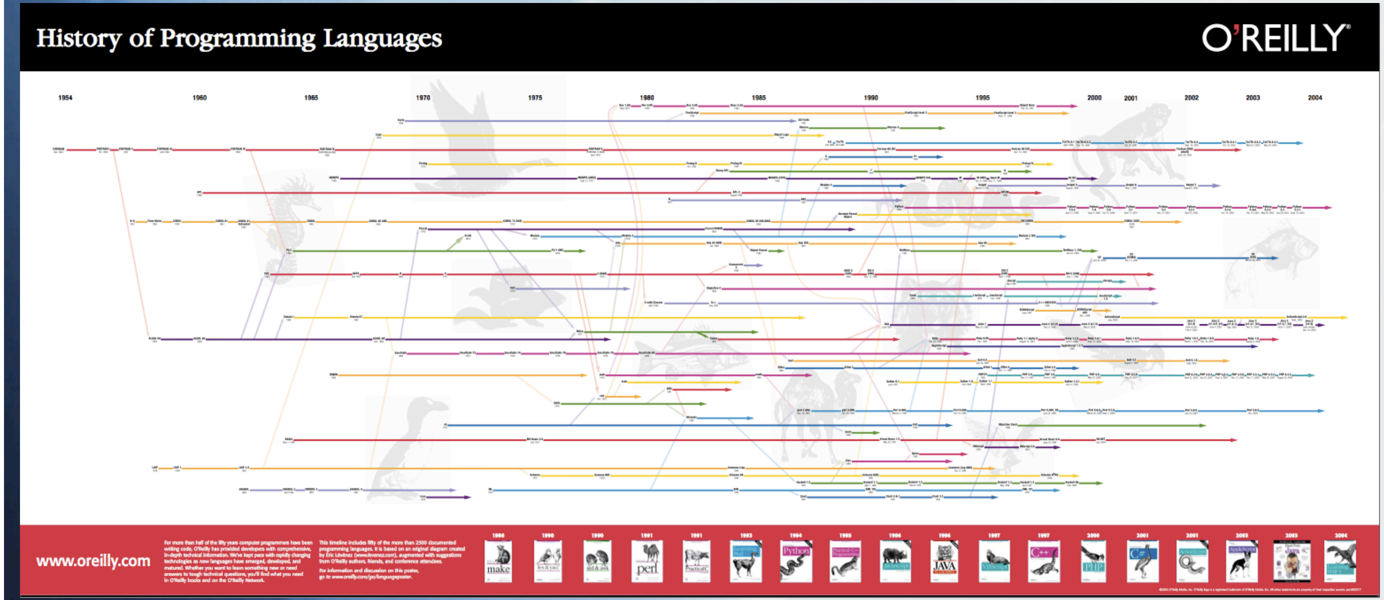
RICE
George R. Brown
School of Engineering
Computer Science

Array SSA Form and its use in Program Analysis and Transformation

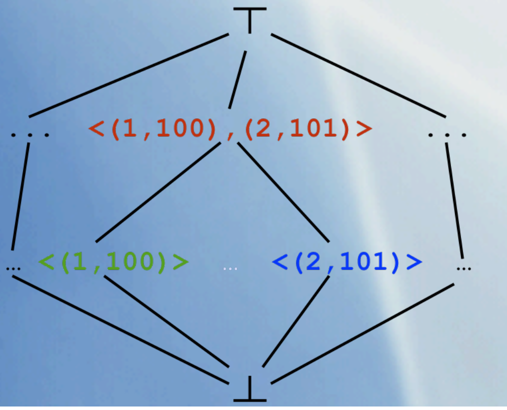


Vivek Sarkar

Rice University

vsarkar@rice.edu

Work done jointly with
K. Knobe & S. Fink

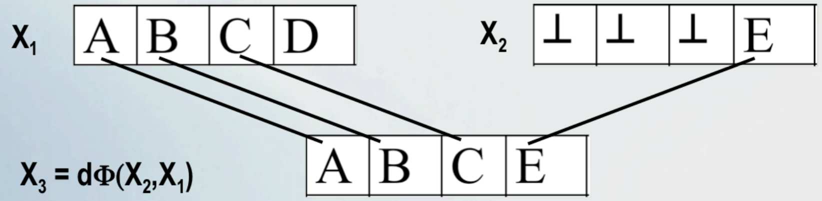


$\top$

. . . <(1,100),(2,101)> . . .

... <(1,100)> ... <(2,101)> ...

$\bot$



X_1 A B C D

X_2 $\bot$ $\bot$ $\bot$ E

$X_3 = d\Phi(X_2,X_1)$ A B C E

Array SSA Form and Related Work

	Control Flow	Array Subscripts	Renaming + Static Single Assignments
Classical Data Flow	X		
Scalar SSA	X		X
Array Dependence		X	
Array Data Flow	Limited	X	
Array SSA	X	X	X

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

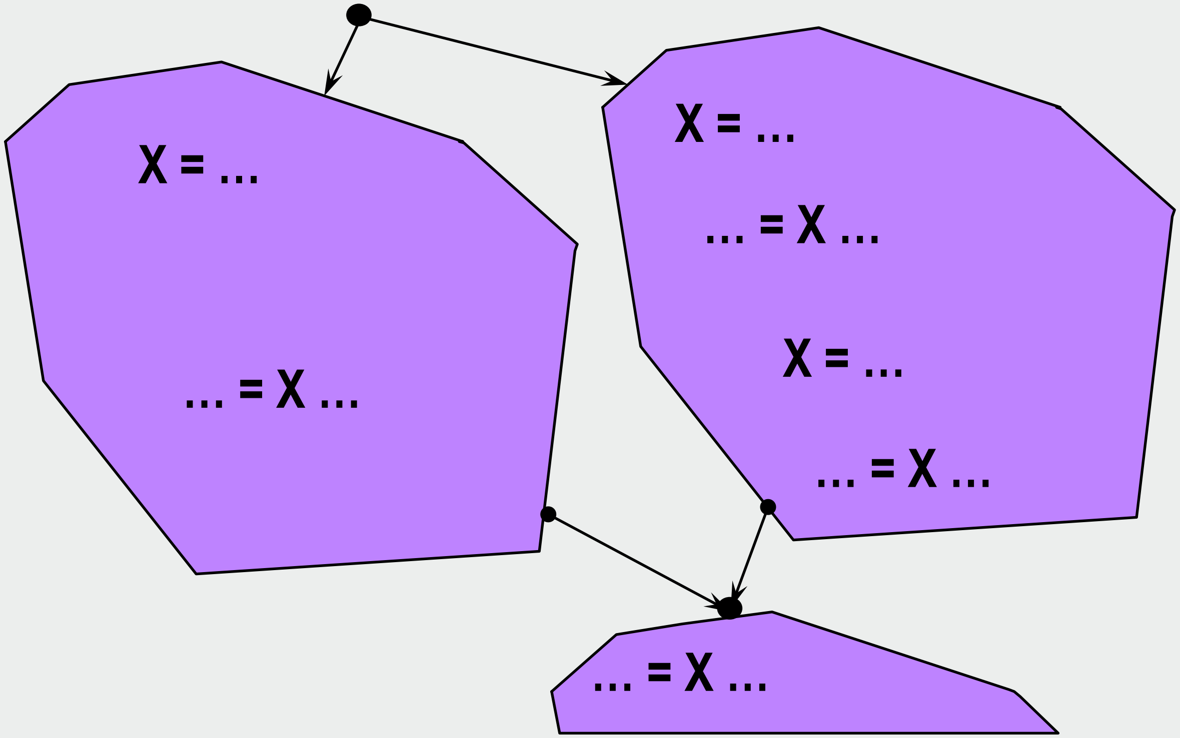
References

- Implementation of Array SSA Form in Jikes RVM
- Unified Analysis of Array and Object References in Strongly Typed Languages. S.Fink, K.Knobe, V.Sarkar. Proceedings of the 2000 Static Analysis Symposium (SAS ’00), October 2000.
- Enhanced Parallelization via Analyses and Transformations on Array SSA Form. K.Knobe, V.Sarkar. Workshop on Compilers for Parallel Computers (CPC), Jan 2000.
- Enabling Sparse Constant Propagation of Array Elements via Array SSA Form. V.Sarkar and K.Knobe. Proceedings of the 1998 Static Analysis Symposium (SAS ’98), October 1998.
- Array SSA form and its use in Parallelization. Kathleen Knobe and Vivek Sarkar. Proceedings of the 25th ACM SIGPLAN -SIGACT Symposium on Principles of Programming Languages, San Diego, California, January 1998.

