
ROPAS
Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
ROPAS-2006-31
July 19, 2006

Experiment on a Modular Program Aanlysis That Saves Memory

Jaehwang Kim

School of Computer Science and Engineering
Seoul National University

Master thesis
Advisor: Kwangkeun Yi

Abstract

We present a modular fixpoint computation that consumes smaller memory than a global fixpoint iterations yet with no loss of accuracy. Our motivation comes from designing a scalable global program analysis. Program analysis is to compute a fixpoint of a system of equations. For large-scale global analysis for bug-finding or verification, analysis' memory consumption is more decisive factor for its usability than its time consumption. Our method is to re-arrange the original system of equations into two hierarchies. We partition the equations into smaller sets, and then we prepare a set of equations that "links" the partitions. After solving the set of link equations with nested fixpoint iterations, we solve each partition separately. If the size of link equations set is small, than we can solve each partition and the set of linker equations with smaller memories. This method is implemented inside a realistic C analyzer. From our experiment of analyzing 5,000LOC - 20,000LOC GNU programs, we observed our modular method have 50-71% memory peak saving.

1 Introduction

1.1 Motivation

Termination remains in theory if without scalability. From our experience in developing a sound program analyzer - Airac[14] for bug-finding in realistic setting, we have found that economical memory consumption is critical for its scalability. Though Airac was designed to terminate for any programs, if input programs had over 15,000LOC, Airac in its global analysis mode started to fail because of out-of-memory.

If we can solve a system of equations with less memory, we can make a more scalable program analyzer. It's because a typical program analyzer solves a system of equations which are derived from a given program. In case of flow-sensitive analysis, unknown variables in the system of equations correspond to program points of the program. The analyzer usually solves the system of equations by an iterative fixpoint algorithm. During fixpoint iterations, the analyzer keeps intermediate values of all unknown variables. Therefore amount of memory required for programs is determined mostly by programs' size.

Using Bekić's theorem[1, 5] we can partition a system of equations into independent sub systems. For example, two equations $x = f(y)$ and $y = g(x)$ can be separated to $x = f(\text{lfp}\lambda y.g(x))$ and $y = g(\text{lfp}\lambda x.f(y))$. Bekić's theorem ensures us that the solutions of the new equations are same with the solutions of the original equations.

But Bekić's theorem-based partitioning is not enough to guarantee memory saving. For the solution of the previous example, we compute $\text{lfp}\lambda x.\,\text{lfp}\lambda y.g(x)$ and $\text{lfp}\lambda y.\,\text{lfp}\lambda x.g(y)$ with

nested fixpoint iterations. To achieve a feasible memory-efficient modular program analysis, we have to be sure about two things: 1) We must be able to solve each module with less memory. In case of the previous example, we can compute $\mathrm{lfp}\lambda x.\,\mathrm{lfp}\lambda y.g(x)$ and $\mathrm{lfp}\lambda y.\,\mathrm{lfp}\lambda x.g(y)$ separately, but we can't save memory. Each of $\mathrm{lfp}\lambda x.\,\mathrm{lfp}\lambda y.g(x)$ and $\mathrm{lfp}\lambda y.\,\mathrm{lfp}\lambda x.g(y)$ requires memory cells to store x and y. So does solving $\{x = f(y),\ y = g(x)\}$; 2) There should be no serious overhead due to modularization. In $\mathrm{lfp}\lambda x.\,\mathrm{lfp}\lambda y.g(x)$ and $\mathrm{lfp}\lambda y.\,\mathrm{lfp}\lambda x.g(y)$ fixpoint iterations are nested. And the inner iterations always start with $\bot$. Therefore there are redundant computations.

This paper answers a question "Can we achieve a feasible memory-efficient modular program analysis using Bekić's theorem?". To answer the question, we made a modular program analyzer Kairac using Bekić's theorem. In the rest of this paper, we present our modular approach for solving systems of equations, designs of Kairac and experiment results.

1.2 Our Idea

We present our modular fixpoint computation by an example. Let's consider the following system E of equations:

$$E = \left\{ \begin{array}{lcllcl} x_1 & = & f_1\langle x_1, x_2\rangle & y_1 & = & g_1\langle y_1, y_2\rangle \\ x_2 & = & f_2\langle x_2, y_1\rangle & y_2 & = & g_2\langle y_2, y_1\rangle \end{array} \right\}$$

To solve E with an iterative fixpoint algorithm, we need 4 memory cells to store the 4 unknown variables. Note that no equation uses all 4 variables. And if we divide E into two parts - one for x_i and another for y_i, then only y_1 is shared by the two parts. We call y_1 a *link variable*, since the two parts are linked by y_1. So if we know y_1's solution, then we can get solutions of x_i and y_i separately with less than 4 memory cells. Using y_1 as a link variable, we can define a new system E' of equations by introducing a new variable z for the link variable:

$$\begin{array}{rcccccccc} E' & = & \left\{ \begin{array}{lcl} x_1 & = & f_1\langle x_1, x_2\rangle \\ x_2 & = & f_2\langle x_2, z\rangle \end{array} \right\} & \cup & \left\{ \begin{array}{lcl} y_1 & = & g_1\langle z, y_2\rangle \\ y_2 & = & g_2\langle y_2, z\rangle \end{array} \right\} & \cup & \left\{ \begin{array}{lcl} z & = & y_1 \end{array} \right\} \\ & = & \{\mathbf{x} = F\langle \mathbf{x}, z\rangle\} & \cup & \{\mathbf{y} = G\langle \mathbf{y}, z\rangle\} & \cup & \{z = \mathbf{y}.1\} \end{array}$$

$\mathbf{x} = \langle x_1, x_2\rangle$, $\mathbf{y} = \langle y_1, y_2\rangle$ and $\mathbf{v}.n$ is the n-th component of vector $\mathbf{v}$. $F\langle \mathbf{x}, z\rangle = \langle f_1\mathbf{x}, f_2\langle \mathbf{x}.2, z\rangle\rangle$ and $G\langle \mathbf{y}, z\rangle = \langle g_1\langle z, \mathbf{y}.2\rangle, g_2\langle \mathbf{y}.2, z\rangle\rangle$. To get the solutions of $\mathbf{x}$ and $\mathbf{y}$ separately, we have to solve the new equation $z = \mathbf{y}.1$ first. For the solution of z, we define a *link equation* which consists of just link variables. By Bekić's theorem, we can replace $\mathbf{y}$ by $\mathrm{lfp}\lambda\mathbf{y}.G\langle \mathbf{y}, z\rangle$ without loss of accuracy. Therefore we have the following equation and its solution:

$$\begin{array}{rcll} z & = & Hz & \text{(equation)} \\ z & \stackrel{\triangle}{=} & \mathrm{lfp}H & \text{(solution)} \end{array}$$

where $H = \lambda z.(\mathrm{lfp}(G'z)).1$ and $G' = \lambda z.\lambda\mathbf{y}.G\langle \mathbf{y}, z\rangle$. Now note that we can solve the link equation by computing $\mathrm{lfp}H$ with 3 memory cells. At each iteration step for $\mathrm{lfp}H$, $\mathrm{lfp}(G'z)$ uses 1 memory cell for z and 2 memory cells for $\mathbf{y}$. With $\mathrm{lfp}H$, we can get the solutions of $\mathbf{x}$ and $\mathbf{y}$ by solving $\{\mathbf{x} = F\langle \mathbf{x}, z\rangle,\ z = \mathrm{lfp}H\}$ and $\{\mathbf{y} = G\langle \mathbf{y}, z\rangle,\ z = \mathrm{lfp}H\}$ respectively. We can solve these two systems of equations with 3 memory cells separately.

