Scalable Array SSA and Array Dataflow Analysis

Silvius Rus
Guobin He
Lawrence Rauchwerger

SSA Program Representation: Scalars

Parasol

Original code

```
v = 100
If (x>0) Then
  v = 100
EndIf
Print v
```

```
v = 100
If (x>0) Then
  v = 50
EndIf
If (x>0) Then
  Print v
EndIf
```

Original code

SSA Form

```
v1 = 100
If (x>0) Then
  v2 = 100
EndIf
v3 = φ(v2, v1)
Print v3
```

```
v1 = 100
If (x>0) Then
  v2 = 50
EndIf
v3 = γ(x>0, v2, v1)
If (x>0) Then
  Print v3
EndIf
```

Gated SSA Form

Constant Propagation using SSA

Parasol

Before CP

```
v1 = 100
If (x>0) Then
  v2 = 100
EndIf
v3 = ϕ(v2, v1)
Print v3
```

```
v1 = 100
If (x>0) Then
  v2 = 50
EndIf
v3 = γ(x>0, v2, v1)
If (x>0) Then
  Print v3
EndIf
```

Before CP

SSA Form

Gated SSA Form

```
v1 = 100
If (x>0) Then
  v2 = 100
EndIf
v3 = ϕ(100, 100)
Print 100
```

```
v1 = 100
If (x>0) Then
  v2 = 50
EndIf
v3 = γ(x>0, 50, 100)
If (x>0) Then
  Print 50
EndIf
```

After CP

After CP

Array SSA: Motivation

Simple Solution

Treat arrays as scalars

```
A1(1) = 100
A2(2) = 200
Print A2(1)
```

Too conservative!

Must consider subscripts!

Call Sites

Loops

Control

```
Subroutine set (A, k, v)
  A(k) = v
End

...
Call   set(A, 1,   0)
If (x>0) Then
  Call set(A, 2,   1)
EndIf
Do j = 3, 10
  Call set(A, j,   3)
  Call set(A, j+8, 4)
EndDo
Print A(1) + A(5)
If (x>0) Then
  Print A(2)
EndIf
```

Previous Work

- Analytical array subregion-based dataflow frameworks
  - Scalable and expressive
  - No standard form = harder to use than SSA; sometimes biased towards a particular analysis technique
  - *Triolet CC '86*, *Callahan SC '87*, *Gross SPE '90*, *Burke TOPLAS '90*, *Feautrier IJPP '91*, *Maydan SPPL '93*, *Tu LCPC '93*, *Pugh TR '94*, *Gu SC '95*, *Hall SC '95*, *Creusillet LCPC '96*, *Haghighat TOPLAS '96*, *Hoeflinger '98*, *Moon ICS '98*, *Wonnacott LCPC '00*

- Element-wise data flow information as Array SSA by enumeration
  - More accurate than treating arrays as scalars, easy to use
  - Complexity proportional to the dimension of the array = not scalable
  - At compile-time, only applicable to constant subscript expressions
  - *Knobe SPPL '98*, *Sarkar SAS '98*

Array SSA Desiderata

Analytical and explicit
data flow information
at array element level

Array Data Flow: Partial Kills


```
x1 = 100

x2 = 200

Print x2
```

```
A1(1) = 100
DEF(A1) = {1}
A2(2) = 200
KILL(A2) = {2}
Print A2(1)
USE(A2)  = {1}
```

Disjoint

Scalars:

The use of x_2 may be replaced with the value defined by x_2 because it kills all its reaching definitions (x_1)

Array SSA:

The use of A_2 may not be replaced with the value defined by A_2 because it does not kill $A_1(1)$

But how do we get from A_2 to A_1?

