Access Analysis-Based Tight Localization of Abstract Memories

Hakjoo Oh[1], Lucas Brutschy[2], and Kwangkeun Yi[1]

[1] Seoul National University
[2] RWTH Aachen University

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. Our technique first estimates, by an efficient pre-analysis, the set of locations that 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. Localizing more frequently such as at loop bodies and basic blocks as well as procedure bodies, the generalized localization additionally reduces analysis time by an average of 31.8%.

1 Introduction

In global abstract interpretation of imperative programs, memory localization (sometimes called "abstract garbage collection" or "framing") is vital for reducing analysis cost [5, 8, 14, 16, 24]. Not to mention the immediate benefit of the reduced memory footprint, memory localization has other important impact on cost reduction. In flow-sensitive, semantically dense 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-computations 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.

The conventional localization scheme is reachability-based [5, 23, 24, 8, 21, 20, 14, 16]. When applied to procedure bodies, reachability-based approaches allow a procedure to be analyzed with only a memory portion that is reachable from

Table 1. Reachability-based approach is too conservative. The table shows a comparison of accessed and reachable (abstract) memory portions during abstract interpretations of 5 open-source programs. For each a/b (r%), a is the average number of memory entries accessed in the called procedures, b is the average size of the reachable input state, and r is their ratio.

Program	LOC	accessed memory / reachable memory
spell-1.0	2,213	5 / 453 (1.1%)
gzip-1.2.4a	7,327	22 / 1,002 (2.2%)
jwhois-3.0.1	9,344	28 / 830 (3.4%)
bc-1.06	13,093	24 / 824 (2.9%)
less-290	18,449	86 / 1,546 (5.6%)

actual parameters or global variables. Because the reachable portion is often smaller than the entire memory state, analysis cost is reduced, both in time and memory. The method is popular in various kinds of program analysis: for example, in shape analysis [23, 8, 21, 14, 6] and higher-order flow analysis [16, 5].

However, the reachability-based approach has inefficient aspects especially in analyzing real C programs. 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.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 [11–13, 18, 19] that uses the reachability-based localization.

In this paper, 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 memory entries that are reachable but possibly not accessed. The main problem is to find the memory parts that will be accessed during the analysis of a block *before* actually analyzing the block. We solve the problem 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 filters out memory parts that will not be accessed within the block. The pre-analysis applies a conservative abstraction to the abstract semantics of the original analysis and quickly finds an over-approximation of resources that the actual analysis requires. This over-approximate nature of our pre-analysis ensures the correctness of our approach (Section 3.2).

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 [11–13, 18, 19]. In experiments, the 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–100KLOC). 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.

We generalize the idea of localization at procedure entries to localization of arbitrary code blocks. 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.

This paper makes the following contributions.

– We present a new localization technique, access-based localization. We employ a pre-analysis that is a conservative abstraction of the abstract semantics of the actual analysis. As far as we know published program analyzers do not perform access-based localization: previous analyses use reachability-based techniques (e.g., [16, 8, 20]) or their variants (e.g., [5, 15]).
– We present a generalized localization algorithm that applies to arbitrary code blocks as well as procedure bodies. As far as we know, other published program analyzers apply localization only to procedures.
– We prove the effectiveness of our technique by experiments with industrial-strength C static analyzer [11–13, 18, 19].

Example 1. 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 parameter 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.

```
1: struct S { int a; int b; }
2: int g = 0;
3: void f (S* p) { p->a = 1; }
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
```

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

(a) Non-localized memory

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

(b) Reachability-based localization

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

(c) Access-based localization

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

Outline. Section 2 presents our analysis framework. Section 3 develops our approach on top of our analysis framework. Section 4 shows experimental results. Section 5 presents related work and discussion.

2 Setting: Baseline Analyzer

We describe our localization technique on top of a flow-sensitive and context-insensitive abstract interpreter of C programs, based on the interval abstract domain. We present the representation of programs (Section 2), abstract domain, and abstract semantics (Section 2).

Our abstract domain and semantics are rather conventional, similar to ones used in other abstract interpretations for C or binary programs (e.g., [2]). Our abstract semantics estimate numeric and pointer values within a monolithic abstract interpretation.

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

$$\texttt{set(}lv\texttt{,}e\texttt{)} \mid \texttt{alloc(}lv\texttt{,}a\texttt{)} \mid \texttt{call(}f_x\texttt{,}e\texttt{)}$$

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

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

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. All program variables, including formal parameters, have unique names. The command `set(`lv`,`e`)` assigns the value of e into the location of lv. The command `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 of the allocation site. For simplicity, we only consider structures with one field in this explanation.

A call-site in a program is represented by a call node and its corresponding return node. A call node `call(`f_x`,`e`)` indicates that it invokes a procedure f, its formal parameter is x, and actual parameter is e. For simplicity, we assume that there are no function pointers and only consider procedures with one parameter. Edges are assembled by two functions $\mathsf{predof} \in \mathit{Node} \to 2^{\mathit{Node}}$ and $\mathsf{succof} \in \mathit{Node} \to 2^{\mathit{Node}}$, which map each node to its predecessors and successors, respectively.

