Array SSA Form and its use in Program Analysis and Transformation

Vivek Sarkar
Rice University
vsarkar@rice.edu

Work done jointly with
K. Knobe & S. Fink
Array SSA Form and Related Work

<table>
<thead>
<tr>
<th></th>
<th>Control Flow</th>
<th>Array Subscripts</th>
<th>Renaming + Static Single Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Data Flow</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar SSA</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Array Dependence</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Array Data Flow</td>
<td>Limited</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Array SSA</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Advantages of Array SSA Form

Renaming + Element-level use-def information

- Increases analysis and reordering potential for programs with array, structure and pointer variables
  - Value analysis, parallelism, code motion

- Enables speculative execution
  - Executing more than specified, and selecting the right value

- Enables element-level optimizations
  - Executing less than specified using element-level liveness
References

- Implementation of Array SSA Form in Jikes RVM
Outline

- Array SSA form vs. Traditional SSA form
- Conditional Constant Propagation using Array SSA form
- Loop Parallelization using Array SSA form
- Load elimination of object fields and array accesses using Array SSA form
- Conclusions and Future Work
Example Program
(Control Flow Graph)
Traditional Scalar SSA Form

\[ X_1 = \ldots \]
\[ \ldots = X_1 \ldots \]
\[ X_2 = \ldots \]
\[ \ldots = X_2 \ldots \]
\[ X_3 = \ldots \]
\[ \ldots = X_3 \ldots \]
\[ X_4 = \Phi(X_1, X_3) \]
\[ \ldots = X_4 \ldots \]

\( \Phi(x_1, x_3) \) is not a pure function. It has implicit parameters.
Making $\Phi$ functions executable in Scalar SSA Form through @ variables (vector timestamps)

Original program
(for-loop example):
for $i := 1$ to $m$ do
  $S := \ldots$
  if ( . . . ) then
    $S := \ldots$
  end if
end for

$S_3 = \text{if } (@S_2 \geq @S_1)\text{ then } S_2 \text{ else } S_1 \text{ end if}$

Array SSA form:
$@S_1 := ( ); @S_2 := ( )$
for $i := 1$ to $m$ do
  $S_1 := \ldots$
  $@S_1 := ( i )$
  if ( . . . ) then
    $S_2 := \ldots$
    $@S_2 := ( i )$
  end if
end for
$S_3 := \Phi(S_2, @S_2, S_1, @S_1)$
$@S_3 := \max(@S_2, @S_1)$

$\Phi$ function was optimized in this example via copy propagation
Scalar SSA form does not work for Arrays

Scalar SSA does not support
*preserving definitions*

Scalar SSA $\Phi$ functions do not support
*element-level merge*
Array SSA Form --- Definition $\Phi$

**Definition** $\Phi = \text{data merge of array element modified in current def with array elements of previous def}$

**Original Program**

X[1:n] = ...

... 

X[k] = ...

... = X[j]

**Array SSA Form**

X₁[1:n] = ...

...

X₂[k] = ...

X₃ = $d\Phi(X₂,X₁)$

... = X₃[j]

\[
\begin{array}{cccc}
A & B & C & D \\
\hdashline
\downarrow & \downarrow & \downarrow & E \\
A & B & C & E
\end{array}
\]
@ variables for Arrays

Original program:

for i := ... do
    . . .
    X[k] := . . .
    ... := X[j]
end for

Array SSA form:

@X_1[*] := ( )
@X_2[*] := ( )
for i := ... do
    X_1[1:n] := ...
    @X_1[1:n] := ( i )
    X_2[k] := ... 
    @X_2[k] := ( i )
end for
X_3 := \Phi(X_2, @X_2, X_1, @X_1)
@X_3 := \max(@X_2, @X_1)
... := X_3[j]
Outline