Use-Def Chains: δ Nodes


```
A(1) = …
A(2) = …
A(1) = …

Print A(1)
Print A(2)
Print A(10)
```

$A_1(1)$ = …
$[A_2, \{1\}] = \delta(A_0, [A_1,\{1\}])$

$A_3(2)$ = …
$[A_4, [1:2]] = \delta(A_0, [A_2,\{1\}], [A_3,\{2\}])$

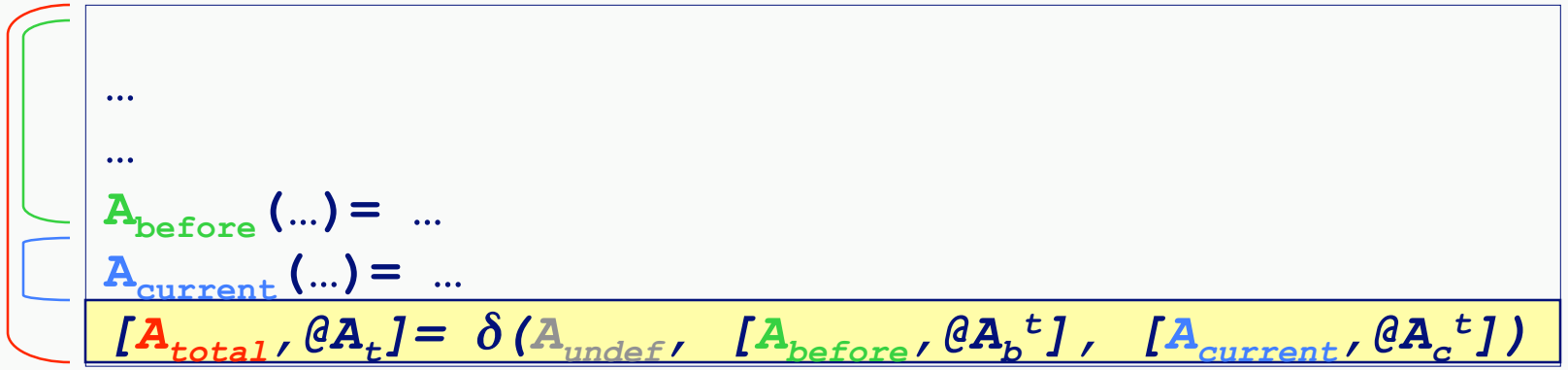
$A_5(1)$ = …
$[A_6, [1:2]] = \delta(A_0, [A_4,\{2\}], [A_5,\{1\}])$