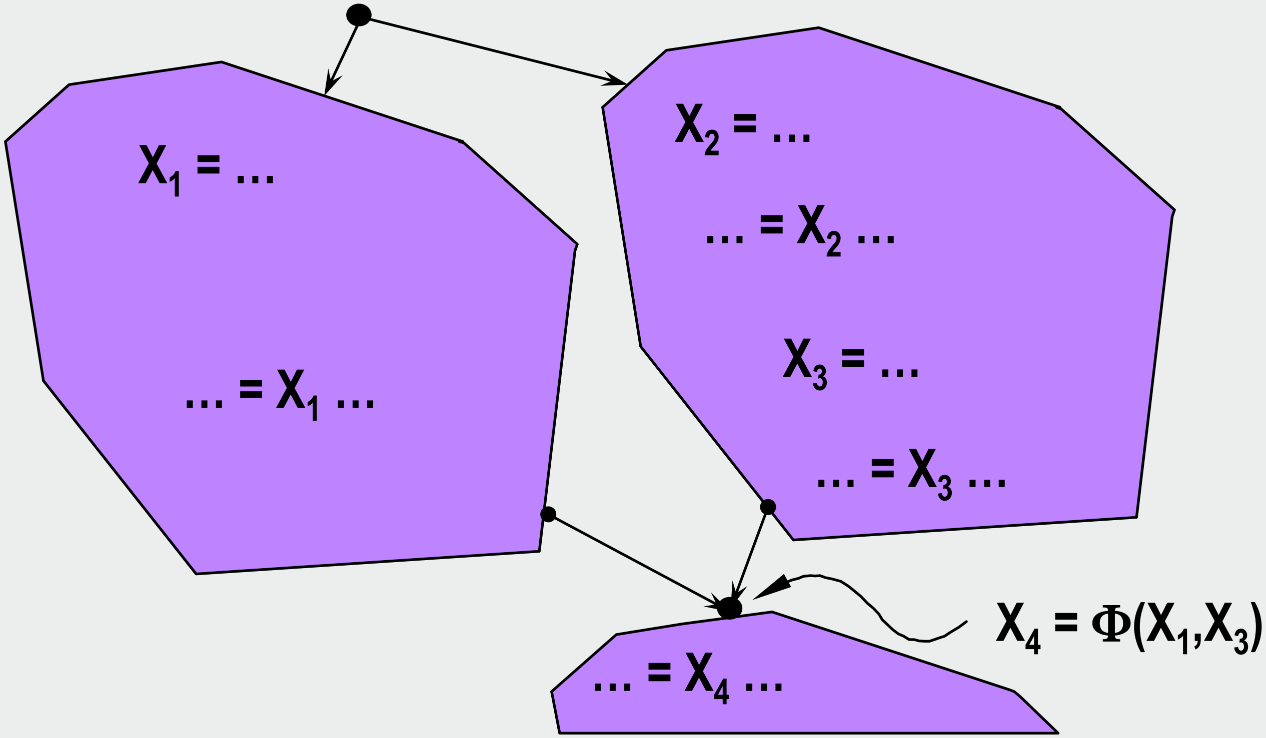
Outline

- 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 Φ functions executable in Scalar SSA Form through @ variables (vector timestamps)

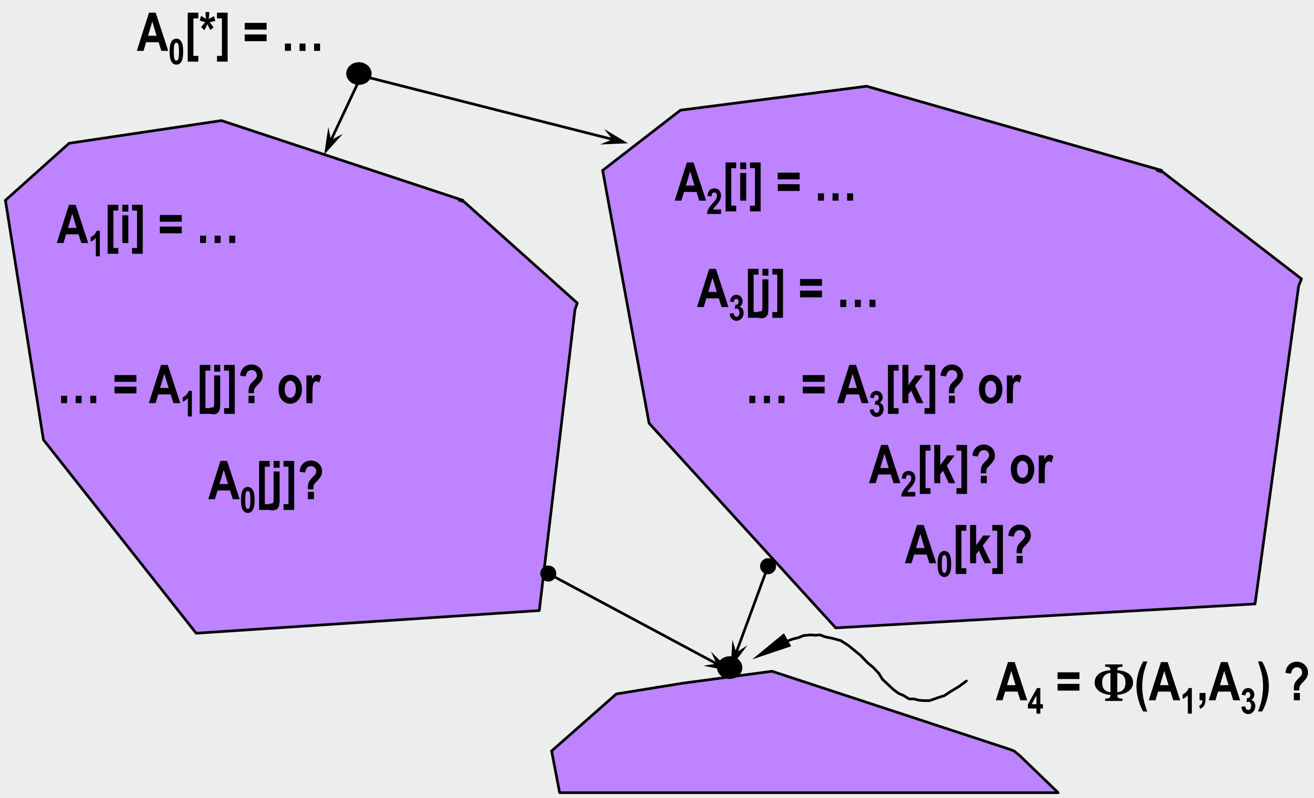
```
Original program
(for-loop example):
for i := 1 to m do
    S := …
    if ( . . .) then
        S := …
    end if
end for
```

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

```
$S_3$ = if ( $@S_2$ >= $@S_1$)
        then $S_2$
        else $S_1$
        end if
```

Φ function was optimized in this example via copy propagation

Scalar SSA form does not work for Arrays



$A_0[*] = \ldots$

$A_1[i] = \ldots$

$\ldots = A_1[j]?$ or

$A_0[j]?$

$A_2[i] = \ldots$

$A_3[j] = \ldots$

$\ldots = A_3[k]?$ or

$A_2[k]?$ or

$A_0[k]?$

$A_4 = \Phi(A_1, A_3)$?

