Heap Dependence Analysis for Sequential Programs

Barnali Basak

Supervisors

Prof. Amey Karkare

&

Prof. Sanjeev K. Aggarwal

Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

June 8, 2011

Outline

Introduction

Dependence Analysis : produces execution order constraints between program statements.

Introduction

- Data Dependences : Occurs due to data flow of program.
  - Flow Dependence (Read after Write):

    $\boxed{\mathrm{x}} = a + b;$

    $y = \boxed{\mathrm{x}} + z;$

Write access Read access

Introduction

- Data Dependences : Occurs due to data flow of program.

  - Anti Dependence (Write after Read):
    $a = \boxed{\text{x}} + b;$
    $\boxed{\text{x}} = y + z;$

Write access Read access

Introduction

- Data Dependences : Occurs due to data flow of program.

  - Output Dependence (Write after Write):
    $\boxed{x} = a + b;$
    $\boxed{x} = y + z;$

Write access Read access

Introduction

$$for(i = 1; i <= n; i++)\{$$
$$S1 : x = a[i] + b;$$
$$S2 : a[i] = y + z;$$
$$S3 : a[i + 1] = x; \}$$

- Loop Dependences :

Iteration 1	**Iteration 2**
$S1$: x = a[1] + b;	$S1$: x = a[2] + b;
$S2$: a[1] = y + z;	$S2$: a[2] = y + z;
$S3$: a[2] = x;	$S3$: a[3] = x;

  - Loop independent : $< S1, S2 >$

Introduction

$$for(i = 1; i <= n; i++)\{$$
$$S1 : x = a[i] + b;$$
$$S2 : a[i] = y + z;$$
$$S3 : a[i+1] = x;\}$$

- Loop Dependences :

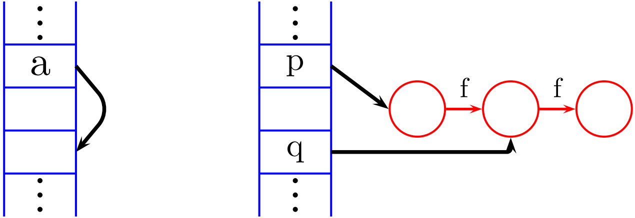
Iteration 1	**Iteration 2**
$S1:$ x = a[1] + b;	$S1:$ x = a[2] + b;
$S2:$ a[1] = y + z;	$S2:$ a[2] = y + z;
$S3:$ a[2] = x;	$S3:$ a[3] = x;

  - Loop carried : $< S3, S1 >$ and $< S3, S2 >$

Introduction

Dependences arise due to :

- scalars and pointers to stack allocated objects (stack-directed pointers)
- pointers to heap heap allocated objects (heap-directed pointers)

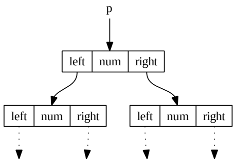


Introduction

Difficulties in static analysis of heap :

- Structure of heap is unknown.
- Size of heap structure is potentially unbounded.
- Lifetime of heap object is not limited by the scope that creates it.
- Presence of pointer induced aliasing, ex. `p→f→f` and `q→f` access same heap object.

Motivation

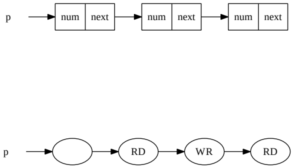


(a) Data structure

```
          void treeAdd(tree t) {
            if(t == NULL)
              return;
S1.         tl = t→left;
S2.         treeAdd(tl);
S3.         tr = t→right;
S4.         treeAddd(tr);
            t→num = tl→num + tr→num;
          }
```

(b Function traversing the data structure.

Motivation



(a) Nodes read and written by code

```
        ...
S1.     p = list;
        while (p→next != NULL) {
S2.       q = p→next;
S3.       temp = q→num;
S4.       r = q→next;
S5.       r→num = temp;
S6.       p = r;
        }
        ...
```

(b) Loop traversing the data structure.