Our modular fixpoint computation can save memory but it has redundancy in solving link equations. Fixpoint iterations for link equations are nested. So the inner fixpoint computation does redundant computations comparing flat fixpoint iterations. At each iteration step for $\mathrm{lfp}H$, we have to compute values for $\mathbf{y}$ from $\bot$. But, in flat fixpoint iterations, values of unknown variables never decrease. If this overhead of the nested iterations is too big, then our memory saving method is not appropriate for practical purposes.

In our experiment, memory saving was big enough not only to compensate the overhead but also to save running time. We made a new program analyzer Kairac by applying our

method to Airac. Given a program P, Airac solves a system of equations for all program points in P. For the program P, Kairac solves the same system of equations using entry and exit points of procedures as link variables. In our experiment comparing Airac and Kairac, memory peak of Kairac was 29-50% of Airac's memory peak for GNU programs with 5,000-20,000LOC. For the same programs, in spite of the nested iterations, Kairac could save 32-34% of running time due to reduced size of memory.

From here, every domain is a cpo and every function is continuous, if we do not state otherwise. $\bot$ is used to represent the least elements for any domains without subscripts.

1.3 Our Contributions

- We made a memory-efficient method for fixpoint computation. Given a system of equations, we derive link equations using Bekić's theorem. Solutions of link equations can be used to solve the original problem in parts. Therefore we can solve each small problem with less memory.
- We found that redundancy of nested fixpoint iterations is not serious. We implemented a modular program analyzer Kairac. In our experiment, Kairac achieved memory saving and time saving together. Since the redundancy of the nested iterations is evident, we don't say that our method can save time for all cases. But we are sure that the overhead is not too big to make our memory-efficient fixpoint method infeasible.

2 Modular Fixpoint Computation

Our method for memory-efficient fixpoint computation is to solve problems in parts with the most accurate global information. Suppose that we divide a system of equations into two parts. Then there are variables which are defined in one part and used in the other part. Through those variables the two parts are linked. So if we have the solutions of the link variables, we can solve each part separately. For the solutions of link variables, we derive link equations that consist of just link variables using Bekić's theorem.

Using Bekić's theorem, we can partition a system of equations in an arbitrary manner. Suppose that we have a system E of equation $E = \{\mathbf{x} = f\langle \mathbf{x}, \mathbf{y}\rangle,\ \mathbf{y} = g\langle \mathbf{x}, \mathbf{y}\rangle\}$. We can derive two new equations $\mathbf{x} = f\langle \mathbf{x}, \mathrm{lfp}\lambda\mathbf{y}.g\langle \mathbf{x}, \mathbf{y}\rangle\rangle$ and $\mathbf{y} = g\langle \mathrm{lfp}\lambda\mathbf{x}.f\langle \mathbf{x}, \mathbf{y}\rangle, \mathbf{y}\rangle$ from the system of equation. And we can solve these equations separately. But we can't save memory, because f and g need $\mathbf{x}$ and $\mathbf{y}$ simultaneously as E does.

To achieve memory saving, we must be able to solve each module with less memory than what is required to solve the original problem. For the system E of equations, our modularization makes a new system E' of equations of equations:

$$E' = \{\mathbf{x} = f'\langle \mathbf{x}, \mathbf{z}\rangle,\ \mathbf{y} = g'\langle \mathbf{y}, \mathbf{z}\rangle,\ \mathbf{z} = h\langle \mathbf{x}, \mathbf{y}\rangle\}$$

where $\mathbf{z}$ is a vector for link variables. We can solve E' with less memory, if E' satisfies the followings:

C1(separation) $h\langle \mathbf{x}, \mathbf{y}\rangle = h_1\mathbf{x} \sqcup h_2\mathbf{y}$

From $\mathbf{z} = h\langle \mathbf{x}, \mathbf{y}\rangle$ we derive a link equation $\mathbf{z} = h\langle \mathrm{lfp}\lambda\mathbf{x}.f'\langle \mathbf{x}, \mathbf{z}\rangle, \mathrm{lfp}\lambda\mathbf{y}.g'\langle \mathbf{y}, \mathbf{z}\rangle\rangle$. The least solution of the link equation is same with the least $\mathbf{z}$ that satisfies an inequation $\mathbf{z} \sqsupseteq h\langle \mathrm{lfp}\lambda\mathbf{x}.f'\langle \mathbf{x}, \mathbf{z}\rangle, \mathrm{lfp}\lambda\mathbf{y}.g'\langle \mathbf{y}, \mathbf{z}\rangle\rangle$. Therefore if $h\langle \mathbf{x}, \mathbf{y}\rangle = h_1\mathbf{x} \sqcup h_2\mathbf{y}$, then, instead of solving $\mathbf{z} = h\langle \mathbf{x}, \mathbf{y}\rangle$, we can find the least $\mathbf{z}$ that satisfies an inequation system $\{\mathbf{z} \sqsupseteq h_1\,\mathrm{lfp}\lambda\mathbf{x}.f'\langle \mathbf{x}, \mathbf{z}\rangle,\ \mathbf{z} \sqsupseteq h_2\,\mathrm{lfp}\lambda\mathbf{y}.g'\langle \mathbf{y}, \mathbf{z}\rangle\}$. Since we can evaluate h_1 and h_2 separately during fixpoint iterations, same memory cells can be shared between $\mathbf{x}$ and $\mathbf{y}$.

C2(sparseness) $|\mathbf{z}| + \max(|\mathbf{x}|, |\mathbf{y}|) < |\mathbf{x}| + |\mathbf{y}|$

Computations for $\mathbf{z} \sqsupseteq h_1 \,\mathrm{lfp}\lambda\mathbf{x}.f'\langle\mathbf{x},\mathbf{z}\rangle$ and $\mathbf{z} \sqsupseteq h_2 \,\mathrm{lfp}\lambda\mathbf{y}.g'\langle\mathbf{y},\mathbf{z}\rangle$ need $|\mathbf{z}| + |\mathbf{x}|$ and $|\mathbf{z}| + |\mathbf{y}|$ memory cells respectively. With the solution of the link equation, we can solve $\mathbf{x} = f'\langle\mathbf{x},\mathbf{z}\rangle$ and $\mathbf{y} = g'\langle\mathbf{y},\mathbf{z}\rangle$ with $|\mathbf{z}| + |\mathbf{x}|$ and $|\mathbf{z}| + |\mathbf{y}|$ memory cells respectively. Since memory cells for $\mathbf{x}$ and $\mathbf{y}$ can be shared, it requires $|\mathbf{z}| + \max(|\mathbf{x}|, |\mathbf{y}|)$ memory cells to find the least solution of E'. And since fixpoint iterations for E use $|\mathbf{x}| + |\mathbf{y}|$ memory cells, to save memory, above condition must be satisfied.

In the following sections, we show that we can derive a link equation that satisfies C1 from any systems of equations and, in case of program analyses, C2 is apt to be satisfied.