Scalar SSA does not support *preserving definitions*

Scalar SSA Φ functions do not support *element-level merge*

Array SSA Form --- Definition ϕ

Definition Φ = *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

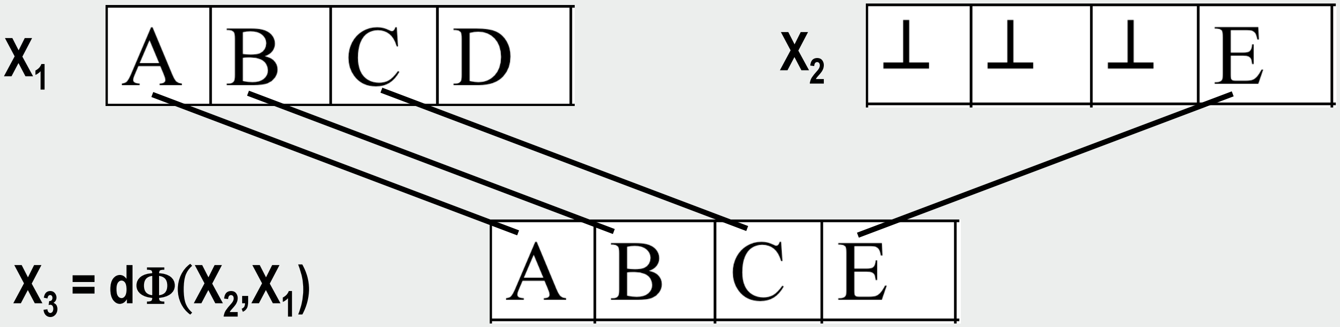
X_1[1:n] = . . .

. . .

X_2[k] = . . .

$X_3 = d\Phi(X_2,X_1)$

... = X_3[j]



@ variables for Arrays

```
Original program:

for i := ... do
   X[1:n] := . . .
   . . .
   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]
```

```
$X_3$[j] = if ( $@X_2$[j] >= $@X_1$[j] )
        then $X_2$[j]
        else $X_1$[j]
        end if
```

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

$L(k_1) = 99$

$k_1 := 99$

. . .