Objective

- Intraprocedural analysis
  - works on each procedure separately, does not cross process boundary
  - finds out both fine grained and coarse grained parallelism

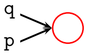
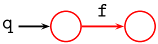
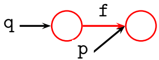
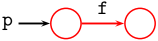
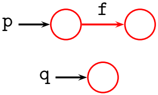
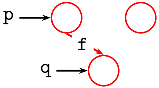
Objective

- Intraprocedural analysis
  - works on each procedure separately, does not cross process boundary
  - finds out both fine grained and coarse grained parallelism
- Loop sensitive analysis
  - targets loop sensitive processes
  - efficiently extracts loop level parallelism

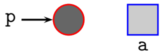
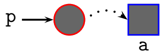
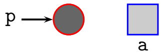
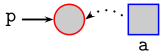
Objective

- Intraprocedural analysis
  - works on each procedure separately, does not cross process boundary
  - finds out both fine grained and coarse grained parallelism
- Loop sensitive analysis
  - targets loop sensitive processes
  - efficiently extracts loop level parallelism
- Interprocedural analysis
  - works on whole program, crosses process boundary
  - handles function calls more efficiently

Programming Model

<table>
<tr><th></th><th>Before execution</th><th>Statement</th><th>After execution</th></tr>
<tr><td>Allocations</td><td></td><td><code>p = malloc()</code></td><td></td></tr>
<tr><td rowspan="3">Pointer Assignments</td><td></td><td><code>p = q</code></td><td></td></tr>
<tr><td></td><td><code>p = q→f</code></td><td></td></tr>
<tr><td></td><td><code>p = NULL</code></td><td></td></tr>
<tr><td rowspan="2">Structure updates</td><td></td><td><code>p→f = NULL</code></td><td></td></tr>
<tr><td></td><td><code>p→f = q</code></td><td></td></tr>
</table>

Programming Model

	Before execution	Statement	After execution
Heap reading/writing		`a = p→data`	
		`p→data = a`	

Statements are normalized

$\mathtt{x} = \mathtt{p} \rightarrow \mathtt{f} \rightarrow \mathtt{data}$

$$\begin{aligned} \mathtt{q} &= \mathtt{p} \rightarrow \mathtt{f} \\ \mathtt{x} &= \mathtt{q} \rightarrow \mathtt{data} \end{aligned}$$

Outline

Workflow

Steps of intraprocedural analysis :

- **Preprocessing - Initialization and Annotation**
- **Computation of Read and Write sets of abstract access paths**
- **Detection of dependences**

Preprocessing

- Initialize global pointer variables and pointer parameters to symbolic locations.

Preprocessing

- Annotate statements with tagging directive, consisting of four attributes
  - Used pointer set
  - Defined pointer set
  - Access field
  - Access type

Preprocessing

Stmt	Used ptr	Def ptr	Acc field	Acc type
`p = q`	{q}	{p}	`Null`	alias
`p = q→next`	{q}	{p}	`next`	link trav
`··· = p→data`	{p}	{}	`Null`	read heap
`p→data = ···`	{p}	{}	`Null`	write heap
`fun(p, q)`	{p, q}	{}	`Null`	func call

Computation of Read and Write Sets

- **Access Path :** symbolic heap location $\mathtt{l}_0$ or location followed by pointer fields like $\mathtt{l}_0 \rightarrow \mathtt{f}_1 \rightarrow \cdots \rightarrow \mathtt{f}_k$

$< \mathtt{l}_0 >$ $\quad$ $< \mathtt{l}_0 \rightarrow \mathtt{f}_1 >$ $\quad$ $< \mathtt{l}_0 \rightarrow \mathtt{f}_1 \cdots \mathtt{f}_k >$

Computation of Read and Write Sets

- **Access Path :** symbolic heap location $\mathtt{l_0}$ or location followed by pointer fields like $\mathtt{l_0} \rightarrow \mathtt{f_1} \rightarrow \cdots \rightarrow \mathtt{f_k}$

$< \mathtt{l_0} >$ $\quad$ $< \mathtt{l_0} \rightarrow \mathtt{f_1} >$ $\quad$ $< \mathtt{l_0} \rightarrow \mathtt{f_1} \cdots \mathtt{f_k} >$

- **Abstraction Scheme :**
  - Length of access path is limited to length $\mathtt{k}$.
  - Summary field '*' abstracts fields dereferenced beyond length $\mathtt{k}$. (Here $k = 1$)

$< \mathtt{l_0} >$ $\quad$ $< \mathtt{l_0} \rightarrow \mathtt{f_1} >$ $\quad$ $< \mathtt{l_0} \rightarrow \mathtt{f_1} \rightarrow * >$

Computation of Read and Write Sets

State Analysis :

Definition

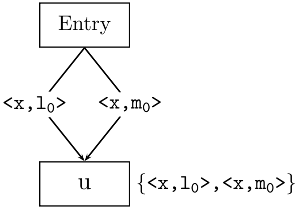
The state of heap directed pointer variable `x` *at a program point* `u` *is the set symbolic memory locations such that some paths from the* `Entry` *point to* `u` *result in the access of symbolic locations by the variable* `x`*.*