Print $A_6(1)$ → A_5
Print $A_6(2)$ → A_5 → A_4 → A_3
Print $A_6(10)$ → A_0

Array SSA Use-Def Chains

Just like scalar SSA + compare access regions

δ Nodes: Formal Definition



$$@A_b^t \cap @A_c^t = \varnothing$$

$$@A_t = @A_b^t \; @A_c^t$$

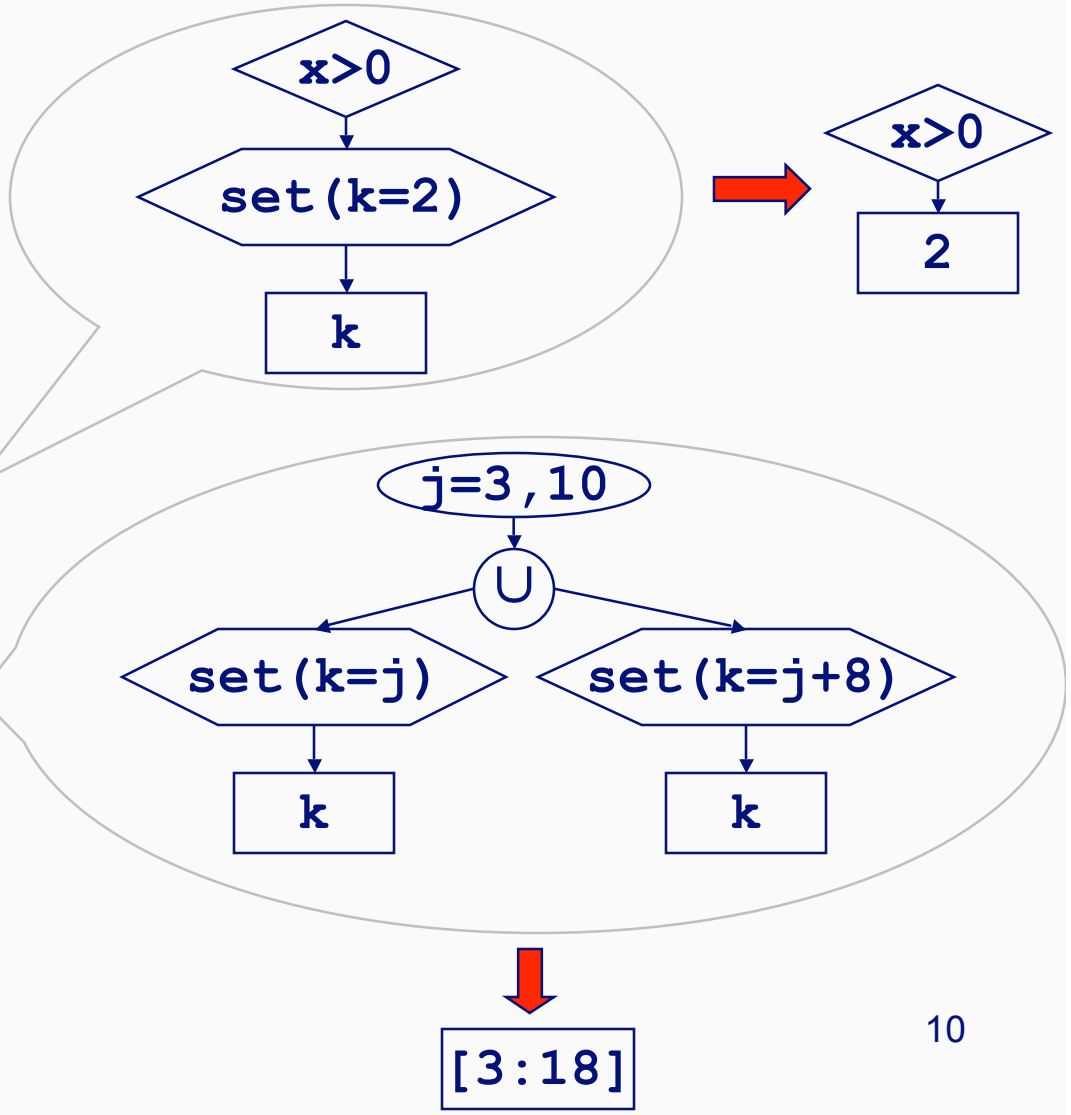
- *$@A_b^t$ is the array region defined before $A_{current}$ and reaching A_{total}*
  - *$@A_b^t = @A_{before} - @A_{current}$*
- *$@A_c^t$ is the array region defined by $A_{current}$ and reaching A_{total}*
  - *$@A_c^t = @A_{current}$*
- *Need analytical representation for @ sets!*

Expressing @ Sets: Array Region Representation

Parasol

```
Subroutine set (A, k, v)
  A(k) = v
End

...
Call   set(A, 1,   0)
If (x>0) Then
  Call set(A, 2,   1)
EndIf
Do j = 3, 10
  Call set(A, j,   3)
  Call set(A, j+8, 4)
EndDo
Print A(1) + A(5)
If (x>0) Then
  Print A(2)
EndIf
```



Run-time Linear Memory Access Descriptor (RT_LMAD)

T = { **LMAD**, ∩, ∪, –, **(**, **)**, **#**, **x**, Θ, **Gate**, **Recurrence**, **Call Site**}

N = { RT_LMAD }

S = RT_LMAD

P = { RT_LMAD → **LMAD** | **(**RT_LMAD**)**
RT_LMAD → RT_LMAD ∩ RT_LMAD
RT_LMAD → RT_LMAD ∪ RT_LMAD
RT_LMAD → RT_LMAD – RT_LMAD
RT_LMAD → RT_LMAD **#** **Gate**
RT_LMAD → RT_LMAD **x** **Recurrence**
RT_LMAD → RT_LMAD Θ **Call Site** }

LMAD = Start + [$Stride_1$:$Span_1$, $Stride_2$:$Span_2$, ...]