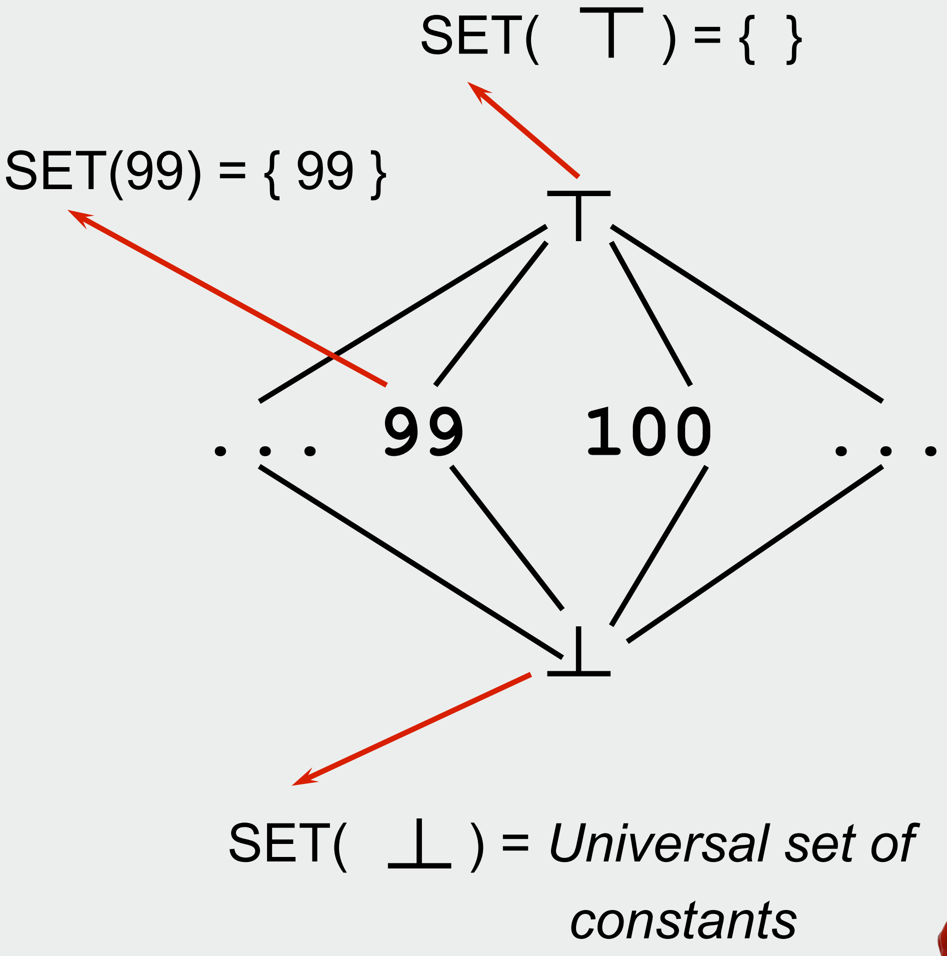
if (**false**) **then**

  $k_2 := \ldots$

end if $L(k_2) = \top$

$k_3 := \phi(k_2, k_1)$

$$L(k_3) = L(k_1) \sqcap L(k_2)$$
$$= 99 \sqcap \top$$
$$= 99$$



SET($\top$) = { }

SET(99) = { 99 }

SET($\bot$) = *Universal set of constants*

Extending the Lattice for Array Values

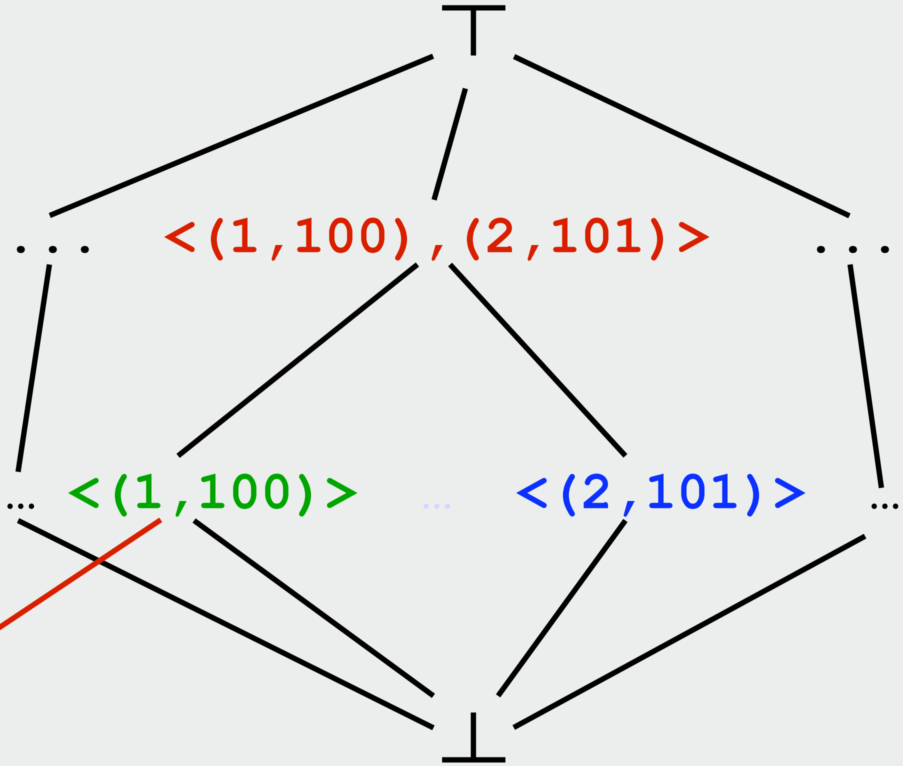
Lattice value for array variable

= 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



$$\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle$$

$$\Rightarrow \quad \text{SET}(\mathcal{L}(A)) = \{(i_1, e_1), (i_2, e_2), \ldots\} \cup (\mathcal{U}^A_{ind} - \{i_1, i_2, \ldots\}) \times \mathcal{U}^A_{elem}$$

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	L(i) = 5
. . .	
if (i = 5) then	L(i=5) = TRUE
k := 3	L(k) = 3
X_1[k] := 99	L(X_1) = <(3,99)>
X_2 := dϕ(X_1,X_0)	L(X_2) = <(3,99)>
endif	
X_3 := mϕ(X_2,X_0)	L(X_3) = <(3,99)>
X_4[i] := 101	L(X_4) = <(5,101)>
X_5 := dϕ(X_4,X_3)	L(X_5) = <(3,99),(5,101)>
y := X_5[k]	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* $V_i \neq V_k$

$$\mathcal{L}(\mathrm{X}_0) = < \ldots (\mathrm{V}_\mathrm{i}, 43)(\mathrm{V}_\mathrm{k}, 12) \ldots >$$

$\mathrm{X}_1[\,\mathrm{j}\,] := 100$
$\mathrm{X}_2 := \phi(\mathrm{X}_1, \mathrm{X}_0)$
$\ldots := \mathrm{X}_2[\,\mathrm{j}\,]$

$$\begin{aligned}\mathcal{L}(\mathrm{X}_2) &= \mathrm{INSERT}(\mathcal{L}(\mathrm{X}_0), (\mathrm{V}_\mathrm{j}, 100)) \\ &= < \ldots (\mathrm{V}_\mathrm{j}, 100)(\mathrm{V}_\mathrm{k}, 12) \ldots >\end{aligned}$$

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 ϕ and definition ϕ 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 ϕ's, definition ϕ's and @ variables
- Overhead depends on which @ variables and ϕ 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

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

Initial Array SSA form

```
$X_0$[*] := ...
do i := ...
   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
$X_4$ := $\Phi$($X_2$,@$X_2$,$X_0$,@$X_0$)
```

```
$X_4$[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:

```
do i := …
  if (. . .) then
    $X_2$[f(i)] := rhs(i)
    @$X_2$[f(i)] := . . .
  endif
enddo
```

Execution Phase

```
$X_4$ := $\Phi$($X_2$,@$X_2$,$X_0$,@$X_0$)
```

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]

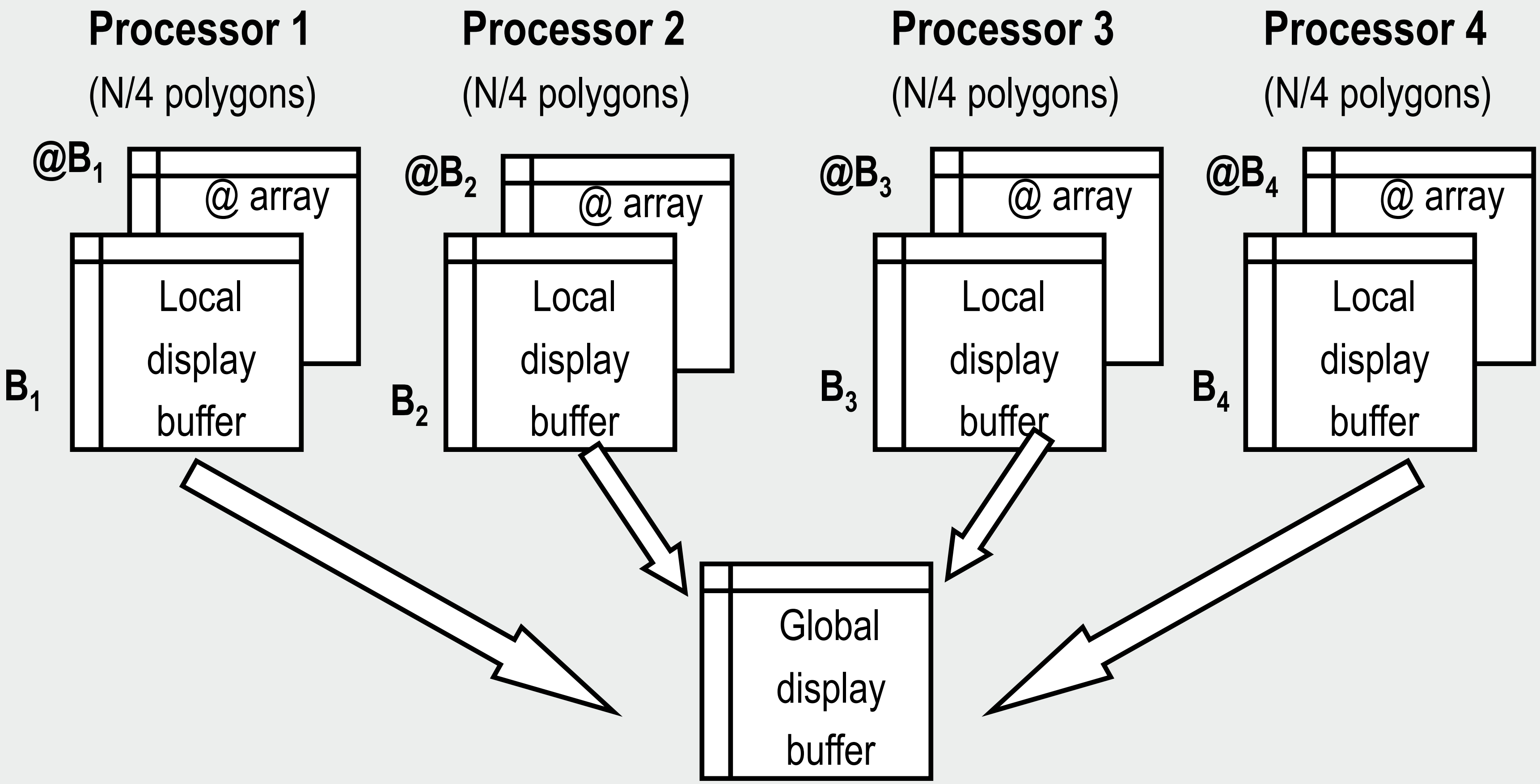
```
@$X_2$[*,*] := -1
doall i := …
  if (. . .) then
    $X_2$[f(i),i] := rhs(i)
    @$X_2$[f(i),i] := i
  endif
enddo
```

Finalization [data space]