- Array SSA form vs. Traditional SSA form
- Conditional Constant Propagation using Array SSA form
- Loop Parallelization using Array SSA form
- Load elimination of object fields and array accesses using Array SSA form
- Conclusions and Future Work
Lattice Values in Constant Propagation using Scalar SSA form

\[ L(k_1) = 99 \]

\[ k_1 := 99 \]

\[ \ldots \]

\[ \text{if ( false ) then} \]

\[ k_2 := \ldots \]

\[ \text{end if} \]

\[ L(k_2) = \top \]

\[ k_3 := \phi(k_2, k_1) \]

\[ L(k_3) = L(k_1) \mathbin\square L(k_2) \]

\[ = 99 \mathbin\square \top \]

\[ = 99 \]

SET(\top) = \{ \}

SET(99) = \{ 99 \}

\[ \ldots 99 \ 100 \ldots \]

SET(\bot) = Universal set of constants
Extending the Lattice for Array Values

Lattice value for array variable
= finite list of constant index-value pairs

Lattice element represents a set of index-value pairs as shown below

It is always safe to approximate a lattice element by a lower value
⇒ Lattice height can be bounded as a compiler parameter

\[
\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle \\
\Rightarrow \text{SET}(\mathcal{L}(A)) = \{ (i_1, e_1), (i_2, e_2), \ldots \} \cup (\mathcal{U}_{ind}^A - \{i_1, i_2, \ldots \}) \times \mathcal{U}_{elem}^A
\]
Use Partial Array SSA Form (w/o @ variables) for Analysis

Original code:
X := ...
if (...) then
    X[k] := ...
endif

Partial Array SSA form:
X_0 := ...
if (...) then
    X_1[k] := ...
    X_2 := dφ(X_1, X_0)
endif
X_3 := mφ(X_2, X_0)

Definition φ
x_2[j] =
    if (j == k) then
        x_1[j]
    else
        x_0[j]
    endif

Merge φ
x_3[j] = x_2[j] or x_0[j]
Conditional Constant Propagation using Array SSA form (Example)

\[
i := 5
\]
\[
\ldots
\]
\[
\text{if } (i = 5) \text{ then}
\]
\[
k := 3
\]
\[
X_1[k] := 99
\]
\[
X_2 := d\phi(X_1,X_0)
\]
\[
\text{endif}
\]
\[
X_3 := m\phi(X_2,X_0)
\]
\[
X_4[i] := 101
\]
\[
X_5 := d\phi(X_4,X_3)
\]
\[
y := X_5[k]
\]
\[
L(i) = 5
\]
\[
L(i=5) = \text{TRUE}
\]
\[
L(k) = 3
\]
\[
L(X_1) = <(3,99)>
\]
\[
L(X_2) = <(3,99)>
\]
\[
L(X_3) = <(3,99)>
\]
\[
L(X_4) = <(5,101)>
\]
\[
L(X_5) = <(3,99), (5,101)>
\]
\[
L(X_5[k]) = 99
\]
Summary of Constant Propagation using Array SSA form

- Algorithm performs constant propagation through *array elements*.
- Execution time of algorithm is *linear* in size of Array SSA form.
- Algorithm propagates constants for arrays only when array element has *constant index and constant value*. (SAS 2000 paper shows how to propagate constants through symbolic indices, by determining equality & inequality of index expressions.)
Constants Propagation with Symbolic Index Values (Sneak Preview)

Let $V_i$, $V_j$, $V_k$ be value numbers for $i$, $j$, $k$
and assume $V_i = V_j$ and $V_i \neq V_k$

\[ X_1[j] := 100 \]
\[ X_2 := \phi(X_1, X_0) \]
\[ \ldots := X_2[j] \]

\[ \mathcal{L}(X_0) = < \ldots (V_i, 43) (V_k, 12) \ldots > \]

\[ \mathcal{L}(X_2) = \text{INSERT}(\mathcal{L}(X_0), (V_j, 100)) \]
\[ = < \ldots (V_j, 100) (V_k, 12) \ldots > \]
Outline

- Array SSA form vs. Traditional SSA form
- Conditional Constant Propagation using Array SSA form
- Loop Parallelization using Array SSA form
- Load elimination of object fields and array accesses using Array SSA form
- Conclusions and Future Work
Compile-time vs. Run-time usage of Array SSA form