Static Analysis. In our analysis, the set of (possibly infinite) concrete memory states are represented by an abstract memory state $\hat{\mathit{Mem}} = \hat{\mathit{Addr}} \xrightarrow{\text{fin}} \hat{\mathit{Val}}$, denoting a finite map from abstract locations ($\hat{\mathit{Addr}}$) to the abstract values ($\hat{\mathit{Val}}$).

$$\begin{aligned}
\hat{\mathit{Addr}} &= \mathit{Var} + \mathit{AllocSite} + \mathit{AllocSite} \times \mathit{FieldName} \\
\hat{\mathit{Val}} &= \hat{\mathbb{Z}} \times 2^{\hat{\mathit{Addr}}} \times 2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}} \times 2^{\mathit{AllocSite} \times 2^{\mathit{FieldName}}} \\
\hat{\mathbb{Z}} &= \{\bot\} \cup \{[l,u] \mid l \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge l \le u\}
\end{aligned}$$

An abstract location may be a program variable (Var), an allocation site ($\mathit{AllocSite}$), or a structure field ($\mathit{AllocSite} \times \mathit{FieldName}$). All elements of an array allocated at l are abstracted by l. The abstract location for field x of a structure allocated at l is represented by $\langle l, x\rangle$ (the analysis is field-sensitive). An abstract value is a quadruple. Numeric values are tracked by the interval values ($\hat{\mathbb{Z}}$). Points-to information is kept by the second component ($2^{\hat{\mathit{Addr}}}$): it indicates pointer targets an abstract locations may point to. Allocated arrays of memory locations are represented by array blocks ($2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$): an array block $\langle l, o, s\rangle$ consists of abstract base address (l), offset (o), and size (s). A structure block $\langle l, \{x\}\rangle \in 2^{\mathit{AllocSite} \times 2^{\mathit{FieldName}}}$ abstracts structure values that are allocated at l and have a set of fields $\{x\}$.

We first define two functions $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$ that compute abstract values and locations, respectively. Given an expression e and an abstract memory state $\hat{m}$, $\hat{\mathcal{V}}(\in e \to \hat{\mathit{Mem}} \to \hat{\mathit{Val}})$ evaluates the abstract value of e under $\hat{m}$. Similarly, $\hat{\mathcal{L}}(\in lv \to \hat{\mathit{Mem}} \to 2^{\hat{\mathit{Addr}}})$ evaluates the set of abstract locations of lv under $\hat{m}$.

$$\begin{array}{rlrl}
\hat{\mathcal{V}}(n)(\hat{m}) &= \langle [n,n], \bot, \bot, \bot\rangle & \hat{\mathcal{L}}(x)(\hat{m}) &= \{x\} \\
\hat{\mathcal{V}}(e_1 \texttt{ + } e_2)(\hat{m}) &= \hat{\mathcal{V}}(e_1)(\hat{m}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{m}) & \hat{\mathcal{L}}(\texttt{*}e)(\hat{m}) &= \hat{\mathcal{V}}(e)(\hat{m}).2 \cup \{l \mid \langle l,o,s\rangle \in \hat{\mathcal{V}}(e)(\hat{m}).3\} \\
\hat{\mathcal{V}}(lv)(\hat{m}) &= \bigsqcup\{\hat{m}(\hat{\mathit{Addr}}) \mid & & \cup\{\langle l,x\rangle \mid \langle l,\{x\}\rangle \in \hat{\mathcal{V}}(e)(\hat{m}).4\} \\
 & \quad \hat{\mathit{Addr}} \in \hat{\mathcal{L}}(lv)(\hat{m})\} & \hat{\mathcal{L}}(e_1\texttt{[}e_2\texttt{]})(\hat{m}) &= \{l \mid \langle l,o,s\rangle \in \hat{\mathcal{V}}(e_1)(\hat{m}).3\} \\
\hat{\mathcal{V}}(\&lv)(\hat{m}) &= \langle \bot, \hat{\mathcal{L}}(lv)(\hat{m}), \bot, \bot\rangle & \hat{\mathcal{L}}(e\texttt{.}x)(\hat{m}) &= \{\langle l,x\rangle \mid \langle l,\{x\}\rangle \in \hat{\mathcal{V}}(e)(\hat{m}).4\}
\end{array}$$

where, $\hat{\mathcal{V}}(e)(\hat{m}).n$ indicates the n-th element of the tuple that $\hat{\mathcal{V}}(e)(\hat{m})$ evaluates. We skip the conventional definition of the abstract binary ($\hat{+}$) and join ($\sqcup$) operations. In our analysis, all of the array elements are smashed into a single element, and hence, the definition of $\hat{\mathcal{L}}(e_1\texttt{[}e_2\texttt{]})$ does not involve e_2.