```
doall j := 1, m
 imax := max(@$X_2$[j,1:n])
 if ( imax != -1 ) then
    $X_4$[j] := $X_2$[j,imax]
 else
    $X_4$[j] := $X_0$[j]
 endif
enddo
```

Rasterization Example

(Array SSA Renaming performed on display buffer)



Processor 1

(N/4 polygons)

$@B_1$ @ array

B_1 Local display buffer

Processor 2

(N/4 polygons)

$@B_2$ @ array

B_2 Local display buffer

Processor 3

(N/4 polygons)

$@B_3$ @ array

B_3 Local display buffer

Processor 4

(N/4 polygons)

$@B_4$ @ array

B_4 Local display buffer

Global display buffer

$$B_5 := F\Big(B_1, @B_1, B_2, @B_2, B_3, @B_3, B_4, @B_4 \Big)$$

Rasterization Example

Time in Seconds

No. of Polygons	Serial	Parallel P = 1	Parallel P = 4	Speedup
10,000	3.6	3.8	1.4	2.6
50,000	17.1	17.4	4.8	3.6
100,000	34.5	34.0	9.1	3.8

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

Region Parallelism

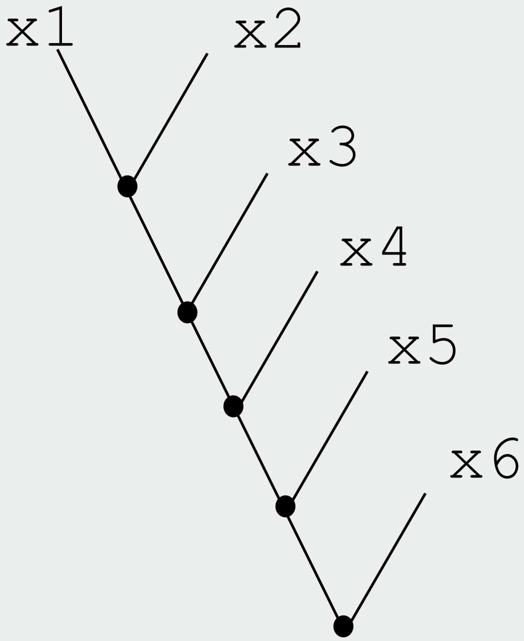
```
do i = …
   x(…) =
enddo

do i = …
   if … then
     x(i) = t + …
     t = x(i)
   endif
enddo
```

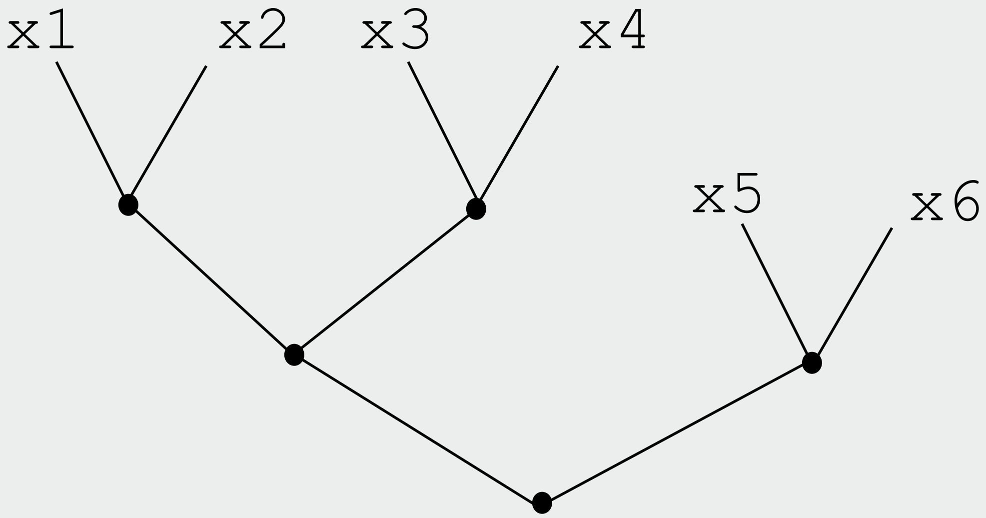
Region Parallelism

```
do i = …
   x1(…) =
   @x1(…) = i
enddo
do i = …
   if … then
      x2(i) = t + …
      t = x2(i)
      @x2(i) = i
   endif
enddo
x3 = ϕ(x2,@x2,x1,@x1,x0)
```

```
x3(j) =
   if (@x2(j) <> ⊥) then
         x2(j)
   elseif (@x1(j) <> ⊥) then
         x1(j)
   else
         x0(j)
   endif
```

Many Possible Factorings





ϕ 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 acceses to field x as accesses to a 1-D Heap Array

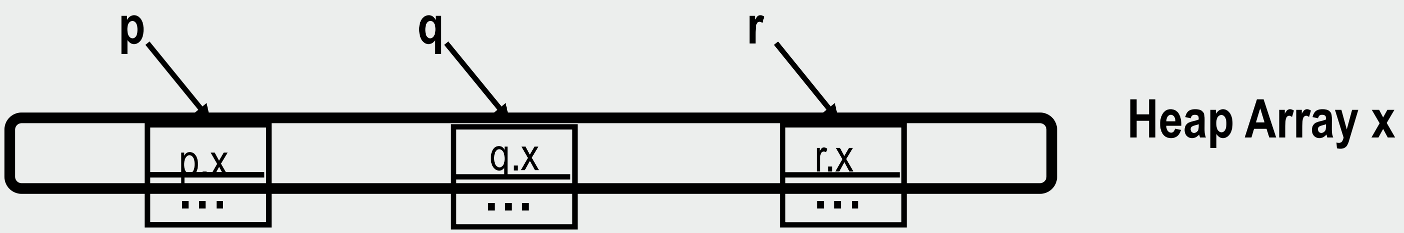
- GETFIELD p.x -> read of x[p]
- PUTFIELD q.x -> write of x[q]

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
```

```
class Z { int x; };
...
$a_9$ = new Z( )
$x_1[a_9]$ = 0
if (...) {
    $x_2$ [$a_9$]= 1
    $x_3$ = $d\phi$ ($x_1$,$x_2$)
} else {
    $x_4$ [$a_9$]= 2
    $x_5$ = $d\phi$ ($x_1$,$x_4$)
 }
$x_6$ = $\phi$ ($x_3$,$x_5$)
y = $x_6$ [$a_9$]
```

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

```
p  := new Z
q  := new Z
r  := p
 . . .
p.x  := ...
q.x := ...
 ...  := r.x
```

Transformed Program

```
p  := new Z
q  := new Z
r  := p
 . . .
T1  := ...
p.x  := T1
q.x := ...
 ...  := T1
```

▪Index Propagation Example

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

Extended Array SSA representation

p := new Z

q := new Z

r := p

$Z.x_1$ [p] := …

$Z.x_2$ = dϕ ($Z.x_0$,$Z.x_1$)

$Z.x_3$ [q] := …

$Z.x_4$ = dϕ ($Z.x_2$,$Z.x_3$)

… = **$Z.x_4$** [r]

$Z.x_5$ = uϕ ($Z.x_3$,$Z.x_4$)