**Compile-time usage**
- Use Partial Array SSA form with merge $\phi$ and definition $\phi$ functions
- Compile-time space is linearly proportional to scalar SSA space
- No overhead incurred at run-time

**Run-time usage**
- Use Full Array SSA form with merge $\phi$’s, definition $\phi$’s and @ variables
- Overhead depends on which @ variables and $\phi$ functions are made manifest at run-time
Loop Parallelization using Array SSA form

- **Input**
  - Loop with no loop-carried true dependence (no recurrence)
  - Can have *arbitrary* loop-carried anti and output dependences (storage-related dependences)

- **Output**
  - Parallelized execution and finalization loops based on Array SSA form
Example Loop in Array SSA Form

\( X_0[*] := \ldots \)
do \( i := \ldots \)
    \( X_1 := \phi(X_0, X_4) \)
    if (...) then
        \( X_2[f(i)] := rhs(i) \)
        \( X_3 := \phi(X_2, X_1) \)
    end if
end do
\( X_4 := \phi(X_3, X_1) \)
end do
\( X_5 := \phi(X_4, X_0) \)

Initial Array SSA form

\( X_0[*] := \ldots \)
do \( i := \ldots \)
    if (...) then
        \( X_2[f(i)] := rhs(i) \)
    end if
end do
\( X_5 := \phi(X_2, X_0) \)

Simplified
(Assumes no read of \( X \) in original loop)
Simplified Version with @ variables inserted

@X_2[*] := ( )

X_0[*] := …
@X_0[*] := (1)
do i := …
    if (. . .) then
        X_2[f(i)] := rhs(i)
        @X_2[f(i)] := (1,i)
    endif
enddo

X_4 := Φ(X_2, @X_2, X_0, @X_0)

X_4[j] = if (@X_2[j] >= @X_0[j])
    then X_2[j]
    else X_0[j]
end if
Parallelization using Array SSA Form

**Step 1:** Array SSA form naturally partitions a loop into execution and finalization phases:

```plaintext
do i := ...
    if (. . .) then
        x2[f(i)] := rhs(i)
        @x2[f(i)] := ... endif
enddo

x4 := Φ(x2, @x2, x0, @x0)
```

**Execution Phase**

**Finalization Phase**
Iteration Parallelism

**Step 2:** Use *array expansion* to parallelize both loops (degree of expansion can be contracted to degree of parallelism exploited)

### Execution [iteration space]

- $\forall i \in \mathbb{I}, \forall j \in \mathbb{J}$
  
  $X[i,j] := \text{rhs}(i)$
  
  $\forall i \in \mathbb{I}, \forall j \in \mathbb{J}$

### Finalization [data space]

- $\forall i \in \mathbb{I}, \forall j \in \mathbb{J}$
  
  $X[i,j] := \text{rhs}(i)$
  
  $\forall i \in \mathbb{I}, \forall j \in \mathbb{J}$

- $\forall i \in \mathbb{I}, \forall j \in \mathbb{J}$
  
  $X[i,j] := X[0,i,j]$
Rasterization Example
(Array SSA Renaming performed on display buffer)

Processor 1
(N/4 polygons)

Processor 2
(N/4 polygons)

Processor 3
(N/4 polygons)

Processor 4
(N/4 polygons)

B_5 := F( B_1, @B_1, B_2, @B_2, B_3, @B_3, B_4, @B_4 )
### Rasterization Example

#### Time in Seconds

<table>
<thead>
<tr>
<th>No. of Polygons</th>
<th>Serial</th>
<th>Parallel ( P = 1 )</th>
<th>Parallel ( P = 4 )</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>3.6</td>
<td>3.8</td>
<td>1.4</td>
<td>2.6</td>
</tr>
<tr>
<td>50,000</td>
<td>17.1</td>
<td>17.4</td>
<td>4.8</td>
<td>3.6</td>
</tr>
<tr>
<td>100,000</td>
<td>34.5</td>
<td>34.0</td>
<td>9.1</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Execution times measured on a Digital AlphaServer 4100 SMP with 400 MHz Alpha 21164 processors.
Region Parallelism