For each node n, we define a transfer function $\hat{f} : \mathit{Node} \rightarrow \hat{\mathit{Mem}} \rightarrow \hat{\mathit{Mem}}$ that, given an input memory state, computes the effect of the command in node n on the input state :

$$\hat{f}\ n\ \hat{m} = \begin{cases} \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \hat{\mathcal{L}}(lv)(\hat{m})\} & \text{if } n = \texttt{set(}lv\texttt{,}e\texttt{)} \\ \hat{m}\{\langle\bot,\bot,\{\langle l,[0,0],\hat{\mathcal{V}}(e)(\hat{m}).1\rangle\},\bot\rangle /\!\!/ \hat{\mathcal{L}}(lv)(\hat{m})\} & \text{if } n = \texttt{alloc(}lv\texttt{,[}e\texttt{]}_l\texttt{)} \\ \hat{m}\{\langle\bot,\bot,\bot,\{\langle l,\{x\}\rangle\}\rangle /\!\!/ \hat{\mathcal{L}}(lv)(\hat{m})\} & \text{if } n = \texttt{alloc(}lv\texttt{,}\{x\}_l\texttt{)} \\ \hat{m}\{\hat{\mathcal{V}}(e)(\hat{m}) /\!\!/ \hat{\mathcal{L}}(x)(\hat{m})\} & \text{if } n = \texttt{call(}f_x\texttt{,}e\texttt{)} \end{cases}$$

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 $\texttt{set(}lv\texttt{,}e\texttt{)}$ is to (weakly) assign the abstract value of e into the locations in $\hat{\mathcal{L}}(lv)(\hat{m})$[1]. The array allocation command $\texttt{alloc(}lv\texttt{,[}e\texttt{]}_l\texttt{)}$ creates a new array block with offset 0 and size e. The structure block command $\texttt{alloc(}lv\texttt{,}\{x\}_l\texttt{)}$ creates a new structure block. In both cases, we use the allocation site l as a base address, which means that many, possibly infinite, concrete locations are summarized by finite abstract locations. The call node command $\texttt{call(}f_x\texttt{,}e\texttt{)}$ 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.

Airac computes a fixpoint table $\mathcal{T} \in \mathit{Node} \rightarrow \hat{\mathit{Mem}}$ that maps each node in the program to its output abstract memory state. The abstract memory state at each program point approximates all the concrete memory states occurring at the node in the concrete executions. The map is defined by the least fixpoint of the following function:

$$\hat{\mathcal{F}} : (\mathit{Node} \rightarrow \hat{\mathit{Mem}}) \rightarrow (\mathit{Node} \rightarrow \hat{\mathit{Mem}})$$

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

Fig. 2(a) (without the shaded line) shows the worklist-based fixpoint algorithm. In addition, we use widening and narrowing [7], the worklist order is weak topological ordering [4].

3 Memory Localization by Access Analysis

This section describes our localization technique. We first describe localization for procedure bodies: Section 3.1 develops the reachability-based localization on top of our analysis framework (Section 2) and Section 3.2 extends it to our access-based approach for procedures. We then generalize the procedure-level localization for arbitrary code blocks in Section 3.3.

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

3.1 Conventional Reachability-Based Localization for Procedures

We first formalize the reachability-based approach on top of our baseline analyzer Airac. We call our analyzer based on this approach $\mathsf{Airac}_{\mathsf{Reach}}$.

When calling a procedure, $\mathsf{Airac}_{\mathsf{Reach}}$ passes the memory parts that are reachable from global variables or parameters. Formally, given a call node $\mathtt{call}(f_x, e)$ and its input memory state $\hat{m}$ (parameter-bound), $\mathsf{Airac}_{\mathsf{Reach}}$ computes the following set of abstract locations (let $\mathsf{Globals}$ be the set of global variables in the program):

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

We use $\mathsf{Reach}(X, \hat{m})$ to denote the set of abstract locations in $\hat{m}$ that are (directly or transitively) reachable from a location set X.

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

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

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

Given an input memory $\hat{m}$ to a call node $\mathtt{call}(f_x, e)$, the definition of the transfer function $\hat{f}$ is changed as follows:

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

We also have to consider procedure returns. When a procedure returns to a return node, in order to recover the local information from the corresponding call node, we combine the returned memory with the memory parts that were not propagated to called procedures from the call node. Note that the issue of cutpoints [20] is not involved in our analysis because our semantics is store-based and every object is represented by a fixed location.

3.2 Access-Based Localization for Procedures

For performing the localization more aggressively, we separate the entire analysis into two phases: (1) the set of abstract locations that are accessed by a procedure during actual analysis is conservatively estimated by a pre-analysis; (2) then, the actual analysis uses the access-information and filters out memory entries that will definitely not be accessed by called procedures. The pre-analysis is derived from the abstract semantics of the original analysis by applying conservative abstractions. We call our analyzer based on the new technique $\mathsf{Airac}_{\mathsf{ProcAcc}}$.

Pre-analysis. For ensuring the correctness of the actual analysis, the pre-analysis should be an over-approximation of the actual analysis. The pre-analysis should not only find locations that are accessed in real executions, but find a set of abstract locations that contains all locations required by the actual analysis. For example, consider an expression `*p`. Suppose `p` points to a variable `a` during real execution. Suppose further, during actual analysis, `p` points to variables `a` and `b`, where `b` is a by-product of abstraction. In this case, our pre-analysis results should contain both a and b because both locations will be accessed in actual analysis. Hence we derive the pre-analysis from actual analysis of interest rather than using a separate pre-analysis such as an existing pointer analysis.

Our pre-analysis computes a map access $\in \mathit{ProcId} \to 2^{\hat{Addr}}$ that maps each procedure to a set of abstract locations that are possibly accessed (directly or transitively) during the actual analysis of the procedure body. In order to compute such a map, we first instrument the analysis (Section 2) so that accessed locations are collected during the course of the analysis. Then the abstract semantics of the analysis is conservatively abstracted to a less precise but more efficient analysis, the pre-analysis.