Dataflow Solution

DD (p,q) = true

DS (p,r) = true

L($Z.x_0$) = { }

L($Z.x_1$) = { V(p) }

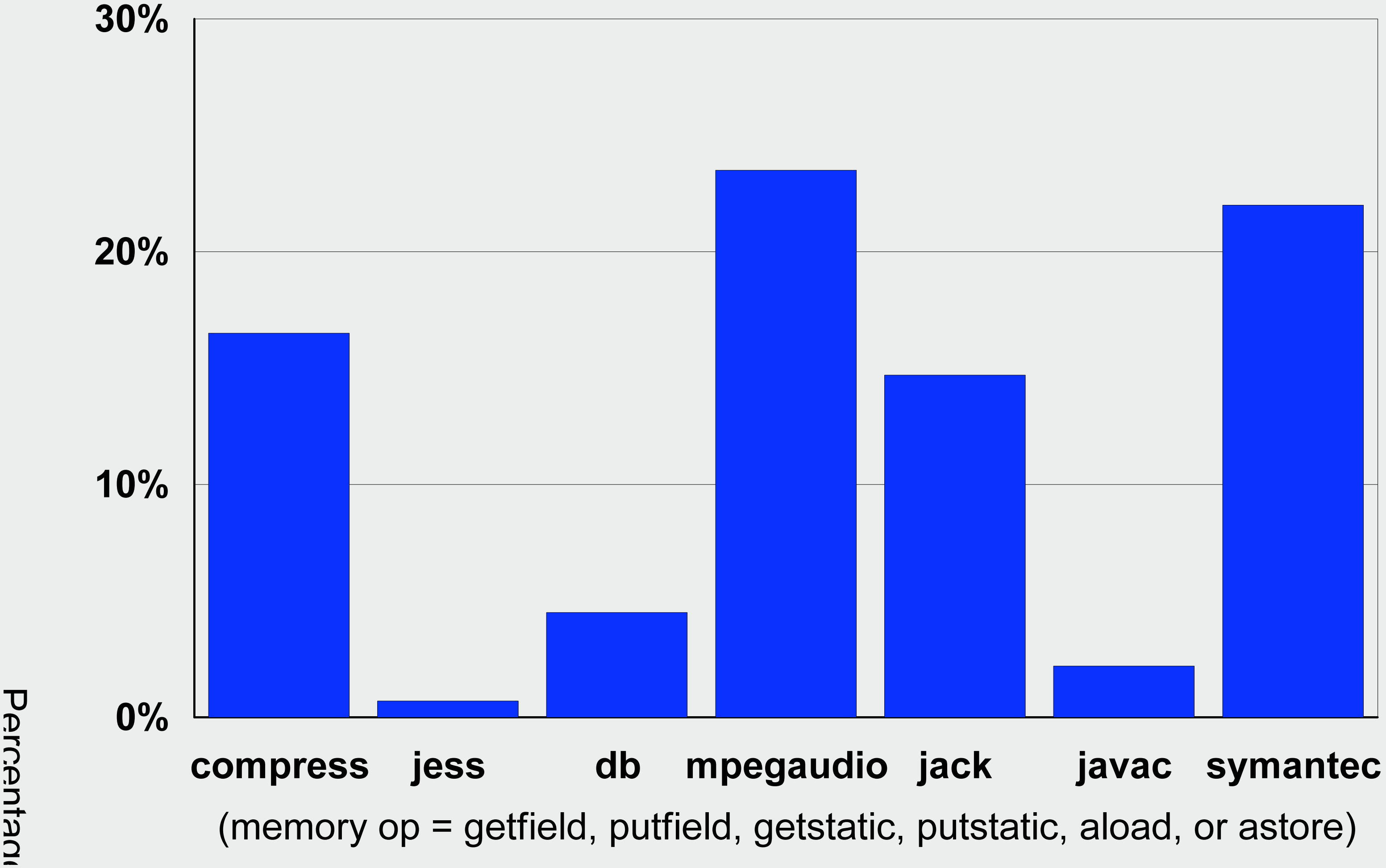
L($Z.x_2$) = { V(p) }

L($Z.x_3$) = { V(q) }

L($Z.x_4$) = { V(p), V(q) }

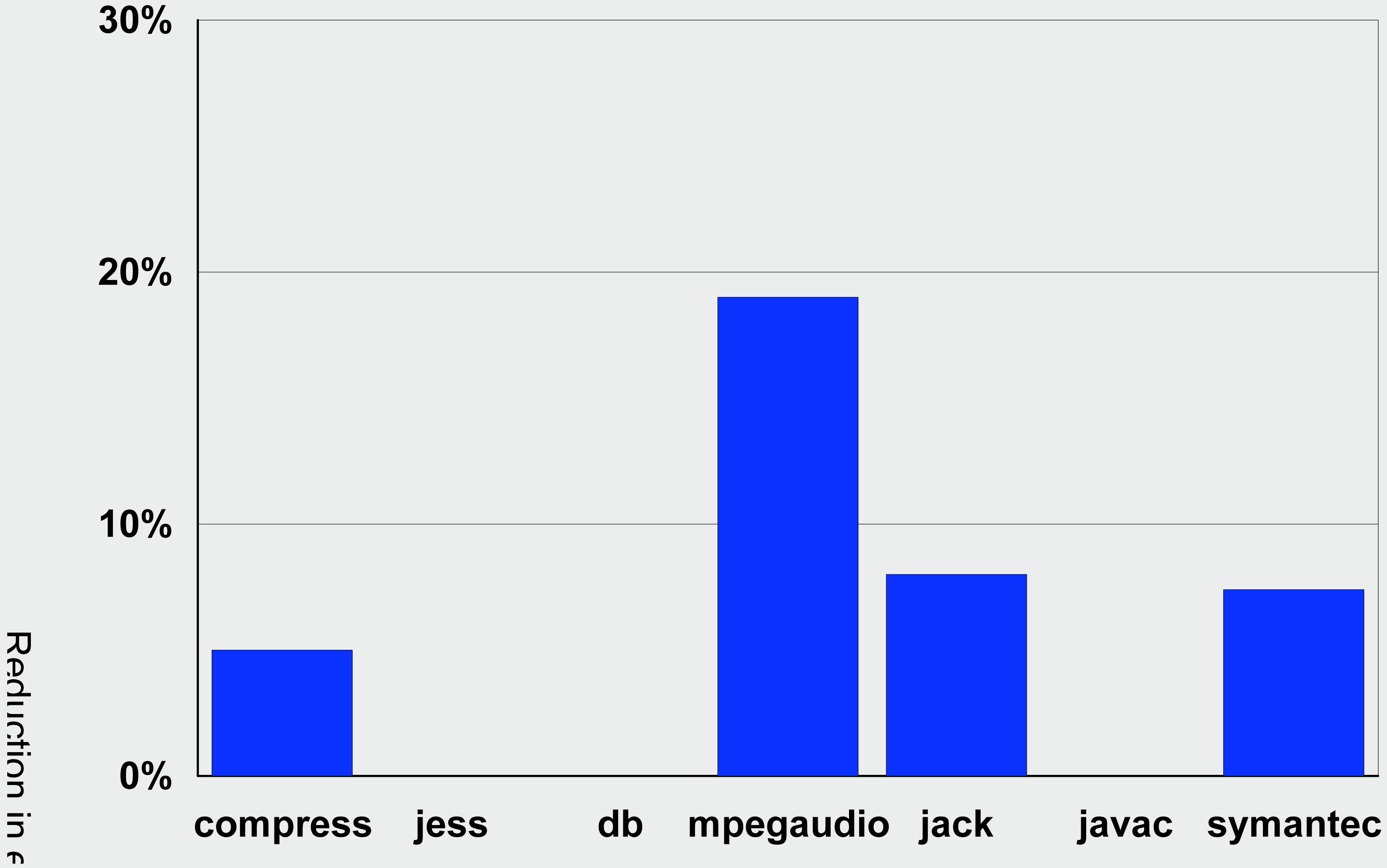
L($Z.x_5$) = { V(p), V(q) }

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 Φ functions and @ variables
- Parallelization with Speculative Execution
- Parallelism across Regions
- Modeling Structures as Arrays

Other SSA-related work that I’ve been involved in

- ABCD: Eliminating Array Bounds Checks on Demand. R.Bodik, R.Gupta, V.Sarkar. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI), June 2000.
- Incremental Computation of Static Single Assignment Form. Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Proceedings of the 1996 International Conference on Compiler Construction, Linkoping, Sweden, April 1996.
- ASTI optimizer, 1991 – 1993. IBM’s first optimizer to use SSA form in a product (XL Fortran 4.1, shipped in 1996).
- Compact Representations for Control Dependence. Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Proceedings of the ACM SIGPLAN ’90 Conference on Programming Language Design and Implementation, White Plains, New York, pages 337-351, June 1990.