2.1 Modularizing Systems of Equations

Our modularization has two steps - dividing systems of equations and setting up link equations. Suppose that we have a system E of equations:

$$E = \{x_i = e_i\}_{i \in U} \tag{1}$$

We divide E into two disjoint parts $\{x_i = e_i\}_{i \in I}$ and $\{x_i = e_i\}_{i \in J}$ where $U = I \cup J$ and $I \cap J = \emptyset$. From the two parts, we find variables that are defined in one part but used in the other part. Such variables are link variables. We can define a set of link variables K as follows:

$$K = (\{i \mid x_i \in \bigcup_{i \in I} \mathrm{FV}(e_i)\} - I) \cup (\{i \mid x_i \in \bigcup_{i \in J} \mathrm{FV}(e_i)\} - J)$$

where $\mathrm{FV}(e)$ is a set of free variables in e. With K we can define a new system E' of equations by introducing new variables for the link variables:

$$\begin{array}{lcl} E' & = & E_1 \cup E_2 \cup E_3 \\ E_1 & = & \{x_i = e_i[x'_j/x_j \mid j \in K]\}_{i \in I} \\ E_2 & = & \{x_i = e_i[x'_j/x_j \mid j \in K]\}_{i \in J} \\ E_3 & = & \{x'_i = x_i\}_{i \in K} \end{array}$$

where $e[x'/x]$ is e in which all occurrences of x are replaced with x'. It is easy to show that E' is actually same with E by eliminations. E_3 is a system of link equations of E since we can solve E_1 and E_2 separately with the solution of E_3. Let $\mathbf{x} = \langle x_i\rangle_{i \in I}$, $\mathbf{y} = \langle x_i\rangle_{i \in J}$ and $\mathbf{z} = \langle x'_i\rangle_{i \in K}$. Then we can rewrite E' as follows:

$$E' = \{\mathbf{x} = f\langle\mathbf{x},\mathbf{z}\rangle,\ \mathbf{y} = g\langle\mathbf{y},\mathbf{z}\rangle,\ \mathbf{z} = h\langle\mathbf{x},\mathbf{y}\rangle\} \tag{2}$$

With $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, we can trivially define f, g and h. For further explanation, we show the definition of h from E_3:

$$h\langle\mathbf{x},\mathbf{y}\rangle = \sigma_K(\mathbf{x}) \sqcup \sigma_K(\mathbf{y})$$

$\sigma_K(\mathbf{v})$ selects components of $\mathbf{v}$ corresponding link variables. Using Bekić's theorem we can get rid of the dependency between $\mathbf{z}$ and $\mathbf{x}$, $\mathbf{y}$:

$$\mathbf{z} = \sigma_K(\mathrm{lfp}\lambda\mathbf{x}.f\langle\mathbf{x},\mathbf{z}\rangle) \sqcup \sigma_K(\mathrm{lfp}\lambda\mathbf{y}.g\langle\mathbf{y},\mathbf{z}\rangle) \tag{3}$$

As we intended the equation for link variables is defined by join of two fixpoint computations for modules.

2.2 Nested Fixpoint Iteration

We solve equation (3) with nested fixpoint iterations. The solution of equation (3) is given as $\mathrm{lfp}R$:

$$R = \lambda\mathbf{z}.h_1\mathbf{z} \sqcup h_2\mathbf{z} \tag{4}$$

where $h_1 = \lambda\mathbf{z}.\sigma_K(\mathrm{lfp}\lambda\mathbf{x}.f\langle\mathbf{x},\mathbf{z}\rangle)$ and $h_2 = \lambda\mathbf{z}.\sigma_K(\mathrm{lfp}\lambda\mathbf{y}.g\langle\mathbf{y},\mathbf{z}\rangle)$. Inner iterations for $\mathrm{lfp}\lambda\mathbf{x}.f\langle\mathbf{x},\mathbf{z}\rangle$ and $\mathrm{lfp}\lambda\mathbf{y}.g\langle\mathbf{y},\mathbf{z}\rangle$ are *local iterations* and outer iteration for $\mathrm{lfp}R$ is *global iterations*.

Since $\mathrm{lfp}R$ and the least $\mathbf{z}$ satisfying a system of inequations $\{\mathbf{z} \sqsupseteq h_1\mathbf{z},\ \mathbf{z} \sqsupseteq h_2\mathbf{z}\}$ are same, we define a chain $\{Z^k\}_{k\geq 0}$ for $\mathrm{lfp}R$:

$$\begin{array}{lcll} Z^0 & = & \bot & \\ Z^k & = & h_{s(k)}Z^{k-1} \sqcup Z^{k-1} & (k>0) \end{array}$$

where $\forall k > 0.s(k) \in \{1,2\}$ and $\forall k > 0.\exists k' > k.\{s(k'') \mid k < k'' \leq k'\} = \{1,2\}$. Since only one of h_1 and h_2 is evaluated for each Z^k, we can recycle memory between evaluations of h_1 and h_2. Memory cells for $\mathbf{z}$ should be maintained during global iterations. Therefore memory required for solving (3) is $|\mathbf{z}| + \max(|\mathbf{x}|,|\mathbf{y}|)$.

Having $\mathrm{lfp}R = \bigsqcup_{k\geq 0} Z^k$ as a solution of equation (3), we can get solutions for $\mathbf{x}$ and $\mathbf{y}$ by computing the least fixpoints of $R_\mathbf{x} = \lambda\mathbf{x}.f\langle\mathbf{x},\mathrm{lfp}R\rangle$ and $R_\mathbf{y} = \lambda\mathbf{y}.g\langle\mathbf{y},\mathrm{lfp}R\rangle$ respectively. $\mathrm{lfp}R_\mathbf{x}$ costs memory of which size is $|\mathbf{z}| + |\mathbf{x}|$ and $\mathrm{lfp}R_\mathbf{y}$ costs memory of which size is $|\mathbf{z}| + |\mathbf{y}|$. Therefore if we compute $\mathrm{lfp}R_\mathbf{x}$ and $\mathrm{lfp}R_\mathbf{y}$ separately, required memory space is $|\mathbf{z}| + \max(|\mathbf{x}|,|\mathbf{y}|)$ too.

If the number of the link variables is small enough, then we can save memory required. If the condition (5) is satisfied, then we can save memory by solving E' instead of E.

$$|\mathbf{z}| + \max(|\mathbf{x}|,|\mathbf{y}|) < |\langle x_i\rangle_{i\in U}| \tag{5}$$

Systems of equations in flow-sensitive program analyses are apt to satisfy (5), if we divide the systems of equations by procedure. Suppose that we have a program which consists of two procedures `f` and `g`. The system E' of equations in (2) can correspond to the program as the following manner: f is derived from the procedure `f` and $\mathbf{x}$ represents the program points of `f`; in the same manner g and $\mathbf{y}$ correspond to the procedure `g` and the program points of `g` respectively; $\mathbf{z}$ represents entry and exit points of procedures. Since the number of procedures is much smaller than the number of program points, (5) is satisfied by the program analyses for most cases.

For practical memory-efficiency redundancy of nested fixpoint iteration should not be serious. During global iterations, values for $\mathbf{x}$ and $\mathbf{y}$ are computed from $\bot$ by local iterator. This redundancy is inevitable cost since we don't remember values for $\mathbf{x}$ and $\mathbf{y}$. If we can acquire satisfactory memory efficiency at affordable cost of such redundant computations, then our modular approach is worth trying for fixpoint computation.

2.3 Correctness

Correctness of our modular fixpoint computation is proved as a corollary of Bekić's theorem[1, 5]. Let's see Bekić's theorem first.

Theorem 1 (Bekić's Elimination of Simultaneous Recursion)**.** *Suppose that there are two systems equations S and S':*

$$S\left\{\begin{array}{lcl} x & = & f\langle x,y\rangle \\ y & = & g\langle x,y\rangle \end{array}\right. \qquad S'\left\{\begin{array}{lcl} x & = & L(y) \\ y & = & R(y) \end{array}\right.$$

where $L = \lambda y.\,\mathrm{lfp}(\lambda x.f\langle x,y\rangle)$ and $R = \lambda y.g\langle L(y),y\rangle$. Then S and S' have the same least solution.