Fig. 2(a) shows how we collect abstract locations during the original analysis. Without the shaded line, the algorithm is a normal fixpoint algorithm. With the shaded line, the algorithm performs an analysis recording accessed locations. After the effect of a node n is computed (using $\hat{f}$), the abstract locations that are accessed during the evaluation of n are collected by collect. Throughout the analysis, the accessed locations for n are accumulated in $\mathcal{A}(n)$. When the analysis terminates, all the abstract locations that have been accessed during the analyses of n are collected by $\mathcal{A}(n)$.

In order to define function collect, we define two functions $\hat{\mathcal{AV}} \in e \to \hat{Mem} \to 2^{\hat{Addr}}$ and $\hat{\mathcal{AL}} \in lv \to \hat{Mem} \to 2^{\hat{Addr}}$. Given an expression e (resp., lv) and a memory state $\hat{m}$, $\hat{\mathcal{AV}}(e)(\hat{m})$ (resp., $\hat{\mathcal{AL}}(lv)(\hat{m})$) collects abstract locations that are accessed during the evaluation of $\hat{\mathcal{V}}(e)(\hat{m})$ (resp., $\hat{\mathcal{L}}(lv)(\hat{m})$). $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$ are naturally derived by examining the definition of $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$.

$$\begin{array}{rlrl}
\hat{\mathcal{AV}}(n)(\hat{m}) &= \emptyset & \hat{\mathcal{AL}}(x)(\hat{m}) &= \emptyset \\
\hat{\mathcal{AV}}(e_1 \texttt{ + } e_2)(\hat{m}) &= \hat{\mathcal{AV}}(e_1)(\hat{m}) \cup \hat{\mathcal{AV}}(e_2)(\hat{m}) & \hat{\mathcal{AL}}(*e)(\hat{m}) &= \hat{\mathcal{AV}}(e)(\hat{m}) \\
\hat{\mathcal{AV}}(lv)(\hat{m}) &= \hat{\mathcal{AL}}(lv)(\hat{m}) \cup \hat{\mathcal{L}}(lv)(\hat{m}) & \hat{\mathcal{AL}}(e_1\texttt{[}e_2\texttt{]})(\hat{m}) &= \hat{\mathcal{AV}}(e_1)(\hat{m}) \\
\hat{\mathcal{AV}}(\&lv)(\hat{m}) &= \hat{\mathcal{AL}}(lv)(\hat{m}) & \hat{\mathcal{AL}}(e\texttt{.}x)(\hat{m}) &= \hat{\mathcal{AV}}(e)(\hat{m})
\end{array}$$

Consider $\hat{\mathcal{AV}}$ (defined in the left column) first. When $e = n$, we see that the definition of $\hat{\mathcal{V}}$ does not read (nor write to) any location, and hence there are no accessed locations ($\emptyset$). When $e = e_1 \texttt{ + } e_2$, accessed locations are just collected recursively. When an l-value lv is used as an r-value (the third case), from the definition of $\hat{\mathcal{V}}(lv)(\hat{m})$, we see that abstract locations of lv and abstract locations that are accessed during the evaluation of $\hat{\mathcal{L}}(lv)(\hat{m})$ are accessed, which are collected by $\hat{\mathcal{L}}(lv)(\hat{m})$ and $\hat{\mathcal{AL}}(lv)(\hat{m})$, respectively. When an l-value lv is used as an address-of expression (the last case), from the definition of $\hat{\mathcal{V}}(\&lv)(\hat{m})$, we see that abstract locations that lv denotes ($\hat{\mathcal{L}}(lv)(\hat{m})$) are not accessed during the evaluation of $\hat{\mathcal{V}}(\&lv)(\hat{m})$ and hence the fourth case only includes $\hat{\mathcal{AL}}(lv)(\hat{m})$.

Similarly, the definition of $\hat{\mathcal{AL}}$ (defined in the right column) is derived from the definition of $\hat{\mathcal{L}}$. For example, $\hat{\mathcal{AL}}(x)(\hat{m})$ is $\emptyset$ because $\hat{\mathcal{L}}(x)(\hat{m})$ just produces a location x but does not read (resp., write) any value from (resp., to) x. Note that, when $lv = e_1\texttt{[}e_2\texttt{]}$ (the third case), we do not collect the locations accessed during the evaluation of e_1 because the definition of $\hat{\mathcal{L}}$ for this case does not involve e_2 (array elements are smashed into a single element in the analysis).

```
$\mathcal{W} \in \mathit{Worklist} = 2^{\mathit{Node}}$
$\mathcal{T} \in \text{Table} = \mathit{Node} \to \hat{\mathit{Mem}}$
$\hat{f} \in \mathit{Node} \to \hat{\mathit{Mem}} \to \hat{\mathit{Mem}}$
$\mathcal{A} \in \mathit{Node} \to 2^{\hat{\mathit{Addr}}}$

$\mathit{FixpointIterate}\ (\mathcal{W}, \mathcal{T}) =$
$\mathcal{W} := \mathit{Node}$
$\mathcal{T} := \lambda n.\bot_{\hat{\mathit{Mem}}}$
repeat
  $n := \mathsf{choose}(\mathcal{W})$
  $m := \hat{f}\ n\ (\bigsqcup_{p \in \mathsf{predof}(n)} \mathcal{T}(p))$
  $\mathcal{A}(n) := \mathcal{A}(n) \cup \mathsf{collect}(n, \bigsqcup_{p \in \mathsf{predof}(n)} \mathcal{T}(p))$
  if $m \not\sqsubseteq \mathcal{T}(n)$
    $\mathcal{W} := \mathcal{W} \cup \mathsf{succof}(n)$
    $\mathcal{T}(n) := \mathcal{T}(n) \sqcup m$
until $\mathcal{W} = \emptyset$
return $\mathcal{T}$
```