Computation of Read and Write Sets

State Analysis :

Definition

The state of heap directed pointer variable `x` *at a program point* `u` *is the set symbolic memory locations such that some paths from the* `Entry` *point to* `u` *result in the access of symbolic locations by the variable* `x`*.*



Computation of Read and Write Sets

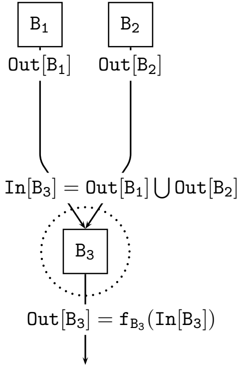
State Analysis :

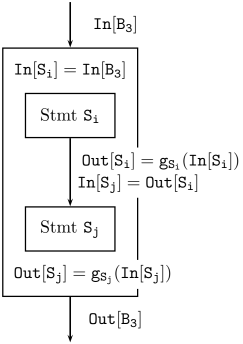
- control flow sensitive forward-flow analysis

```
fun stateAnalysis(CFG[f]:(N, E, Entry, Exit)) {
  Out[Entry] = InitSet;      /* Boundary condition  */
  for each basic block B other than Entry
    Out[B] = ϕ;                      /* Initialization  */
  while (changes to any Out[ ] occur)    /* Iterate  */
  {
    for each basic block B other than Entry
    {
      In[B] = ⋃(Out[P]), for all predecessors P of B;
      Out[B] = f_B(In[B]);
    }
  }
}
```

Top level algorithm of state analysis

Computation of Read and Write Sets





$f_{B_3}(In[B_3]) = g_{S_k}(\cdots(g_{S_1}(In[B_3])))$

$g_{S_i}(In[S_i]) = (In[S_i] - Kill[S_i]) \bigcup Gen[S_i]$

Computation of Read and Write Sets

Gen and kill sets of statements :

Statement	Gen set	Kill set
`p = q`	`{<p,m> | <q,m>∈In[S]}`	`{<p,l> | <p,l>∈In[S]}`
`p = q→next`	`{<p,m→next> | <q,m>∈In[S]}`	`{<p,l> | <q,l>∈In[S]}`
`··· = p→data`	{}	{}
`p→data = ···`	{}	{}
`fun(p,q)`	{}	{}

Computation of Read and Write Sets

Gen and kill sets of statements :

Statement	Gen set	Kill set
`p = q`	{<p,m> | <q,m>∈In[S]}	{<p,l> | <p,l>∈In[S]}
`p = q→next`	{<p,m→next> | <q,m>∈In[S]}	{<p,l> | <q,l>∈In[S]}
`··· = p→data`	{}	{}
`p→data = ···`	{}	{}
`fun(p,q)`	{}	{}

Read/Write set of treeAdd function :

```
     void treeAdd(tree t) {
       if(t == NULL)
         return;
S1.    tl = t→left;
S2.    treeAdd(tl);
S3.    tr = t→right;
S4.    treeAddd(tr);
       t→num = tl→num + tr→num;
     }
```

Initialization : $\{<\mathtt{t}, \mathtt{l}_0>\}$	
Stmt	State
S1	$\{<\mathtt{t}, \mathtt{l}_0>, <\mathtt{tl}, \mathtt{l}_0 \rightarrow \mathtt{left}>\}$
S3	$\{<\mathtt{t}, \mathtt{l}_0>, <\mathtt{tl}, \mathtt{l}_0 \rightarrow \mathtt{right}>\}$

Shows state

Stmt	Read set	Write set
S2	<tl,l_0 →left>, <tl,l_0 →left→*>}	<tl,l_0 →left>, <tl,l_0 →left→*>}
S4	<tr,l_0 →right>, <tr,l_0 →right→*>}	<tr,l_0 →right>, <tr,l_0 →right→*>}

Shows read/write sets

Dependence Detection

- Two access paths `p.`α' and `q.`β' interfere if

$$\text{isInterfering}(p, \alpha, q, \beta) = True$$

α and β are prefixes of α' and β'.