Corollary 1 (Correctness of Modular Fixpoint Computation)**.** *Suppose that there are two systems equations S and S′′:*

$$S\begin{cases} x &= f\langle x, y\rangle \\ y &= g\langle x, y\rangle \end{cases} \qquad S''\begin{cases} x &= f\langle x, \mathrm{lfp}R\rangle \\ y &= R(y) \end{cases}$$

where $L = \lambda y.\,\mathrm{lfp}(\lambda x.f\langle x, y\rangle)$ *and* $R = \lambda y.g\langle L(y), y\rangle$*. Then S and S′′ have the same solution.*

proof. By solving each equation of S'' separately, we can have that $\langle \mathrm{lfp}\lambda x.f\langle x, \mathrm{lfp}R\rangle, \mathrm{lfp}R\rangle$ is the least solution of S''. And, by Theorem 1, we also have that $\langle \mathrm{lfp}\lambda x.f\langle x, \mathrm{lfp}R\rangle, \mathrm{lfp}R\rangle = \langle L(\mathrm{lfp}R), \mathrm{lfp}R\rangle$ is the least solution of S. □

By Corollary 1, we can say our modular fixpoint computation can solve systems of equations without loss of accuracy.

2.4 Upper Approximation of Infinite Chains

To cope with domains which have infinite chains, we use widening[8] and narrowing[8]. Widening is used to guarantee termination of fixpoint iterations. Since our fixpoint computation consists of nested fixpoint iterations, we have to be careful to use those operators:

- We have to make the transfer functions for the global iterators extensive. Since widening and narrowing are not monotone, these operators in the local iterators can make the global iterators fail to terminate. Therefore R in (4) should be modified to be extensive:

  $$R_{\mathrm{ext}} = \lambda \mathbf{z}.\mathbf{z} \sqcup (\sigma_K(\mathrm{lfp}^{\triangledown}_{\triangle}\lambda \mathbf{x}.f\langle \mathbf{x}, \mathbf{z}\rangle) \sqcup \sigma_K(\mathrm{lfp}^{\triangledown}_{\triangle}\lambda \mathbf{y}.g\langle \mathbf{y}, \mathbf{z}\rangle))$$

  where $\mathrm{lfp}^{\triangledown}_{\triangle}$ is a fixpoint operator with widening($\triangledown$) and narrowing($\triangle$).

- We don't apply narrowing to global iterator because extensive functions can't make decreasing chains:

  $$\mathrm{lfp}^{\triangledown} R_{\mathrm{ext}}$$

When we compute a upper bound for a chain defined with widening and narrowing, computation order can change the result of the upper approximation. It's because widening and narrowing are not monotone. If we apply widening more lately, then we can get more accurate results. In our experiment we used chaotic iteration with worklist and we got different analysis results from Airac and Kairac. In Section 4 there are tables that show such results of the two analyzers.

3 Design of Modular Program Analysis

In this section we present how our modular fixpoint computation works with static program analyses. We describe a typical flow-sensitive program analysis $\mathcal{A}$ which approximates possible states at program points. And then we will show another analysis $\mathcal{A}_K$ which is made by integrating modular fixpoint computation and $\mathcal{A}$. We implemented two analyzers Airac and Kairac using $\mathcal{A}$ and $\mathcal{A}_K$ respectively. For fixpoint iterations, we used chaotic iteration[4] with worklist. Our analyses are based on abstract interpretation[6, 7].

For a given program $\mathcal{A}$ makes a table which maps each program point to an abstract memory. An abstract memory at a program point approximates possible run-time states at the program point. The table is computed as a solution of a system of semantic equations derived from the program. Unknown variables of the system of equations correspond to

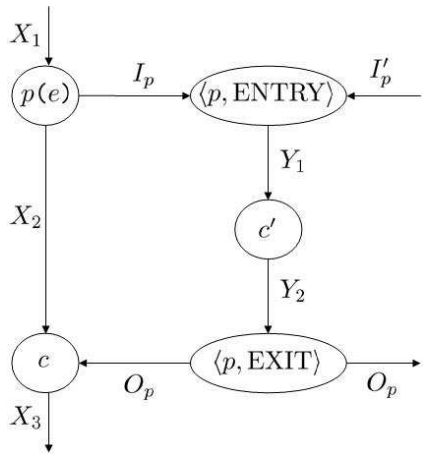

Figure 1: A snippet of program P in flow graph representation

the program points of the program. For the snippet of program P in Figure 1 $\mathcal{A}$ solves the following semantic equations:

$$E_P = \left\{ \begin{array}{rcl} X_2 & = & X_1 \\ X_3 & = & C[\![c]\!](X_2 \sqcup O_p) \\ I_p & = & X_1\{x \mapsto \mathcal{V}[\![e]\!]X_1\} \\ \vdots & & \end{array} \right\} \cup \left\{ \begin{array}{rcl} Y_1 & = & I_p \sqcup I'_p \\ Y_2 & = & C[\![c']\!]Y_1 \\ O_p & = & Y_2 \end{array} \right\} \cup \left\{ \begin{array}{rcl} I'_p & = & \cdots \\ \vdots & & \end{array} \right\} \cup \cdots$$

where x is the formal parameter of procedure p. X_i and Y_i are abstract memories at bodies of procedures. I_p and I'_p are abstract memories for input states of procedure p at call sites. O_p is an abstract memories for output states of procedure p. $C[\![c]\!]$ returns an abstract memory created by executing the command c with an input abstract memory. $\mathcal{V}[\![e]\!]$ evaluates the expression e with an input abstract memory.

$\mathcal{A}_K$ divides a system of equations of by procedure and uses entry and exit points of procedures as link variables. I_p, I'_p and O_p are link variables of E_p. Since abstract memories at call sites are always joined at entry nodes, we don't have to keep individual abstract memories at call sites. Instead of assigning different auxiliary variables to I_p and I'_p, we use just one auxiliary variable Z_{I_p} for $I_p \sqcup I'_p$. So E_p is modularized to be E'_p by $\mathcal{A}_K$:

$$\begin{array}{rcl} E'_P & = & \left\{ \begin{array}{rcl} X_2 & = & X_1 \\ X_3 & = & C[\![c]\!](X_2 \sqcup Z_{O_p}) \\ I_p & = & X_1\{x \mapsto \mathcal{V}[\![e]\!]X_1\} \\ \vdots & & \end{array} \right\} \cup \left\{ \begin{array}{rcl} Y_1 & = & Z_{I_p} \\ Y_2 & = & C[\![c']\!]Y_1 \\ O_p & = & Y_2 \end{array} \right\} \cup \left\{ \begin{array}{rcl} I'_p & = & \cdots \\ \vdots & & \end{array} \right\} \cup \cdots \\ & \cup & \left\{ \begin{array}{rcl} Z_{I_p} & = & I_p \sqcup I'_p \\ Z_{O_p} & = & O_p \\ \vdots & & \end{array} \right\} \end{array}$$

Since the number of procedures is far less than the number of program points in most programs, $\mathcal{A}_K$ can save memory. Suppose that a program has n procedures $p_1, \ldots, p_n$. Then the number of memory cells that $\mathcal{A}_K$ uses is $2 * n + \max_{i=1}^{n} |p_i|$. $|p_i|$ is the number of program points in p_i. It is hard for most programs not to satisfy $2 * n + \max_{i=1}^{n} |p_i| < \Sigma_{i=1}^{n} |p_i|$.