(a) The (worklist-based) analysis algorithm

```
$\mathit{old}, \mathit{new} \in \hat{\mathit{Mem}}$
$\hat{f}' \in \mathit{Node} \to \hat{\mathit{Mem}} \to \hat{\mathit{Mem}}$
$\mathcal{A} \in \mathit{Node} \to 2^{\hat{\mathit{Addr}}}$

$\mathit{Preliminary}\ (\mathit{old}, \mathit{new}) =$
$\mathit{new} := \bot_{\hat{\mathit{Mem}}}$
repeat
  $\mathit{old} := \mathit{new}$
  for all $n \in \mathit{Node}$ do
    $\mathit{new} := \hat{f}'\ n\ \mathit{new}$
    $\mathcal{A}(n) := \mathcal{A}(n) \cup \mathsf{collect}(n, \mathit{new})$
until $\mathit{new} \sqsubseteq \mathit{old}$
return $\mathit{new}$
```

(b) The pre-analysis algorithm

Fig. 2. (a) Collecting accessed abstract locations during a (worklist-based) fixpoint computation. Without the shaded line, it shows a normal worklist algorithm. The shaded line collects abstract locations that are accessed by node n currently being analyzed. (b) The pre-analysis performs flow-insensitive fixpoint computation.

Using $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$, function collect collects abstract locations that are accessed during the analysis of each command. collect is derived from the definition of the transfer function $\hat{f}$, which is similar to the derivation of $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$.

$$\mathsf{collect}(n, \hat{m}) = \begin{cases} \hat{\mathcal{AL}}(lv)(\hat{m}) \cup \hat{\mathcal{AV}}(e)(\hat{m}) \cup \hat{\mathcal{L}}(lv)(\hat{m}) & \text{if } n = \texttt{set(}lv\texttt{,}e\texttt{)} \\ \hat{\mathcal{AL}}(lv)(\hat{m}) \cup \hat{\mathcal{AV}}(e)(\hat{m}) \cup \hat{\mathcal{L}}(lv)(\hat{m}) & \text{if } n = \texttt{alloc(}lv\texttt{,[}e\texttt{]}_l\texttt{)} \\ \hat{\mathcal{AL}}(lv)(\hat{m}) \cup \hat{\mathcal{L}}(lv)(\hat{m}) & \text{if } n = \texttt{alloc(}lv\texttt{,}\{x\}_l\texttt{)} \\ \hat{\mathcal{AV}}(e)(\hat{m}) \cup \hat{\mathcal{L}}(x)(\hat{m}) & \text{if } n = \texttt{call(}f_x\texttt{,}e\texttt{)} \end{cases}$$

The problem now is to collect accessed locations in an efficient way. If we used the algorithm of Fig. 2(a) as our pre-analysis, the pre-analysis would take more time than the actual analysis. In order to get a more efficient pre-analysis, we apply the following two abstractions to the analysis of Fig. 2(a): (1) we ignore the orders of program statements, that is, we perform a flow-insensitive analysis; (2) We ignore parts of the semantics that are not directly related to computing abstract locations. In the abstract value $\hat{Val}$, for example, the interval values ($\hat{\mathbb{Z}}$) have nothing to do with abstract locations. The flow function ($\hat{f}$) and abstract evaluation $\hat{\mathcal{V}}$ are changed so that interval values are not computed any more. We call the changed functions $\hat{f}'$ and $\hat{\mathcal{V}}'$, respectively.

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

Fig. 2(b) shows our pre-analysis algorithm. It uses a flow-insensitive fixpoint computation. The analysis starts with a bottom memory state ($\bot_{\hat{Mem}}$). The state is iteratively updated by flow functions for all nodes in the program until the resulting state is subsumed by the state of the previous iteration. After the effect of a node n is computed (using $\hat{f}'$), the abstract locations that are accessed during the evaluation of n are collected using function $\mathsf{collect}$.

The set $\mathsf{access}(f)$ of abstract locations that are (directly or transitively) accessed by procedure f is defined as follows:

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

where, $\mathsf{callees}(f)$ 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.