Dependence Detection

- Two access paths `p.`α' and `q.`β' interfere if

$$\text{isInterfering}(p, \alpha, q, \beta) = True$$

  α and β are prefixes of α' and β'.
- Statements `S` and `T` are dependent on each other if

$$\mathsf{interfere}(\text{set}_1,\ \text{set}_2) \equiv \text{isInterfering}(p, \alpha, q, \beta)$$

  p.$\alpha \in set_1$, q.$\beta \in set_2$

Dependence Detection

- Two access paths `p.`α' and `q.`β' interfere if

$$\text{isInterfering}(p, \alpha, q, \beta) = True$$

  α and β are prefixes of α' and β'.
- Statements `S` and `T` are dependent on each other if

$$\mathsf{interfere}(\text{set}_1,\ \text{set}_2) \equiv \text{isInterfering}(p, \alpha, q, \beta)$$

  p.$\alpha \in set_1$, q.$\beta \in set_2$

$$\text{flow-dep}(S, T) \equiv \mathsf{interfere}(\text{write}(S),\ \text{read}(T))$$

$$\text{anti-dep}(S, T) \equiv \mathsf{interfere}(\text{read}(S),\ \text{write}(T))$$

$$\text{output-dep}(S, T) \equiv \mathsf{interfere}(\text{write}(S),\ \text{write}(T))$$

Dependence Detection

Detecting dependence between function calls :

```
        void treeAdd(tree t) {
          if(t == NULL)
            return;
S1.       tl = t→left;
S2.       treeAdd(tl);
S3.       tr = t→right;
S4.       treeAddd(tr);
          t→num = tl→num + tr→num;
        }
```

Dependence Detection

Query	Predicate	Dependence
flow-dep(S2,S4) anti-dep(S2,S4) output-dep(S2,S4)	isInterfering(t,left,t,right)	No

Detects dependence

Dependence Detection

Detecting dependence of loop :

```
       ...
S1.    p = list;
       while (p→next != NULL) {
S2.       q = p→next;
S3.       temp = q→num;
S4.       r = q→next;
S5.       r→num = temp;
S6.       p = r;
       }
       ...
```

Stmt	Read set	Write set
S3	<q, l_0 →next>, <q, l_0 →next→*>}	{}
S5	{}	<r, l_0 →next→*>,

Shows read/write sets

Dependence Detection		
Query	Predicate	Dependence
flow-dep(S3,S5)	isInterfering(p,next,p,next)	No
anti-dep(S3,S5)		Yes
output-dep(S3,S5)		No

Detects dependence

Shortcomings

- Does not perform well in presence of loops
- Considers worst case summary of called functions
- Imprecise for both loop level and function level parallelism

Outline

Workflow

Steps of loop sensitive analysis :

- **Identification of navigator variable and navigator expression**
- **Computation of Read and Write sets of complete access paths**
- **Detection of dependences (both loop independent and loop carried)**

Identifying Navigator

Navigator of loop consists of :

- navigator variable : pointer variable used to traverse loop
- navigator expression : sequence of pointer field references to navigate loop

...
p = list;
while (p→next != NULL) {
S11. ...= p→num;
S12. p→next→num = ...;
S13. p = p→next→next;
}
...
navigator variable : p
navigator expression : next→next
	...
	p = list;
	while (p→next != NULL) {
	S21. ...= p→num;
	S22. p→next→num = ...;
	S23. p = p→next;
	}
	...
	navigator variable : p
	navigator expression : next

Detecting Dependence

Loop Independent Dependence :

Observation 1: If shape is Tree

- `p`$\rightarrow$`f`$\rightarrow$`f` and `p`$\rightarrow$`f`$\rightarrow$`g` do not access common node
- `p`$\rightarrow$`f`$\rightarrow$`f` and `p`$\rightarrow$`f`$\rightarrow$`f` always access common node

Detecting Dependence

Loop Independent Dependence :

Observation 2: If shape is DAG

- p→f and p→f→g do not access common node as former path is proper subpath of later path
- p→f→f and p→f→g can potentially access common node

Detecting Dependence

...
p = list;
while (p→next != NULL) {
S11. ...= p→num;
S12. p→next→num = ...;
S13. p = p→next→next;
}
...
navigator variable : p
navigator expression : next→next
shape attribute : Tree
	...
	p = list;
	while (p→next != NULL) {
	S21. ...= p→num;
	S22. p→next→num = ...;
	S23. p = p→next;
	}
	...
	navigator variable : p
	navigator expression : next
	shape attribute : Tree

Read Set: {p}, Write Set: {}
Read Set: {}, Write Set: {p→next}

No loop independent dependence

Detecting Dependence

Loop Carried Dependence :

- access paths are generalized for arbitrary iterations and equations are formed
- equations are tested for any integer solution using GCD or Lamport test