Conclusions

Array SSA form is an intermediate form that integrates

- Control flow analysis
- Index analysis
- Renaming

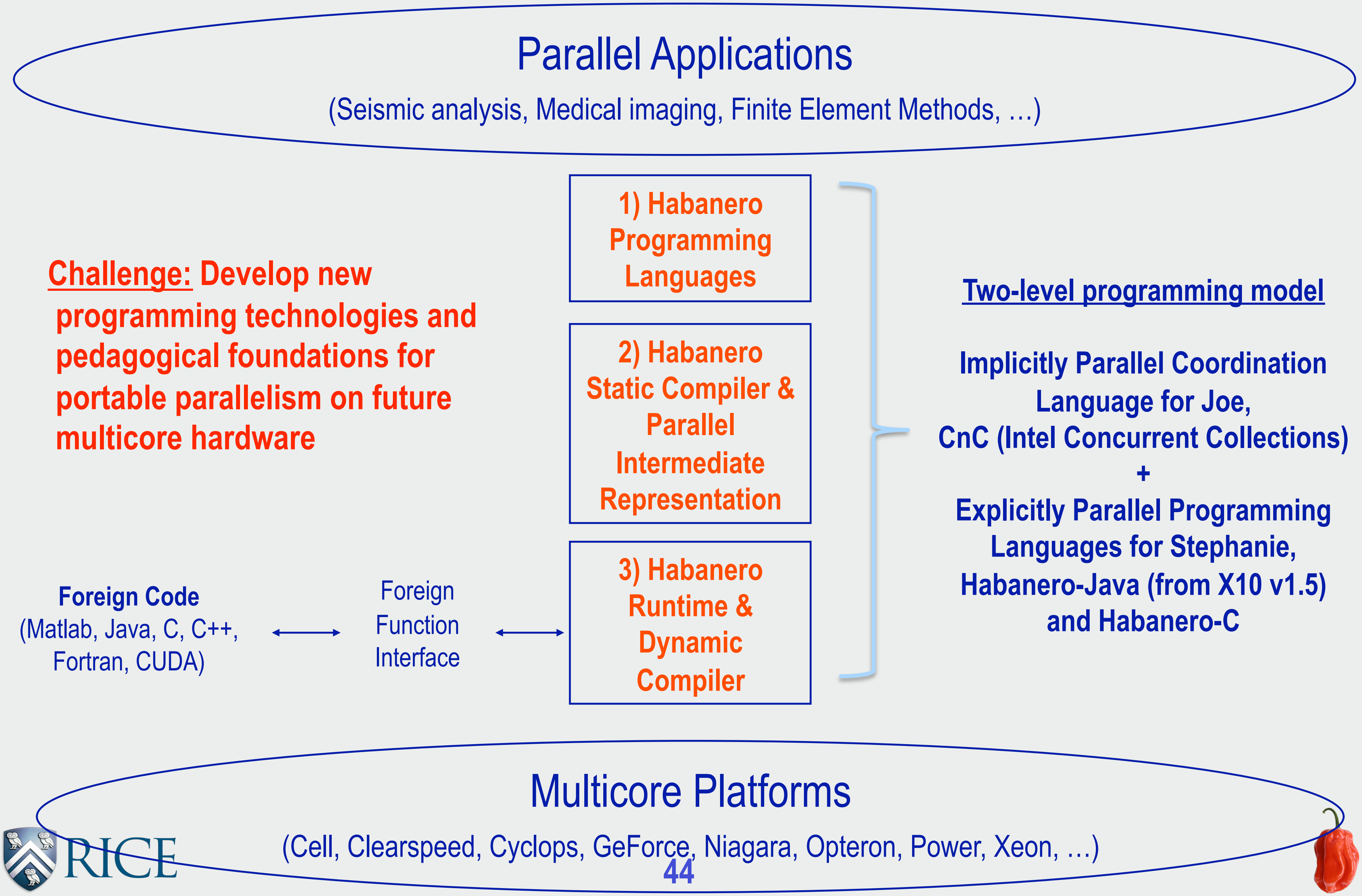
- *Increases reordering potential*
- *Enables speculative execution*
- *Enables element-level optimizations*

RICE

Future Work

- Study of legal ϕ 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

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

Foreign Function Interface

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

Multicore Platforms

(Cell, Clearspeed, Cyclops, GeForce, Niagara, Opteron, Power, Xeon, …)

Habanero Static Parallelizing & Optimizing Compiler

Habanero Languages

Front End

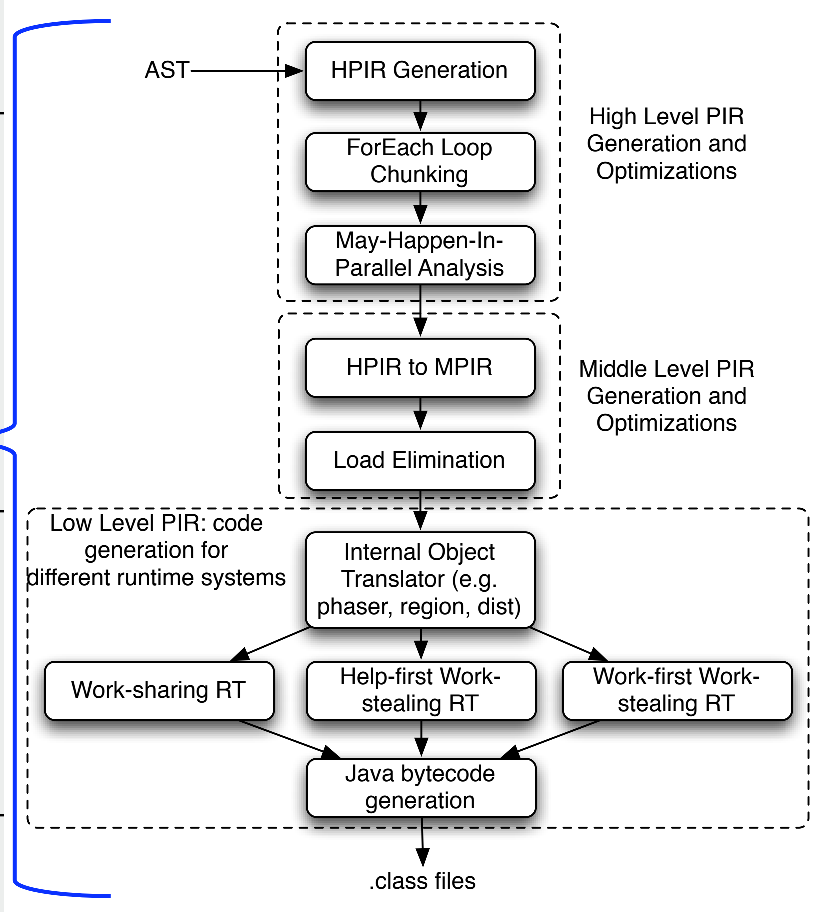
Foreign Function Interface

Sequential C, Fortran, Java, …

Interprocedural Analysis

PIR Analysis & Optimization

Classfile Transformations



ntation (PIR)