Actual Analysis. The actual analysis is the same as $\mathsf{Airac}_{\mathsf{Reach}}$ except that $\mathsf{Airac}_{\mathsf{ProcAcc}}$ uses access information (access) and additionally excludes non-accessed memory parts. Given an input memory state $\hat{m}$ to a call node $\mathtt{call}(f_x,e)$, reachable locations $\mathcal{R}(f_x, \hat{m})$, and accessed locations $\mathsf{access}(f)$, the transfer function $\hat{f}$ for the call statement $\mathtt{call}(f_x,e)$ is changed as follows:

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

After parameter binding ($\hat{m}'$) the memory is first restricted to the reachable locations ($\mathcal{R}(f_x, \hat{m}')$) and then the resulting memory is restricted to $\mathsf{access}(f)$. The reason why we restrict the memory to $\mathcal{R}(f_x, \hat{m}') \cap \mathsf{access}(f)$ is that $\mathsf{access}(f)$ may have locations that are unreachable, i.e., not contained in $\mathcal{R}(f_x, \hat{m}')$, because our pre-analysis is less precise than the actual analysis. Hence, the memory states localized by our approach are always smaller than or equal to those localized by the reachability-based approach. Procedure returns are handled as in $\mathsf{Airac}_{\mathsf{Reach}}$.

3.3 Access-Based Localization for Arbitrary Code Blocks

We generalize the access-based, procedural localization ($\mathsf{Airac}_{\mathsf{ProcAcc}}$) for code blocks smaller than procedures. Given a code block, it is straightforward to collect accessed locations for the block because our pre-analysis provides access information ($\mathcal{A}$ in Fig. 2) for each node in the control flow graph. 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-computations 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 select code blocks $\langle entry, exit, B\rangle$ that consists of one *entry* node, one *exit* node, and a selected block B that satisfy 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.

4 Experiments

We check the performance of our new localization technique by experiments with Airac, a global abstract interpretation engine in an industrialized bug-finding analyzer [11–13, 18, 19]. Because we focus on comparing the performance between different localization schemes, we disabled some improvement techniques for Airac. Specifically, we did not use context pruning, narrowing, and return-site sensitivity [18, 19]. These techniques improve analysis' precision and speed but make it hard to only measure the net effect of respective localization schemes.

From our baseline analyzer Airac, which does not use localization, we have made three analyzers $\text{Airac}_{\text{Reach}}$, $\text{Airac}_{\text{ProcAcc}}$, and $\text{Airac}_{\text{GenAcc}}$ that respectively use procedure-level reachability-based, procedure-level access-based, and generalized access-based localization and differ from Airac only in their respective localization schemes. Hence, performance differences, if any, are solely attributed to the different localization methods. We set the minimum block size k to 6 for $\text{Airac}_{\text{GenAcc}}$, which was shown to be most efficient in our setting. The analyzers are written in OCaml.

We have analyzed 15 software packages. Fig. 3 shows our benchmark programs. 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 three performance measures: (1) *#iters* is the total number of iterations during the worklist algorithm (the number of iterations of the outside loop in Fig. 2(a); (2) *time* is the CPU time spent; (3) *MB* is the peak memory consumption.

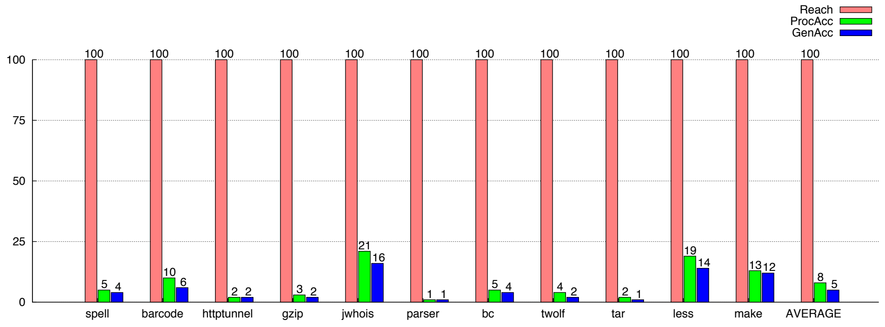
Airac vs. $\text{Airac}_{\text{Reach}}$: The results show that the reachability-based localization reduces the analysis time and memory for most programs. $\text{Airac}_{\text{Reach}}$ consistently reduces *#iters* of Airac by 54.9% on average and reduces the analysis time by 36.7% on average. The effectiveness 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`. However, for some programs (`spell` and `make`), $\text{Airac}_{\text{Reach}}$ took

Program	**LOC**	**Proc**	**BB**	Airac (w.o. localization)			Airac$_{\text{Reach}}$			
				#iters	*time*(sec)	*MB*	*#iters*	*time*(sec)	*MB*	$Save_1$
spell-1.0	2,213	31	782	37,085	46.1	29	23,249	53.0	23	-15.0%
barcode-0.96	4,460	57	2,634	38,742	105.7	291	17,997	92.6	125	12.4%
httptunnel-3.3	6,174	110	2,757	444,354	2808.9	284	201,046	1383.2	154	50.8%
gzip-1.2.4a	7,327	135	6,271	1,327,464	12,756.2	886	393,338	2,866.6	333	77.5%
jwhois-3.0.1	9,344	73	5,147	428,584	3,424.5	633	198,249	1,185.4	254	65.4%
parser	10,900	325	9,298	5,707,185	196,318.8	2,917	2,327,303	60,577.8	1,048	69.1%
bc-1.06	13,093	134	4,924	5,677,277	87,988.5	767	800,474	13,879.2	335	84.2%
twolf	19,700	222	14,610	∞	∞	∞	2,375,894	27,230.3	1,199	N/A
tar-1.13	20,258	222	10,800	6,244,121	157,545.0	2,916	3,819,726	113,061.4	1,797	28.2%
less-382	23,822	382	10,056	7,654,188	148,015.7	2,445	2,998,969	137,827.3	1,480	6.9%
make-3.76.1	27,304	191	11,061	6,162,145	126,908.8	2,757	4,013,647	142,325.6	1,954	-12.1%
wget-1.9	35,018	434	16,544	∞	∞	∞	∞	∞	∞	N/A
screen-4.0.2	44,734	589	31,792	∞	∞	∞	∞	∞	∞	N/A
bison-2.4	56,361	1,203	20,781	∞	∞	∞	∞	∞	∞	N/A
bash-2.05a	105,174	959	28,675	∞	∞	∞	∞	∞	∞	N/A