$$
\begin{array}{rcl}
Pgm \ni pgm & \rightarrow & pdec^{+}\ main\texttt{()} \\
pdec & \rightarrow & p\texttt{(}x\texttt{)\{}c^{*}\texttt{\}} \\
c & \rightarrow & i \mid f \\
Instr \ni i & \rightarrow & x\texttt{:=}e \mid p\texttt{(}e\texttt{)} \mid \texttt{assert(}e\texttt{)} \mid \texttt{skip} \\
f & \rightarrow & \texttt{if}\ e\ c^{*}\ c^{*} \mid \texttt{while}\ e\ c^{*} \\
Expr \ni e & \rightarrow & n \mid x \mid e\texttt{+}e \mid e\texttt{=}e \mid e\texttt{<}e \mid \texttt{!}e
\end{array}
$$

$$
\begin{array}{rcl}
n \in Node & = & Label \times Instr + Pid \times \{\text{ENTRY}, \text{EXIT}\} \\
p, q \in Pid & & \text{procedure identifiers} \\
l \in Label & & \text{labels for each node} \\
\mathsf{succof} & : & Node \rightarrow 2^{Node} \\
\mathsf{predof} & : & Node \rightarrow 2^{Node} \\
\mathsf{callers} & : & Pid \rightarrow 2^{Pid} \\
\mathsf{callsitesin} & : & Pid \rightarrow 2^{Node} \\
\mathsf{callsitesof} & : & Pid \rightarrow 2^{Node}
\end{array}
$$

Figure 2: Language: Syntax and Graph

3.1 Language and Semantics

3.1.1 Language

Let's see what kind of language we deal with. Our language is simple but has a procedure call. We use graph representations for input programs. Procedures are transformed into flow graphs with special entry and exit nodes. Condition parts of branches and loops are dissolved into assert instructions. For example, `if` $e\ c_1\ c_2$ is transformed into two instruction lists `assert(`e`)` $:: c_1$ and `assert(!`e`)` $:: c_2$. Edges in graphs are compiled into functions: succof returns successor nodes for the input node; predof returns predecessor nodes for the input node; callers returns procedure identifiers of procedures which call the input procedure; callsitesin returns all nodes of given procedures whose instructions are procedure calls; callsitesof returns all call sites of the input procedure. Inter-procedural edges are drawn like Figure 1.

The following domains and functions are used in the rest of this paper:

$$
\begin{array}{rcl}
Mem & = & Var \rightarrow Val \\
x \in Var & & \text{set of variables} \\
Val & & \text{a cpo for value} \\
\mathcal{V}[\![\cdot]\!] & : & Expr \rightarrow (Mem \rightarrow Val) \\
\mathsf{istrue} & : & Val \rightarrow \{\mathsf{true}, \mathsf{false}\}
\end{array}
$$

Mem is a abstract domain for memory. Variables are used as abstract addresses. *Val* is a abstract domain for values of expressions and it is a cpo. $\mathcal{V}[\![\cdot]\!]$ returns a value for the input expression and memory. $\mathcal{V}[\![\cdot]\!]$ is strict, i.e., $\mathcal{V}[\![\_]\!]\bot = \bot$. istrue returns true or false according to the input abstract value.

3.1.2 Semantics for $T \in$ *Node* $\rightarrow$ *Mem*

We present semantics for a typical flow-sensitive but context-insensitive program analysis $\mathcal{A}$. Given a program P, $\mathcal{A}$ computes a partial table $T \in Node \rightarrow Mem$ for nodes of P as a fixpoint of a semantic transfer function. T maps each node to memory which approximates possible

$$
\begin{array}{lll}
\tau: & ((Node \to Mem) \to (Node \to Mem)) & \\
& \to 2^{Node} \times (Node \to Mem) \to 2^{Node} \times (Node \to Mem) & \\
= & \lambda\mathcal{F}.\lambda\langle W, T\rangle.\ m : Mem,\ n : Node,\ W' : 2^{Node} & \\
& \quad \text{case } W \text{ of} & \\
& \qquad \emptyset & \Rightarrow \quad \langle W, T\rangle \\
& \quad \mid \{n\} \cup W' & \Rightarrow \quad m := \mathcal{F}\ T\ n \\
& & \qquad \text{if } m \not\sqsubseteq T[n] \\
& & \qquad\quad W' := W' \cup \mathsf{succof}(n) \\
& & \qquad\quad T := T\{n \mapsto m\} \\
& & \qquad \tau\ F\ \langle W', T\rangle
\end{array}
$$

Figure 3: Fixpoint iterator τ for $T \in Node \to Mem$

states created by running an instruction of the node. Program semantics for $\mathcal{A}$ is defined with the following semantic function $\mathcal{F}$:

$$
\begin{array}{lll}
\mathcal{F}: & (Node \to Mem) \to (Node \to Mem) & \\
= & \lambda T.\lambda n.\ m, m' : Mem & \\
& \quad m := \bigsqcup_{n' \in \mathsf{predof}(n)} T[n'] & \\
& \quad \text{case } n \text{ of} & \\
& \quad \mid \langle \_, x\texttt{:=}e\rangle & \Rightarrow \quad m\{x \mapsto \mathcal{V}[\![e]\!]m\} \\
& \quad \mid \langle \_, \texttt{assert(}e\texttt{)}\rangle & \Rightarrow \quad \text{if } \mathsf{istrue}(\mathcal{V}[\![e]\!]m) \text{ then } m \text{ else } \bot \\
& \quad \mid \langle \_, \texttt{skip}\rangle & \Rightarrow \quad m \\
& \quad \mid \langle \_, q\texttt{(}e\texttt{)}\rangle & \Rightarrow \quad m \\
& \quad \mid \langle p, \text{ENTRY}\rangle & \Rightarrow \quad m' := \bot \\
& & \qquad \text{foreach } \langle l, p\texttt{(}e\texttt{)}\rangle \in \mathsf{callsitesof}(p) \\
& & \qquad\quad m := \bigsqcup_{n' \in \mathsf{predof}\langle l, p\texttt{(}e\texttt{)}\rangle} T[n'] \\
& & \qquad\quad m' := m' \sqcup m\{x \mapsto \mathcal{V}[\![e]\!]m\} \text{ where } p\texttt{(}x\texttt{)\{}\ldots\texttt{\}} \\
& & \qquad m' \\
& \quad \mid \langle \_, \text{EXIT}\rangle & \Rightarrow \quad m
\end{array}
$$

Assignment updates the input memory using the variable of left-hand side as the abstract address. Assertion passes the input memory unchanged or bottom memory depending on the input expression's value. Procedure call just joins all input memories from local predecessor nodes and inter-procedural predecessor nodes. Procedure entry joins all memories created by actual parameter bindings at all call sites since $\mathcal{A}$ is context-insensitive. Skip and procedure exit join all input memories.

$\mathcal{A}$ constructs a partial table T for a program P with the semantic function $\mathcal{F}$ and the worklist-based fixpoint iterator τ in Figure 3:

$$
\begin{array}{rcl}
\mathcal{P}[\![]\!] & : & Pgm \to (Node \to Mem) \\
\mathcal{P}[\![P]\!] & = & \tau\ \mathcal{F}\ \langle\{\langle main, \text{ENTRY}\rangle\}, \bot\rangle
\end{array}
$$

τ selects a node n from the worklist W and executes $\mathcal{F}$ with node n and partial table T. $\mathcal{F}\ T\ n$ returns new memory m and τ compares m with the old memory of n. If m differs from $T[n]$, then nodes influenced by n are added to the worklist and the table T is updated with the new memory m. τ runs while the worklist is not empty.

Memory peak of $\mathcal{A}$ is proportional to the number of nodes in programs. It's because τ have to keep a table T with entries for all program nodes in system memory while τ is running.

3.2 Modular Program Analysis

We apply modular fixpoint computation to $\mathcal{A}$ to make another program analysis $\mathcal{A}_K$ which uses less memory than $\mathcal{A}$. $\mathcal{A}_K$ is a modular program analysis that analyzes each procedure separately and accumulates global information from those local analyses.