Detecting Dependence

Loop Carried Dependence :

- access paths are generalized for arbitrary iterations and equations are formed
- equations are tested for any integer solution using GCD or Lamport test

...
p = list;
while (p→next != NULL) {
S11. ...= p→num;
S12. p→next→num = ...;
S13. p = p→next→next;
}
...
	...
	p = list;
	while (p→next != NULL) {
	S21. ...= p→num;
	S22. p→next→num = ...;
	S23. p = p→next;
	}
	...

navigator expression : next→next Read Set: $p \rightarrow next^{2i}$, Write Set: {} Read Set: {}, Write Set: $p \rightarrow next^{2j+1}$ **$2*i = 2*j + 1$** (No Dependence)	navigator expression : next Read Set: $p \rightarrow next^{i}$, Write Set: {} Read Set: {}, Write Set: $p \rightarrow next^{j+1}$ **$i = j + 1$** (Dependence)

Outline

Workflow

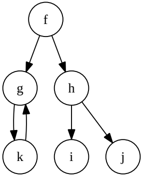
Steps of interprocedural analysis :

- **Topologically order call graph**
- **Compute abstract summary for each procedure node of call graph**
- **Use abstract summary for inter-procedural analysis**

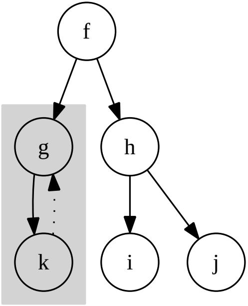
Call Graph

- Each node represents a procedure
- Each directed edge `(e,e')` indicates that `e` calls `e'`

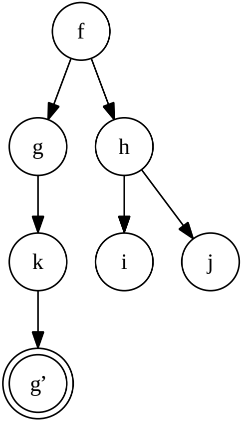
```
S1.    procedure f()
S2.    begin
S3.        call g();
S4.        call h();
S5.    end
S6.    procedure g()
S7.    begin
S8.        call k();
S9.    end
S10.   procedure k()
S11.   begin
S12.       call g();
S13.   end
S14.   procedure h()
S15.   begin
S16.       call i();
S17.       call j();
S18.   end
```



Processing Call Graph



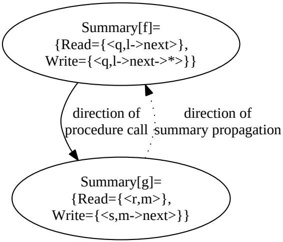
(a) Cyclic call graph



(b) Corresponding DAG

```
S1.   procedure f(p)
S2.   begin
S3.     q = p→next;
S4.     g(q);
S5.   end
S6.   procedure g(r)
S7.   begin
S8.     ··· = r→num;
S9.     s = r→next;
S10.    s→num=···;
S11.  end
```

(a) Example program with procedure call



(b) Call graph showing summary of procedures

Outline

Related Work

- Work done by Ghiya et. al[1]
  - uses coarse shape attribute of data structure
  - computes complete access paths in terms of anchor pointer
  - tests for aliases of access paths using shape information
  - imprecise and conservative

[1]Rakesh Ghiya, Laurie Hendren and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 159-173.

Related Work

- Work done by Navarro et. al[2]
  - obtains Reference Shape Graph of heap structure
  - tags nodes with read and write access
  - identifies dependences based on tagging information
  - expensive in time

[2] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata and E. Zapata. A New Dependence Test Based on Shape Analysis for Pointer-Based Codes. In *the 17th International Workshop on Languages and Compilers for Parallel Computing*, September 2004.

Outline

Future Work

- Improving of summarization technique

Future Work

- Improving of summarization technique
- Developing shape analysis technique to handle complex and cyclic structures

Future Work

- Improving of summarization technique
- Developing shape analysis technique to handle complex and cyclic structures
- Extending loop sensitive analysis to handle irregular control flow constructs

Future Work

- Improving of summarization technique
- Developing shape analysis technique to handle complex and cyclic structures
- Extending loop sensitive analysis to handle irregular control flow constructs
- Further developing prototype model to handle large benchmarks

Future Work

- Improving of summarization technique
- Developing shape analysis technique to handle complex and cyclic structures
- Extending loop sensitive analysis to handle irregular control flow constructs
- Further developing prototype model to handle large benchmarks
- Extending prototype model to implement interprocedural analysis

Thank You !!!