(a) Properties of the benchmarks and analysis results for Airac and Airac$_{\text{Reach}}$

Program	Airac$_{\text{ProcAcc}}$				Airac$_{\text{GenAcc}}$($k = 6$)			
	#iters	*time*:total(pre)	*MB*	$Save_2$	*#iters*	*time*:total(pre)	*MB*	$Save_3$
spell-1.0	5,512	2.4 (0.2)	5	**95.4%**	4,857	2.1 (0.3)	5	**13.5%**
barcode-0.96	9,433	9.4 (0.6)	25	**89.8%**	7,213	5.5 (1.4)	21	**41.6%**
httptunnel-3.3	17,072	31.4 (1.3)	36	**97.7%**	14,824	21.6 (2.4)	24	**31.0%**
gzip-1.2.4a	78,471	94.8 (1.3)	73	**96.7%**	47,454	54.5 (8.0)	67	**42.5%**
jwhois-3.0.1	99,815	254.8 (1.2)	170	**78.5%**	73,000	188.1 (18.5)	148	**26.2%**
parser	206,173	890.0 (3.8)	224	**98.5%**	173,285	617.0 (9.2)	245	**30.7%**
bc-1.06	146,407	730.9 (4.1)	106	**94.7%**	123,398	542.7 (8.5)	116	**25.8%**
twolf	520,561	1,037.7 (7.5)	332	**96.2%**	337,837	480.4 (20.0)	260	**53.7%**
tar-1.13	360,009	2,524.0 (6.0)	338	**97.8%**	219,109	1,638.3 (15.9)	351	**35.1%**
less-382	1,223,535	26,817.6 (40.7)	466	**80.5%**	833,643	18,766.7 (95.0)	528	**30.0%**
make-3.76.1	1,149,151	19,015.2 (39.4)	580	**86.6%**	894,843	17,405.7 (75.9)	740	**8.5%**
wget-1.9	526,975	6,735.8 (20.8)	609	N/A	366,051	3,823.3 (48.5)	623	**43.2%**
screen-4.0.2	6,402,974	340,849.0 (281.5)	2,458	N/A	4,699,777	274,280.3 (667.1)	2,958	**19.5%**
bison-2.4	305,988	2,487.3 (13.1)	301	N/A	234,751	1,696.6 (37.7)	302	**31.8%**
bash-2.05a	379,429	2,011.3 (20.2)	439	N/A	251,175	1,142.7 (59.5)	416	**43.2%**

(b) Analysis results for Airac$_{\text{ProcAcc}}$ and Airac$_{\text{GenAcc}}$



(c) Comparison of analysis time among Airac$_{\text{Reach}}$, Airac$_{\text{ProcAcc}}$ and Airac$_{\text{GenAcc}}$.

Fig. 3. Lines of code (**LOC**) are given before preprocessing. The number of procedures (**Proc**), basic blocks in the supergraph (**BB**) in programs are given after preprocessing. *time* for Airac$_{\text{ProcAcc}}$ and Airac$_{\text{GenAcc}}$ is the total time that includes pre-analysis time. The pre-analysis time is shown inside parentheses. $Save_1$ shows time savings of Airac$_{\text{Reach}}$ against Airac. $Save_2$ shows time savings of Airac$_{\text{ProcAcc}}$ against Airac$_{\text{Reach}}$. $Save_3$ shows time savings of Airac$_{\text{GenAcc}}$ against Airac$_{\text{ProcAcc}}$. Entries with ∞ mean missing data because of the analysis running out of memory.

more time than Airac. This is mainly because of the overhead of localizing operations at procedure calls.

Airac$_{\mathsf{Reach}}$ vs. Airac$_{\mathsf{ProcAcc}}$: Overall, Airac$_{\mathsf{ProcAcc}}$ saved 78.5%–98.5%, on average 92.1%, of the analysis time of Airac$_{\mathsf{Reach}}$. The analysis time of Airac$_{\mathsf{ProcAcc}}$ includes pre-analysis time. The significant time savings are caused by the synergy between reduction (on average 74.3%) in the number of iterations (*#iters*) and improved speed (on average 4.0x) of memory operations. Iterations are reduced because Airac$_{\mathsf{ProcAcc}}$'s more general summaries more effectively avoid re-computation of procedures at different call-sites. Speed is improved because each procedure is analyzed with smaller memory states.