Regarding entry and exit points of procedures as link variables, $\mathcal{A}_K$ computes a partial table $U \in Pid \to Mem \times Mem$ for a given program P. U is an abstract procedure environment[13] which maps each procedure of P to its input/output memory pair. We need new semantic function $\mathcal{G}$ for local iterations. Given a procedure environment U, semantic function $\mathcal{G}$ returns a transfer function for a procedure body:

$$
\begin{array}{rl}
\mathcal{G}: & (Pid \to Mem \times Mem) \to \\
& (Node \to Mem) \to (Node \to Mem) \\
= & \lambda U.\lambda T.\lambda n.\ m : Mem \\
& \quad m := \bigsqcup_{n' \in \mathsf{predof}(n)} T[n'] \\
& \quad \text{case } n \text{ of} \\
& \qquad \langle \_, q(e) \rangle \quad \Rightarrow \quad m \sqcup U[q].2 \\
& \quad | \ \langle \_, x := e \rangle \quad \Rightarrow \quad m\{x \mapsto \mathcal{V}[\![e]\!]m\} \\
& \quad | \ \langle \_, \texttt{assert}(e) \rangle \quad \Rightarrow \quad \text{if } \mathsf{istrue}(\mathcal{V}[\![e]\!]m) \text{ then } m \text{ else } \bot \\
& \quad | \ \langle \_, \texttt{skip} \rangle \quad \Rightarrow \quad m \\
& \quad | \ \langle q, \text{ENTRY} \rangle \quad \Rightarrow \quad U[q].1 \\
& \quad | \ \langle \_, \text{EXIT} \rangle \quad \Rightarrow \quad m
\end{array}
$$

Unlike $\mathcal{F}$ semantics of procedure call and procedure entry are defined to use the procedure environment U.

We construct a procedure environment U for a given program P with fixpoint iterator τ' in Figure 4. τ' and τ corresponds to global iterator and local iterator of modular fixpoint computation respectively. τ' uses two auxiliary functions forward and backward. When a local iteration for a procedure p is finished, we may need to re-evaluate some procedures which are called by p or call p. Procedures whose input memories move during local iteration for p are added into the worklist and U is updated with the new input memories by forward. If output memory of p move after local iteration for p, then backward puts callers of p into the worklist and updates U with new output memory of p. τ called by τ' is the same what is defined in Figure 3. But succof returns node connected by intra-procedural edges when it is used in the context of $\mathcal{A}_K$. Inter-procedural propagations are done by global iterator using callsitesin and callers. Procedure environment U for program P is computed by $\mathcal{A}_K$ as follows:

$$
\begin{array}{rcl}
\mathcal{P}'[\![\,]\!] & \in & Pgm \to (Pid \to Mem \times Mem) \\
\mathcal{P}'[\![P]\!] & = & \tau'\ \mathcal{G}\ \langle \{\text{main}\}, \bot \rangle
\end{array}
$$

Global iterator τ' uses memory whose size is proportional to the sum of procedures number and nodes number of the biggest procedure. Global iterator τ' keeps memories at entry and exit nodes of procedures. Memory space occupied by local iterator τ is bounded by the node number of the biggest procedure of a program and is recycled over local iterations for procedures. After we have a procedure environment U of a program at the end of global iteration, we can compute memories at each node of procedure bodies to verify desired properties. For example, given a procedure environment U and a procedure p, we can construct a partial table $T \in Node \to Mem$ for the procedure p by running $\tau\ (\mathcal{G}\ U)\ \langle \{\langle p, \text{ENTRY} \rangle\}, \bot \rangle$. Memory space for this computations is also proportional to sum of procedure p's size and procedure environment U's size.

3.3 Static Garbage Collection

In addition to modular fixpoint computation, we used *static garbage collection*. Static garbage collection prunes abstract memories according to variables' scopes at exit and entry nodes.

$$
\begin{array}{rl}
\tau' : & ((Pid \to Mem \times Mem) \to (Node \to Mem) \to (Node \to Mem)) \\
& \to 2^{Pid} \times (Pid \to Mem \times Mem) \\
& \to 2^{Pid} \times (Pid \to Mem \times Mem) \\
= & \lambda\mathcal{G}.\lambda\langle W, U\rangle.\ p : Pid,\ W' : 2^{Pid},\ T : Node \to Mem \\
& \ \text{case } W \text{ of} \\
& \quad \emptyset \quad \Rightarrow \quad \langle \emptyset, U\rangle \\
& \quad \mid \ \{p\} \cup W' \quad \Rightarrow \\
& \qquad T := \tau\ (\mathcal{G}\ U)\ \{\langle p, \text{ENTRY}\rangle\}\ \bot \\
& \qquad \langle W', U'\rangle := \mathsf{backward}\ p\ T\ (\mathsf{forward}\ p\ T\ \langle W', U\rangle) \\
& \qquad \tau'\ \mathcal{G}\ \langle W', U'\rangle
\end{array}
$$

Figure 4: Fixpoint iterator τ' for $U \in Pid \to Mem \times Mem$

$$
\begin{array}{rl}
\mathsf{forward} : & Pid \to (Node \to Mem) \to (2^{Pid} \times (Pid \to Mem \times Mem)) \\
& \to (2^{Pid} \times (Pid \to Mem \times Mem)) \\
= & \lambda p.\lambda T.\lambda\langle W, U\rangle.\ m : Mem,\ n : Node \\
& \ \text{foreach } \langle l, q(e)\rangle \in \mathsf{callsitesin}(p) \\
& \quad m := \bigsqcup_{n \in \mathsf{predof}\langle l, q(e)\rangle} T[n] \\
& \quad m := m\{x \mapsto \mathcal{V}[\![e]\!]m\} \text{ where } q(x)\{\ldots\} \\
& \quad \text{if } m \not\sqsubseteq U[q].1 \\
& \qquad U := U \sqcup \{q \mapsto \langle m, \bot\rangle\} \\
& \qquad W := W \cup \{q\} \\
& \ \langle W, U\rangle \\
\\
\mathsf{backward} : & Pid \to (Node \to Mem) \to (2^{Pid} \times (Pid \to Mem \times Mem) \\
& \to (2^{Pid} \times (Pid \to Mem \times Mem) \\
= & \lambda p.\lambda T.\lambda\langle W, U\rangle. \\
& \ \text{if } T\langle p, \text{EXIT}\rangle \not\sqsubseteq U[p].2 \\
& \quad \langle W \cup \mathsf{callers}(p), U\{p \mapsto \langle U[p].1, T[\langle p, \text{EXIT}\rangle]\rangle\}\rangle \\
& \ \text{else} \\
& \quad \langle W, U\rangle
\end{array}
$$

Figure 5: Auxiliary functions for τ'

Scopes of variables are determined syntactically by front-end of the analyzers using declarations and address operator(&). Since our abstract semantics has no stack, all variables are virtually global. So variables which are originally declared as local variables in C can propagate over scopes of procedures. These unnecessary propagations make analyzers keep larger abstract memories in memory. With static garbage collection, we can reduce size of abstract memories at program points.

Table 1: Statistics for programs 5,000-20,000LOC. V_U is the number of unknown variables(program points) and V_L is the number of link variables(exits and entries of procedures). sgc means the static garbage collection. oom means out-of-memory.