1. *Closed form for references in **If** blocks, **Do** loops, sequence of blocks*
2. *Closed with respect to set operations: difference, union*
3. *Control-flow sensitive and interprocedural*

δ Nodes for a Sequence of Blocks

```
Subroutine set (A, k, v)
  A(k) = v
End
```


```
Call    set(A, 1,   0)


If (x>0) Then
  Call set(A, 2,   1)
EndIf

Do j = 3, 10
  Call set(A, j,   3)
  Call set(A, j+8, 4)
EndDo
```

δ Nodes for a Sequence of Blocks

```
Subroutine set (A, k, v)
  A(k) = v
End
```


```
Call    set(A1, 1,    0)


If (x>0) Then
  Call set(A3, 2,    1)
EndIf

Do j = 3, 10
  Call set(A5, j,    3)
  Call set(A5, j+8,  4)
EndDo
```

δ Nodes for a Sequence of Blocks

```
Subroutine set (A, k, v)
  A(k) = v
End
```


```
Call    set(A1, 1,    0)


If (x>0) Then
  Call set(A3, 2,    1)
EndIf

Do j = 3, 10
  Call set(A5, j,    3)
  Call set(A5, j+8, 4)
EndDo
```

$[A_2, \{1\}] = \delta(A_0, [A_1, \{1\}])$

$[A_4, \{1\} \cup ((x>0)\#\{2\})] = \delta(A_0, [A_2, \{1\}], [A_3, (x<0)\#\{2\}])$

$[A_6, \{1\} \cup ((x>0)\#\{2\}) \cup [3:18]] = \delta(A_0, [A_4, \{1\} \cup ((x>0)\#\{2\})], [A_5, [3:18]])$

Print $A_6(100)$ ⇒ A_0
Print $A_6(3)$ ⇒ A_5
Print $A_6(11)$ ⇒ A_5
Print $A_6(1)$ ⇒ A_2

Definitions in Loops: μ nodes


```
Do j = 3, 10

  Call set(A, j,   3)

  Call set(A, j+8, 4)

EndDo
```

```
Print A(j-2)  ⟹ ?
Print A(3)    ⟹ ?
Print A(11)   ⟹ ?
```

Definitions in Loops: μ nodes


```
Do j = 3, 10

  Call set(A1, j,   3)

  Call set(A3, j+8, 4)

EndDo
```

$$[A_2, \{j\}] = \delta(A_5, [A_1, \{j\}])$$
$$[A_4, \{j\} \cup \{j+8\}] = \delta(A_5, [A_2, \{j\}], [A_3, \{j+8\}])$$

Definitions in Loops: μ nodes

Parasol

```
Do j = 3, 10


  Call set(A1, j,   3)

  Call set(A3, j+8, 4)


EndDo
```

$[A_5, [3:j-1] \cup [11:j+7]] = \mu(A_0, [A_1, [3:j-1]], [A_3, [11:j+7]])$

$[A_2, \{j\}] = \delta(A_5, [A_1, \{j\}])$

$[A_4, \{j\} \cup \{j+8\}] = \delta(A_5, [A_2, \{j\}], [A_3, \{j+8\}])$

$[A_6, [3:18]] = \delta(A_0, [A_5, [3:18]])$

Print $A_4(j-2)$ ⟹ ?

Print $A_6(3)$ ⟹ ?

Print $A_6(11)$ ⟹ ?

Definitions in Loops: μ nodes


```
Do j = 3, 10
  Call set(A1, j+2, 3)
  Call set(A3, j+8, 4)
EndDo
```

$[A_5, ?] = \mu(A_0, [A_1, ?], [A_3, ?])$

$$[A_n, @A_n] = \mu(A_0, [A_1, @A_1^n], [A_2, @A_2^n], \ldots, [A_m, @A_m^n]),$$

where

$$@A_k^n(j) = \bigcup_{i=1}^{j-1} [@A_k(i) - (Kill_s(i) \cup \bigcup_{l=i+1}^{j-1} Kill_a(l))],$$

$$Kill_s = \bigcup_{h=k+1}^{m} @A_h, Kill_a = \bigcup_{h=1}^{m} @A_h$$

$@A_k^n(j)$ is the array region defined by A_k that reaches A_n upon entry to iteration j.