do i = ...
x(…) =
enddo

do i = ...
if ... then
    x(i) = t + ...
    t = x(i)
endif
enddo
Region Parallelism

do i = ...
x1(...) =
@x1(...) = i
dendo
do i = ...
if ... then
x2(i) = t + ...
t = x2(i)
@x2(i) = i
endif
dendo

x3 = \phi(x2, @x2, x1, @x1, x0)

x3(j) =
if (@x2(j) \neq \bot) then
x2(j)
elseif (@x1(j) \neq \bot) then
x1(j)
else
x0(j)
endif
Many Possible Factorings

\[ \phi \text{ computation is associative} \]
Outline

- Array SSA form vs. Traditional SSA form
- Conditional Constant Propagation using Array SSA form
- Loop Parallelization using Array SSA form
- Load elimination of object fields and array accesses using Array SSA form
- Conclusions and Future Work
Heap Arrays for analysis of Java programs

Model accesses to field x as accesses to a 1-D Heap Array

GETFIELD p.x -> read of x[p]
PUTFIELD q.x -> write of x[q]

Model accesses to 1-D Java array as accesses to a 2-D Heap Array for array type

e.g., consider arrays of type double[ ]
ALOAD of a[i] -> read of double [a, i]
ASTORE of a[i] -> write of double [a, i]

Leverages type system for disambiguation

Distinct heap arrays for distinct fields and distinct array types
Use single heap array for weakly typed languages
Extended Array SSA example

introduce "Heap" array $x$ for each field $x$

class Z { int x; }

... 
Z a = new Z()
if (...) {
    a.x = 1
} else {
    a.x = 2
}
y = a.x

\[
\begin{align*}
\text{class Z} & \{ \text{int } x; \} ; \\
\ldots & \\
a_9 & = \text{new Z}() \\
x_1[a_9] & = 0 \\
\text{if (...) } \{ \\
x_2[a_9] & = 1 \\
x_3 & = \phi(x_1, x_2) \\
\} \text{ else } \{ \\
x_4[a_9] & = 2 \\
x_5 & = \phi(x_1, x_4) \\
\} \\
x_6 & = \phi(x_3, x_5) \\
y & = x_6[a_9]
\end{align*}
\]
Definitely Same / Definitely Different Relations among Value Numbers

- Assign each scalar \( s \) a value number \( V(s) \)
  - Global Value Numbering

- Definitely Same (DS)
  - if \( V(x) = V(y) \), \( x \) and \( y \) have the same value wherever both are defined

- Definitely Different (DD): construct "equivalence classes" of value numbers that must be distinct
  - pointers from different allocation sites
  - "pre-existing" objects
  - "uniformly-generated" index values

- Equivalence class approach to computing DS and DD is more efficient than points-to graphs
Intraprocedural Load Elimination --- Example

Original Program

\[
\begin{align*}
p & := \text{new } Z \\
q & := \text{new } Z \\
r & := p \\
\ldots \\
p.x & := \ldots \\
q.x & := \ldots \\
\ldots & := r.x
\end{align*}
\]

Transformed Program

\[
\begin{align*}
p & := \text{new } Z \\
q & := \text{new } Z \\
r & := p \\
\ldots \\
T1 & := \ldots \\
p.x & := T1 \\
q.x & := \ldots \\
\ldots & := T1
\end{align*}
\]
Index Propagation Example

compute $L(H) = \{ \text{set of value numbers } v \text{ s.t. } H[v] \text{ is available} \}$

Extended Array SSA representation

```
p := new Z
q := new Z
r := p
Z.x_1[p] := ...
Z.x_2 = d_\phi (Z.x_0, Z.x_1)
Z.x_3[q] := ...
Z.x_4 = d_\phi (Z.x_2, Z.x_3)
... = Z.x_4[r]
Z.x_5 = u_\phi (Z.x_3, Z.x_4)
```