Program	LOC	$\#V_U$	$\#V_L$	Item	$\mathcal{A}$	$\mathcal{A}_K$	$\mathcal{A}$+sgc	$\mathcal{A}_K$+sgc
bison 1.875	16,127	10,579	978	space(MB)	oom	1,843	oom	1,618
(w/o lib)				time(sec)		19,758		57,950
				#alarm		127		134
tar 1.13	17,220	11,099	446	space(MB)	2,109	1,126	1,823	720
				time(sec)	58,475	40,898	67,785	52,989
				#alarm	374	372	228	371
sed 4.0.8	6,053	9,388	492	space(MB)	1,618	754	1,034	336
				time(sec)	39,093	25,902	31,068	16,535
				#alarm	99	101	96	59
grep 2.5.1	9,297	7,248	316	space(MB)	631	312	560	203
				time(sec)	9,412	3,831	8,288	5,045
				#alarm	368	366	368	368
gzip 1.2.4	7,323	3,684	196	space(MB)	419	200	324	139
				time(sec)	5,222	3,446	2,999	2,577
				#alarm	273	266	276	270
			ratio to $\mathcal{A}$	space	100%	50%	78%	29%
			(+sgc)				(100%)	(37%)
				time	100%	66%	98%	68%
			(+sgc)				(100%)	(69%)

4 Experiment

To compare $\mathcal{A}$ and $\mathcal{A}_K$, we implemented Airac and Kairac for $\mathcal{A}$ and $\mathcal{A}_K$ respectively. Actually Airac was created by author's previous work [14]. Kairac was made by applying modular fixpoint computation to Airac. Airac is a buffer overrun analyzer for C programs and it has 2 phases internally: 1) Airac approximates a program's runtime behavior by computing a table which maps program points of the program to abstract memories; 2) Airac verifies every buffer access expressions in the program with the table. Kairac approximates program's runtime behavior by computing a table which maps each procedure to a pair of input/output memories. To verify buffer access expressions in a procedure, Kairac runs the local iterator to make a table that maps program points of the procedure's body to abstract memories.

By $\mathcal{A}_K$, we were able to save space and time together. In the experiment, we analyzed various GNU programs with $\mathcal{A}$ and $\mathcal{A}_K$. Spaces in Table 1, 2 were peaks of memory occupied by the analyzers. Times in Table 1, 2 were running time of the analyzers. Times and memories were measured to cover all required amounts of space and time from fixpoint computations and verifications. Experiment was done on a Linux system with Pentium4 3GHz CPU and 4GB RAM. Table 1 shows that, for programs bigger than 5,000LOC, average memory peaks of $\mathcal{A}_K$ are 29% and 50% of $\mathcal{A}$'s peaks. For programs smaller than 5,000LOC, average memory peaks of $\mathcal{A}_K$ were 39% and 87% of $\mathcal{A}$'s peaks. Table 1 shows that average running time of $\mathcal{A}_K$ is 66% and 68% of $\mathcal{A}$'s running times for 5,000LOC-20,000LOC programs. For programs smaller than 5,000LOC, $\mathcal{A}_K$'s average running times are 14% and 60% of $\mathcal{A}$'s running times. Contrary to Corollary 1, alarm numbers in Table 1 and Table 2 are different. In the following sections we discuss about what portions of memory are saved, how the speed of analysis can increase and why alarm numbers are different.

Table 2: Statistics for programs <5,000LOC. V_U is the number of unknown variables(program points) and V_L is the number of link variables(exits and entries of procedures). sgc means the static garbage collection.

Program	LOC	$\#V_U$	$\#V_L$	Item	$\mathcal{A}$	$\mathcal{A}_K$	$\mathcal{A}$+sgc	$\mathcal{A}_K$+sgc
tar.c of	1,194	830	54	space(MB)	31	32	10	23
tar 1.13				time(sec)	23	42	2	20
				#alarm	0	0	0	0
grep.c of	1,742	1,426	26	space(MB)	30	31	32	19
grep 2.5.1				time(sec)	35	53	29	31
				#alarm	3	3	3	3
gzip.c of	1,744	1,460	58	space(MB)	32	34	34	24
gzip 1.2.4				time(sec)	41	51	50	20
				#alarm	8	9	8	9
sed 4.0.8	4,322	3,182	158	space(MB)	211	169	138	55
(w/o lib)				time(sec)	1,228	660	483	118
				#alarm	2	7	2	2
		ratio to $\mathcal{A}$		space	100%	87%	70%	39%
		(+sgc)					(100%)	(55%)
				time	100%	60%	42%	14%
		(+sgc)					(100%)	(33%)

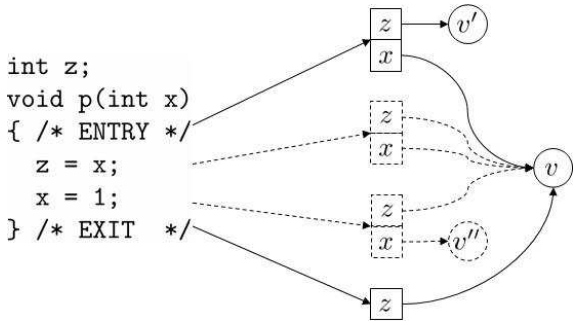

Figure 6: Memory usage of $\mathcal{A}$ and $\mathcal{A}_K$ analyzing the procedure p

4.1 Saved Space

$\mathcal{A}_K$ recycles memory cells which are used by the local fixpoint iterator. Figure 6 shows data items manipulated by $\mathcal{A}$ and $\mathcal{A}_K$. For $\mathcal{A}_K$, dashed objects are shared by the local iterator for procedures. But $\mathcal{A}$ keeps all objects in memory throughout fixpoint iterations. Recycling is done by automatic garbage collection that is provided by nML[1] in which Airac and Kairac is written.

Memory saving of $\mathcal{A}_K$ does not concur with the ratio of procedures number to program points number. Less than 10% of program points are entry and exit points in Table 1, 2. Let's consider an example where ratio of procedures number to program points number is 1:10. Suppose that we have a program with 5 procedures and each procedure has 20 program points. Then $\mathcal{A}$ keeps a table with 100 entries for fixpoint iterations. But $\mathcal{A}_K$ keeps a table

[1] A Korean dialect of ML. `http://ropas.snu.ac.kr/n`

with 10 entries for global iterations and a table with 20 entries for local iterations. In this example, we can expect maximum memory saving of $\mathcal{A}_K$ to be 70%. Since entries and exits are less 10% of program points in our experiment, we may expect over 70% memory-saving. But since $\mathcal{A}$ already has mechanisms for memory-efficiency, there is not that much memory can be saved by $\mathcal{A}_K$. As you can see in Figure 6, some objects such as v are shared by references. Therefore ratio of $\mathcal{A}_K$'s memory peak to $\mathcal{A}$'s memory peak(or $\mathcal{A}_K$+sgc's memory peak to $\mathcal{A}$+sgc's memory peak) is not same with the ratio of entries and exits number to program points number.

Table 1, 2 shows that static garbage collection promotes $\mathcal{A}_K$'s memory saving. With static garbage collection, $\mathcal{A}_K$ saved 21% and 48% memory space more for 5,000-20,000LOC and under 5,000LOC respectively. It's because static garbage collection make more memory cells recyclable for $\mathcal{A}_K$. Since `x`'s scope is the body of the procedure `p`, the abstract memory at exit doesn't have to keep an entry for `x`. Since the abstract memory at exit doesn't have an entry for `x`, memory space for v'' is recycled by $\mathcal{A}_K$. Though $\mathcal{A}$ has static garbage collection too, the value object for v'' can't be recycled, since $\mathcal{A}$ keeps all abstract memories in memory.

4.2 Increased Speed

$\mathcal{A}_K$ accomplished not only memory saving but also time saving. Local iteration of modular fixpoint computation has redundancy, because local iterator always starts from $\perp$. But, in experiment, $\mathcal{A}_K$'s running time was below 70% of $\mathcal{A}$'s running time.

We think that since $\mathcal{A}_K$ deals with smaller objects than $\mathcal{A}$, operators execute faster in $\mathcal{A}_K$ than in $\mathcal{A}$. Though, by sharing, logically big objects can occupy small space in physical memory, operations for the objects require time proportional to the logical size. Furthermore OS can execute programs that use less memory faster. As we exemplified in Section 4.1, $\mathcal{A}_K$ keeps 30% of unknown variables which are kept by $\mathcal{A}$ if entries and exits are 10% of total program points.

