Bypass}}$ is also noticeably effective for other programs. For programs, which have small cycles (consisting of less than 20 procedures), $\mathsf{Airac}_{\mathsf{Bypass}}$ saved 8.9%–44.1% of the analysis time of $\mathsf{Airac}_{\mathsf{AccLoc}}$. For example, in analyzing `parser`, $\mathsf{Airac}_{\mathsf{AccLoc}}$ took 572 seconds but $\mathsf{Airac}_{\mathsf{Bypass}}$ took 319 seconds.

Our technique is also likely to reduce peak memory cost. Because our technique localizes memory states more aggressively than the 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 $\mathsf{Airac}_{\mathsf{Bypass}}$ additionally keeps bypassing information on memory. But, for larger programs (`less, make, wget, screen, bash`), the results show that our technique reduces memory costs. For example, $\mathsf{Airac}_{\mathsf{AccLoc}}$ required 2,228 MB in analyzing `screen` but $\mathsf{Airac}_{\mathsf{Bypass}}$ required 1,875 MB.

$\mathsf{Airac}_{\mathsf{Bypass}}$ is at least as precise as $\mathsf{Airac}_{\mathsf{AccLoc}}$. In principle, more aggressive localization improves our analysis because unnecessary values are not passed to procedures and hence avoids needless widening operations. In experiments (similar to one performed in [13,15], the precision of $\mathsf{Airac}_{\mathsf{Bypass}}$ was the same as $\mathsf{Airac}_{\mathsf{AccLoc}}$.

5 Related Work

In static analysis, localization has been widely used for reducing analysis cost [23,22,3,18,19,11,7,12,14], but previous localization methods have a common limitation as described in this paper. Previous localization schemes are classified into reachability-based and access-based. For example, in shape analysis, called procedures are only passed with reachable parts of the heap, which improves the scalability of interprocedural shape analysis [18,19,11,3,23,22]. Similar reachability-based techniques, which removes unreachable bindings, are also popular in higher-order flow analyses [6,7,12]. Access-based localization [14] refines reachability-based approach so that reachable but non-accessed memory locations are additionally removed. The technique was successfully applied to interval-domain-based global static analysis [14]. These localization methods have a common limitation in handling procedure calls. The inefficient aspect, however, has not been well addressed in the literature. We believe the reason is two-folds: (1) because localization itself greatly improves global analysis performance, such 'small' inefficiency is often neglected; (2) the inefficiency only comes to the fore when we analyze programs that have complex procedural features such as large recursive call cycles. In this paper, we show that the problem is one key reason for why localization sometimes does not have satisfactory performance in practice, and propose a solution that extends the access-based localization technique.

Our technique can be considered as a lightweight sparse analysis. While traditional flow-sensitive analysis propagates information along control flow paths, sparse analysis [20,21,16] uses def-use chains and directly propagate data from definition point to its use points, by which unnecessary computational cost is reduced. Our technique is similar to sparse analysis in that we sometimes bypass data, not propagating them along usual control flow paths. Moreover, the concept of delivery points in section 3.3 is similar to ϕ-functions of SSA-based sparse analysis [4,5] in that both reduces the number of additional join operations. However, we do not require def-use chains to be computed in both analysis and computing delivery points, which is the main challenge of sparse analysis in the presence of pointers [4,5].

6 Conclusion

We presented a new technique to mitigate a performance problem of access-based localization technique. Our technique enables access-based localization to efficiently handle complex procedure calls such as recursive cycles. Our technique is general in that it is applicable to any analysis problems that use access-based localization. We proved the effectiveness of our technique by experiments with a realistic global C static analyzer on a variety of open-source benchmarks.

Acknowledgement. We thank Lucas Brutschy for his helpful comments and suggestions on the early development of the idea. 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 2011-0000971) and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2011.

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation, pp. 196–207 (2003)
2. Cousot, P., Cousot, R.: 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, pp. 238–252 (1977)
3. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Proceedings of the International Symposium on Static Analysis, pp. 240–260 (2006)
4. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 226–238 (2009)
5. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the Symposium on Code Generation and Optimization (2011)
6. Harrison III, W.L.: The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urabana-Champaign (February1989)
7. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 329–341 (1998)
8. Jhee, Y., Jin, M., Jung, Y., Kim, D., Kong, S., Lee, H., Oh, H., Park, D., Yi, K.: 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`
9. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: Proceedings of the International Symposium on Static Analysis, pp. 203–217 (2005)

10. Jung, Y., Yi, K.: Practical memory leak detector based on parameterized procedural summaries. In: Proceedings of the International Symposium on Memory Management, pp. 131–140 (2008)
11. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: Proceedings of the International Conference on Compiler Construction, pp. 245–259 (2008)
12. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming, pp. 13–25 (2006)
13. Oh, H.: Large Spurious Cycle in Global Static Analyses and its Algorithmic Mitigation. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 14–29. Springer, Heidelberg (2009)
14. Oh, H., Brutschy, L., Yi, K.: Access Analysis-based Tight Localization of Abstract Memories. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 356–370. Springer, Heidelberg (2011)
15. Oh, H., Yi, K.: An algorithmic mitigation of large spurious interprocedural cycles in static analysis. Software: Practice and Experience 40(8), 585–603 (2010)
16. Reif, J.H., Lewis, H.R.: Symbolic evaluation and the global value graph. In: Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 104–118 (1977)
17. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61 (1995)
18. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 296–309 (2005)
19. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Proceedings of the International Symposium on Static Analysis, pp. 284–302 (2005)
20. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 12–27. ACM, New York (1988)
21. Wegman, M.N., Kenneth Zadeck, F.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. 13, 181–210 (1991)
22. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
23. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.: On scalable shape analysis. Technical Memorandum RR-07-10, Queen Mary University of London, Department of Computer Science (November 2007)