Dataflow Solution

```
DD (p,q) = true
DS (p,r) = true
L(Z.x_0) = {}
L(Z.x_1) = { V(p) }
L(Z.x_2) = { V(p) }
L(Z.x_3) = { V(q) }
L(Z.x_4) = { V(p), V(q) }
L(Z.x_5) = { V(p), V(q) }
```
Fraction of Dynamic Memory Operations eliminated

(memory op = getfield, putfield, getstatic, putstatic, aload, or astore)
Reduction in running time on 166MHz PowerPC, AIX 4.3, 1GB
Outline

- Array SSA form vs. Traditional SSA form
- Conditional Constant Propagation using Array SSA form
- Loop Parallelization using Array SSA form
- Load elimination of object fields and array accesses using Array SSA form
- Conclusions and Future Work
Other Topics Discussed in Papers

- Data flow equations for array constant propagation
- Optimization of $\Phi$ functions and $\omega$ variables
- Parallelization with Speculative Execution
- Parallelism across Regions
- Modeling Structures as Arrays
Other SSA-related work that I’ve been involved in


Conclusions

Array SSA form is an intermediate form that integrates

- Control flow analysis
- Index analysis
- Renaming

- Increases reordering potential
- Enables speculative execution
- Enables element-level optimizations
Future Work

- Study of legal $\phi$ and $@$ transformations
- Extend scope of other optimizations to array elements
- Program slicing w.r.t. array elements
- Extend framework to perform deeper analysis of pointer structures
- Extend constant propagation algorithm to type propagation in strongly-typed OO languages
- Use in analysis and transformation of parallel X10 and Habanero-Java programs (Habanero project)
- Use in optimization of array accesses in C and Fortran programs (PACE project)
- . . .
Habanero Project Overview (habanero.rice.edu)

Parallel Applications
(Seismic analysis, Medical imaging, Finite Element Methods, …)

Challenge: Develop new programming technologies and pedagogical foundations for portable parallelism on future multicore hardware

1) Habanero Programming Languages
2) Habanero Static Compiler & Parallel Intermediate Representation
3) Habanero Runtime & Dynamic Compiler

Two-level programming model
Implicitly Parallel Coordination Language for Joe, CnC (Intel Concurrent Collections) + Explicitly Parallel Programming Languages for Stephanie, Habanero-Java (from X10 v1.5) and Habanero-C

Foreign Code (Matlab, Java, C, C++, Fortran, CUDA) Foreign Function Interface

Multicore Platforms
(Cell, Clearspeed, Cyclops, GeForce, Niagara, Opteron, Power, Xeon, …)
Habanero Static Parallelizing & Optimizing Compiler

Interprocedural Analysis

PIR Analysis & Optimization

Classfile Transformations

Front End

Foreign Function Interface

Sequential C, Fortran, Java, ...

Habanero Languages

High Level PIR Generation and Optimizations

Middle Level PIR Generation and Optimizations

Low Level PIR: code generation for different runtime systems

Internal Object Translator (e.g. phaser, region, dist)

Work-sharing RT

Help-first Work-stealing RT

Work-first Work-stealing RT

Java bytecode generation

Load Elimination

MPIR to HPIR

May-Happen-In-Parallel Analysis

ForEach Loop Chunking

High Level AST

Portable Managed Runtime

Platform-specific static compiler

Partitioned Code

for targeting accelerators & high-end computing)
Habanero Team Pictures
DARPA awards $16 million to Rice University to improve compilers

The Defense Advanced Research Projects Agency (DARPA), as part of its Architecture Aware Compiler Environment Program, has awarded Rice University $16 million to develop a new set of tools that can improve the performance of virtually any application running on any microprocessor. …

From left to right:
Vivek Sarkar, Keith Cooper, John Mellor-Crummey, Krishna Palem and Linda Torczon.

Subcontractors include OSU (Sadayappan), TI (Tatge), Stanford (Lele), ETI
Send email to Vivek Sarkar (vsarkar@rice.edu) if you are interested in a postdoc, research scientist, or programmer position in the Habanero or PACE projects!