ran

regions

for targeting accelerators

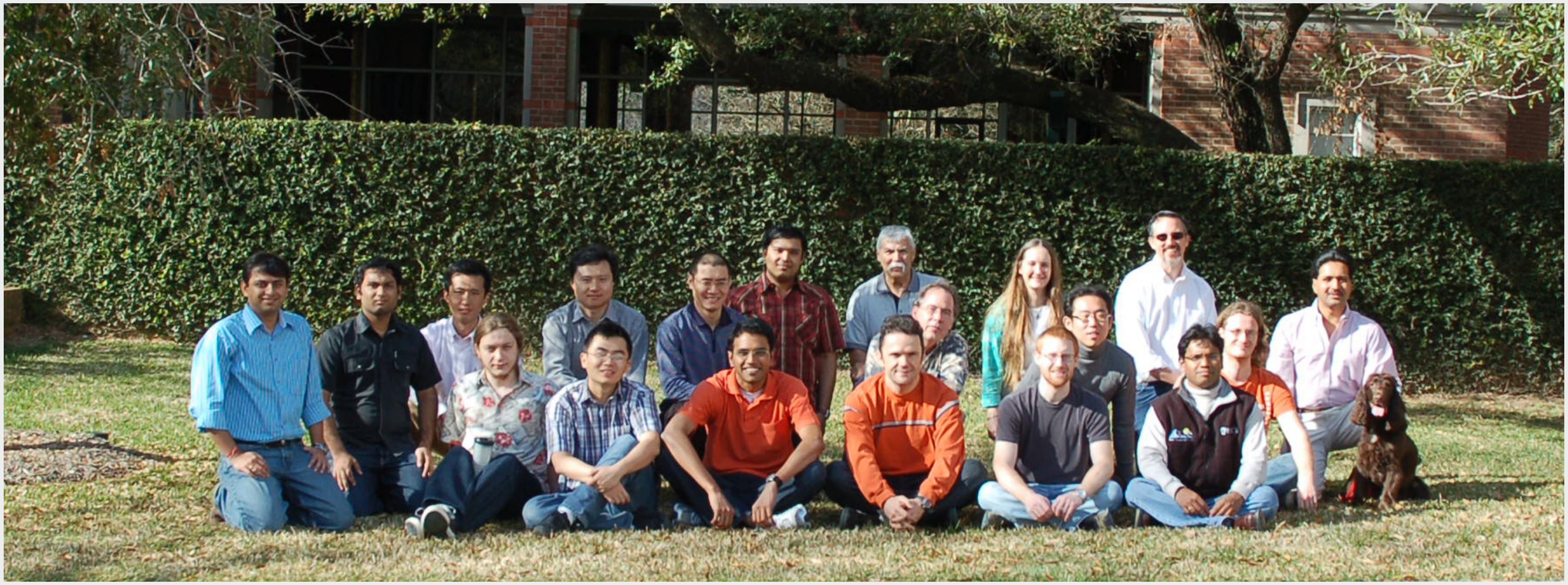
& high-end computing)

Partitioned Code

Portable Managed Runtime

Platform-specific static compiler

Habanero Team Pictures



Platform Aware Compilation Environment project (PACE), April 2009 – October 2013

04/08/2009

DARPA awards $16 million to Rice University to improve compilers

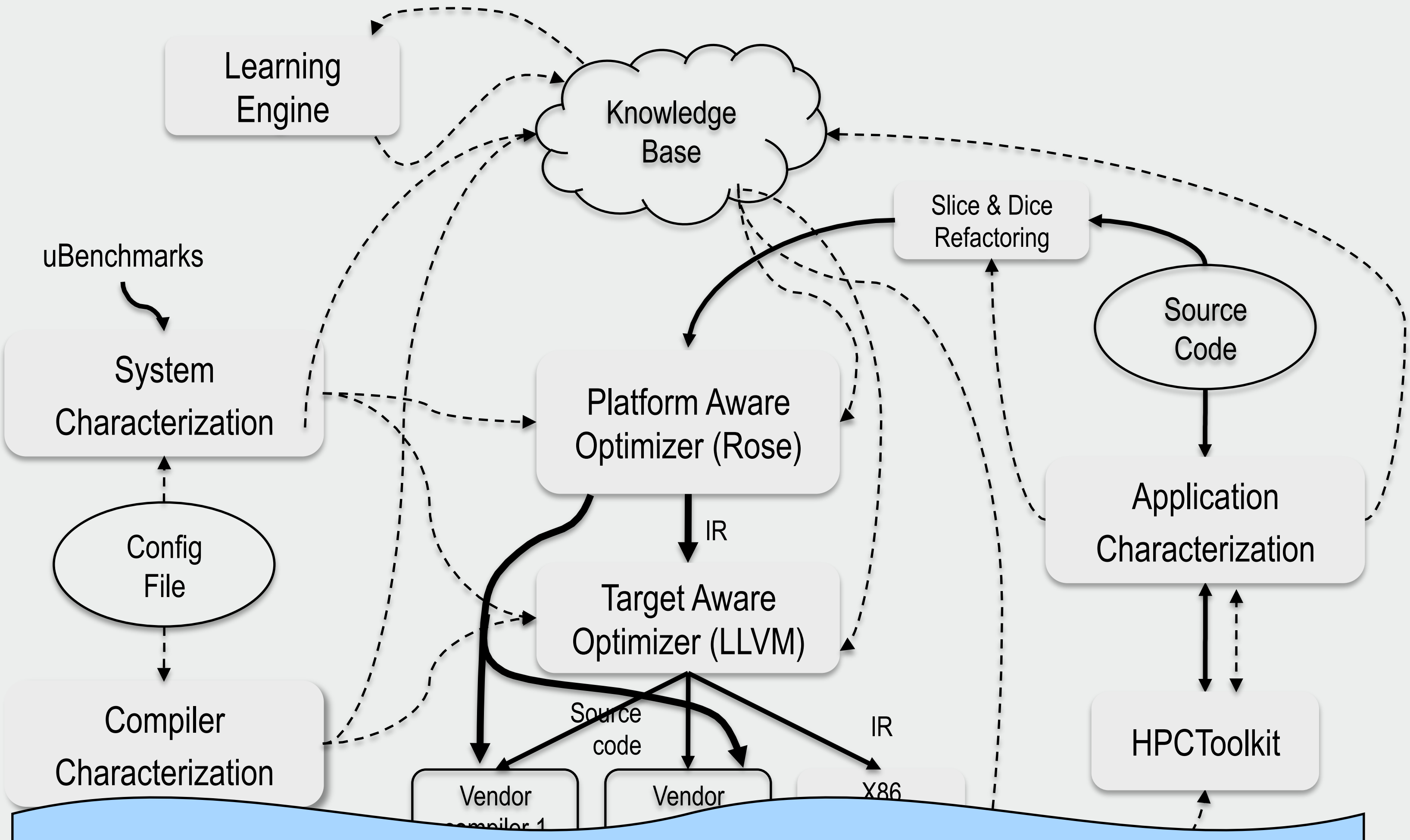
The Defense Advanced Research Projects Agency (DARPA), as part of its Architecture Aware Compiler Environment Program, has awarded Rice University $16 million to develop a new set of tools that can improve the performance of virtually any application running on any microprocessor. …

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

PACE System – the Big Picture



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!