Apart from small object size, $\mathcal{A}_K$ has an iteration strategy named *wait-at-backwarded-procedure* to reduce redundant computation. Suppose that procedure p calls procedure q and procedure r calls procedure p. The entry state of procedure q and the exit state of procedure p can move after local fixpoint iterations for procedure p's body. Then $\mathcal{A}_K$ has to do local fixpoint iterations for procedure r and q. In this case we choose procedure q before procedure r because change of procedure q's exit state can cause re-computations for bodies of procedure p and procedure r as a chain reaction.

4.3 Incompatible Accuracy

If we use widening and narrowing with different chaotic iteration strategies, then we can get different results for the same program. It's because widening and narrowing are not monotone. We will show you two cases. One is analyzed better by $\mathcal{A}_K$ and another is analyzed better by $\mathcal{A}$. Since $\mathcal{A}$ and $\mathcal{A}_K$ do the same interval analysis, we explain the two cases with interval analysis. Interval analysis approximates numeric expressions of programs to have integer intervals defined with minimum and maximum values. Widening and narrowing operations which we used for interval domain is defined in [8, 14].

- CASE 1: more accurately analyzed by $\mathcal{A}_K$

```
void foo(int x){
  printint(x);  /* [1,+inf] v.s. [1,2] */
}
void main(){
  foo(1);
  foo(2);
}
```

$\mathcal{A}_K$ can make more accurate approximation for the formal parameter `x` of the procedure `foo` in above program. Due to context-insensitivity of $\mathcal{A}$'s analysis, sequential procedure calls of the same procedure make a loop at the entry of the procedure `foo`. Since $\mathcal{A}$'s iterator traverses flow graph with depth-first strategy at call sites, $\mathcal{A}$ updates entry states of procedures whenever they are called. Therefore, at the second call site in the procedure `main`, $\mathcal{A}$ applies widening at the entry of the procedure `foo`. As a result of widening, the formal parameter `x` of the procedure `foo` has $[1, +\infty]$. But since $\mathcal{A}_K$ applies widening at entry/exit memories after local iterations, the formal parameter `x` of the procedure `foo` already have $[1, 2]$ when widening is applied by the global iterator. So, finally, the formal parameter `x` is approximated to have $[1, 2]$ by $\mathcal{A}_K$.

- CASE 2: more accurately analyzed by $\mathcal{A}$

```
void loop(int x){
  if(x<10) loop(x+1);
  printint(x); /* [10,10] v.s. [10,+inf] */
}
void main(){
  loop(1);
}
```

$\mathcal{A}_K$ can't analyze above program as accurate as $\mathcal{A}$. Since the procedure `loop` of above program is a recursively defined, loops are created at entry and exit of the procedure `loop`. Due to widening, formal parameter `x` of `loop` is approximated to have $[1, +\infty]$. Since $\mathcal{A}_K$ doesn't use narrowing for global iterations, the formal parameter `x` of the procedure `loop` is finally approximated to have value $[1, +\infty]$ by $\mathcal{A}_K$. But $\mathcal{A}$ applies narrowing to the entry of loop and recovers the value of the variable `x` to be $[1, 10]$.

5 Related Work

For set-based analysis there is a nice modular analysis which was presented by Flanagan and Felleisen in [12]. The process of their analysis are similar to ours. The analysis in [12] simplifies a constraint system for each module with respect to the module's external set variables. External set variables of a module's constraint system are all set variables except those definitely have no interaction with outside the module. By combining simplified constraint systems of modules the analysis constitute a constraint system for the entire program. With the closure of the constraint system for the entire program the analysis analyzes each module separately. Comparing to our approach the modular approach of [12] has narrower application areas.

It's because our modular fixpoint computation is applicable to any equation-based program analysis.

Duesterwald *et al.* [9] present algorithms that minimize the size of the system of equations. The algorithms in [9] divide systems of equations into congruence classes. If two equations have the same fixpoint, then they are included in the same congruence class. By selecting one equation from each congruence class the algorithms set up a new equation system. Granularity of modularization in [9] is determined by the inherent congruence relation of the system of equations. But our approach can divide the system of equations arbitrarily. Since [9] has no experiment result, we don't have any hard evidence of the algorithms' performance.

If we are interested in a subset of unknown variables, then we can use the top-down fixpoint algorithm in [2]. The top-down fixpoint algorithm evaluates equations that are needed by the unknown variables which we want to know. So we can touch a subset of original set of equations. But if we need to know all solutions of unknown variables, then the top-down fixpoint algorithm can't save memory.

$\mathcal{A}_K$ can be regarded as one of modular analyses in [3]. Cousot and Cousot introduced various approaches for modular program analyses in [3]. One approach is to simplifying modules with pre-analyses. We can regard fixpoint computations for link equations as pre-global analyses that simplifies modules to be independent from each other.

Speed-up of $\mathcal{A}_K$ was not expected but happened. This speed-up comes from reduced size of objects in memory. There have been some studies [11, 10, 14] that attempt to decrease time by reduction of strength. Methods in [11, 10, 14] require specially designed operators or domains since operations are done on differences of values. But, in our case, we just decrease problem size without re-designs of operators and domains. Our fixpoint method is to makes small problems from given problems with local fixpoint iterations. So our modular fixpoint computation can work with other techniques without interferences.

6 Conclusion

6.1 Summary

Our modular fixpoint computation achieved feasible memory-efficiency in spite of redundancy in nested fixpoint iterations. We integrated modular fixpoint computation and $\mathcal{A}$ to make $\mathcal{A}_K$. In our experiment, $\mathcal{A}_K$ with static garbage collection reduced memory peak to 29% for 5,000-20,000LOC GNU programs. Memory saving of our method made redundancy of our method be no problem. Even though we can't say that our method can save time for all cases, we are sure that our method doesn't have serious overhead.

Our modular fixpoint computation can be used with other techniques for efficient fixpoint computation in orthogonal ways. Our approach can be regarded as just making small problems from a given problem. So we can use any techniques to solve each small problem more efficiently.

6.2 Future Work

Even though modular fixpoint computation can save much memory space and time required to solve systems of equations, for scalability, there are some further work to be done: 1) It is always possible for any analyzer to exhaust whole system memory when it analyzes really huge programs. In such cases, secondary storages can play key roles to survive the limit of the memory; 2) Modular fixpoint iteration can evolve into a distributed fixpoint algorithm. It is natural to make global fixpoint iterator a coordinator of a distributed fixpoint system. The distributed fixpoint system can have many homogeneous participants for local fixpoint iterations.

References

[1] Hans Bekić. Definable operation in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition - Hans Bekic (1936-1982)*, pages 30–55, London, UK, 1984. Springer-Verlag.

[2] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Providence, RI 02912, 1992.

[3] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR 2001*, Compact disk, L'Aquila, Italy, 6–12 August, 2001 2001. Scuola Superiore G. Reiss Romoli.

[4] Patrick Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Technical Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble, 1977.

[5] Patrick Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, Feb.–May 2005.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[7] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[8] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer-Verlag, 1992.

[9] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *CC '94: Proceedings of the 5th International Conference on Compiler Construction*, pages 357–373, London, UK, 1994. Springer-Verlag.

[10] Hyunjun Eo and Kwangkeun Yi. An improved differential fixpoint iteration method for program analysis. In *Proc. of the 3rd Asian Workshop on Programming Language and Systems*, Shanghai, Nov 2002.

[11] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, 1998.

[12] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 235–248, New York, NY, USA, 1997. ACM Press.

[13] Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 296–306, New York, NY, USA, 1986. ACM Press.

[14] Youngbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *SAS 2005: 12th Annual International Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2005.