Definitions in Loops: Iteration Vectors

- Iteration vectors
  - For a given array element, which iteration wrote it last?
  - Important for: Forward substitution, Last value assignment
  - Not important for: Privatization
  - Hard to express when loop nests span subroutines

- We express the dual entity
  - For a given iteration *j*, what is the set of all memory locations last defined at *j*?
  - Example: Last value assignment
    - Compute LVA(j) as set of memory locations

```
Subroutine set (A, k, v)
  A(k) = v
End
```

Control Dependence: π nodes


```
If (x>0) Then

  Call set(A, 2, 1)

EndIf

If (x>0) Then

  Print A(2)
EndIf
```

Original code

```
If (x>0) Then
 [A1,∅]= π(x>0, A0)
 Call set(A2, 2, 1)
 [A3, {2}]= δ(A1, [A2, {2}])
EndIf
[A4, (x>0)#{2}] = δ(A0,[A3,(x>0)#{2}])
If (x>0) Then
  [A5,∅] = π(x>0, A4)
  Print A5(2)
EndIf
```

Array SSA

- *Different from SSA: new name without definition.*
- *Essential to control-sensitive data flow analysis.*

Reaching Definitions

- Given:
  - An SSA name A_u
  - An array region *Use*
  - A block to limit the search, *GivenBlock*
- Find $[A_1, R_1], [A_2, R_2], \ldots [A_n, R_n], [\perp, R_0]$, such that:
  - *Use* = $R_1 \cup R_2 \cup \ldots \cup R_n \cup R_0$
  - $R_j \cap R_k = \phi, \forall 1 \leq j \neq k \leq n$
  - Region R_k was defined by A_k, k=1,n
  - R_0 was not defined within *GivenBlock*

- Example:
  - Privatization: *GivenBlock* = loop body, prove R_0 empty

Reaching Definitions

$$[A_{total}, @A_t] = \delta(A_{undef}, [A_{before}, @A_b^t], [A_{current}, @A_c^t])$$

Algorithm SearchRD(A_t, *Use*, *GivenBlock*)

If (A_t **not in** *GivenBlock*) **Then** Report [⊥, *Use*]; **Stop**

If (A_t **not an SSA** gate**) Then** Report [A_t, *Use*]; **Stop**

Call SearchRD(A_{before}, $@A_b^t \cap Use$, *Block*(A_{before}))

Call SearchRD($A_{current}$, $@A_c^t \cap Use$, *Block*($A_{current}$))

Call SearchRD(A_{undef}, $Use - @A_t$, *Block*(A_{undef}))

Special operations:

- Expand descriptors at μ gates
- Add conditionals at π gates

Array Constant Propagation

- Array constant collection
  - Attach values to *reaching definitions* sets
  - Unite sets with the same constant value
- Constant propagation and substitution
  - Full loop unrolling
  - Subprogram specialization
  - Aggressive dead code elimination

Constant Collection

- Intraprocedural collection:
  - Use the *SearchRD* algorithm
  - Attach values to reaching definitions sets
  - Collect values from assignment statements
  - Unite sets with same attached value

- Interprocedural collection
  - Push available sets at call sites
  - Collect intraprocedurally
  - Return collected constants back to calling context

Constant Propagation

- Full loop unrolling
  - Compute iteration count based on available constant values
- Subprogram specialization
  - Constants available only at certain call sites
- Aggressive dead code elimination
  - Dead branches
  - Dead assignments
  - Multiplications with 0 or 1; additions of 0

Experimental Results

	Pentium	PA-RISC	Power	MIPS
QCD2	14.0%	17.4%	12.8%	15.5%
173.APPLU	20.0%	4.6%	16.4%	10.5%
048.ORA	1.5%	22.8%	11.9%	20.6%
107.MGRID	12.5%	8.9%	6.4%	12.8%

Intel PC:	*Pentium 4 (520)*	*2.8 GHz*
HP 9000/R390:	*PA-8200*	*200 MHz*
IBM Regatta P690:	*Power 4*	*1.3 GHz*
SGI Origin 3800:	*MIPS R1400*	*500 MHz*

Conclusions

- Array SSA
  - Analytical, scalable, explicit element-level data flow information
- Reaching definitions algorithm
  - Find matching array subregion for each reaching definition of a *use*
- Array constant propagation
  - Speedup on four benchmark programs
- Future uses: array privatization, dependence, liveness