Moreover, our technique noticeably saves peak memory consumption by on average 71.2%. The reduction enabled Airac$_{\mathsf{ProcAcc}}$ to analyze the largest four programs (`wget`, `screen`, `bison`, `bash`) that cannot be analyzed by Airac$_{\mathsf{Reach}}$.

Airac$_{\mathsf{ProcAcc}}$ is at least as precise as Airac$_{\mathsf{Reach}}$. In principle, more aggressive localization improves precision of our analysis because unnecessary memory entries are not passed to procedures and needless widenings are avoided. In the experiments (similar to one performed in [18]), Airac$_{\mathsf{ProcAcc}}$'s precision was the same with Airac$_{\mathsf{Reach}}$ or slightly improved.

Airac$_{\mathsf{ProcAcc}}$ vs. Airac$_{\mathsf{GenAcc}}$: Airac$_{\mathsf{GenAcc}}$ additionally saved, on average 31.8%, of the analysis time of Airac$_{\mathsf{ProcAcc}}$. Memory costs between them is nearly the same (1.9% is reduced). Memory costs for Airac$_{\mathsf{GenAcc}}$ sometimes increase (e.g., `parser`), because we cache access sets for each localization block.

5 Related Work and Discussion

In static program analysis, localization is a well-known idea for reducing analysis cost, however, 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 analysis [6, 20, 21, 14, 8, 24, 23]. Rinetzky et al. [20, 21] define a shape analysis in which called procedures are only passed with reachable parts of the heap. Marron et al. [14] reformulate the idea of [20] for graph-based heap models. In separation-logic-based program verification (both by-hand and automatic checking [3]), 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 in accessibility in logic, is conventionally implemented based on reachability [8, 24, 23]: Gotsman et al. [8] and Yang et al. [24, 23] split states based on reachability. Similar reachability-based techniques are also popular in higher-order flow analyses [9, 10, 16]. Jagannathan et al. [10] use "an abstract form of garbage collection" that removes unreachable bindings. Might et al. [16] formalize the abstract-garbage-collecting control-flow analysis and show that removing unreachable cells significantly improves the analysis performance.In this paper, we present a new approach to localization that is access-based.

Chen et al. [5] 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 two times. 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. [15] observes 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.

We do not argue that our access-based localization is cost-effective in general. Our abstract domain (numeric intervals) is non-relational and, consequently, partitioning of the memory states is relatively simple. But, the partitioning operation will become costly when analysis' abstract semantics involves relational information such as in analysis with relational domain [17] or storeless semantics [20]. 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, localizing operation gets more complicated. We have a plan to investigate our technique for relational analysis.

It is also a well-known idea to scale an analysis by using an efficient pre-analysis. For example, flow-insensitive pre-analysis has been used in dataflow analysis [1], pointer analysis [22]. Our work is an instance of these lines of research: we use a pre-analysis to localize memory states in actual analysis.

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.

Acknowledgements. 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 2010-0001718) and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2010.

References

1. Adams, S., Ball, T., Das, M., Lerner, S., Rajamani, S.K., Seigle, M., Weimer, W.: Speeding up dataflow analysis using flow-insensitive pointer analysis. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 230–246. Springer, Heidelberg (2002)
2. Allamigeon, X., Godard, W., Hymans, C.: Static analysis of string manipulations in critical embedded C programs. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 35–51. Springer, Heidelberg (2006)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: APLAS, pp. 52–68 (2005)
4. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Int. Conf. on Formal Methods in Prog. and their Appl, pp. 128–141 (1993)

5. Chen, L., Harrison III, W.L.: An efficient approach to computing fixpoints for complex program analysis. In: Int. Conf. on Supercomp., pp. 98–106 (1994)
6. Chong, S., Rugina, R.: Static analysis of accessed regions in recursive data structures. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 463–482. Springer, Heidelberg (2003)
7. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
8. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
9. 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 (February 1989)
10. Jagannathan, S., Thiemann, P., Weeks, S., Wright, A.: Single and loving it: must-alias analysis for higher-order languages. In: POPL, pp. 329–341 (1998)
11. 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), `http://ropas.snu.ac.kr/ kwang/paper/30yai-08.pdf`
12. Jung, Y., Kim, J., Shin, J., Yi, K.: Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 203–217. Springer, Heidelberg (2005)
13. Jung, Y., Yi, K.: Practical memory leak detector based on parameterized procedural summaries. In: ISMM, pp. 131–140 (2008)
14. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 245–259. Springer, Heidelberg (2008)
15. Might, M., Chambers, B., Shivers, O.: Model checking via ΓCFA. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 59–73. Springer, Heidelberg (2007)
16. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: ICFP, pp. 13–25 (2006)
17. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
18. Oh, H.: Large spurious cycle in global static analyses and its algorithmic mitigation. In: APLAS (2009)
19. Oh, H., Yi, K.: An algorithmic mitigation of large spurious interprocedural cycles in static analysis. In: Software: Practice and Experience (2010)
20. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL, pp. 296–309 (2005)
21. Rinetzky, N., Sagiv, M., Yahav, E.: Interprocedural shape analysis for cutpoint-free programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 284–302. Springer, Heidelberg (2005)
22. Xu, G., Rountev, A., Sridharan, M.: Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 98–122. Springer, Heidelberg (2009)
23. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
24. 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)