# Increasing the Scope and Resolution of Interprocedural Static Single Assignment

Silvian Calman Jianwen Zhu

University of Toronto, Toronto, Canada

November 2, 2009

www.eecg.toronto.edu/~calman

# Interprocedural SSA (ISSA)

- SSA replaces the uses of scalar stack variables with a single definition
- SSA is widely used in compilers
  - Constant propagation, induction variable identification, etc.
- ISSA expands scope to include globals, singleton heap variables, and record elements
  - Definitions can be used in other procedures

# Interprocedural SSA

- Two additional challenges
  - Merge points due to pointer dereferences
  - Passing values across call sites
- Intermediate Representation Extensions
  - $p.\phi^{S}(var,curr,new)$
  - $p.\phi^{L}(<var_1,val_1>, ...,<var_n,val_n>)$
  - $\phi^{V}{}_{<var,p>}(<ci_1,val_1>, ...)$
  - $p\ .\phi^{C}{}_{<var,ci>}(<func_1,val_1>, ...)$

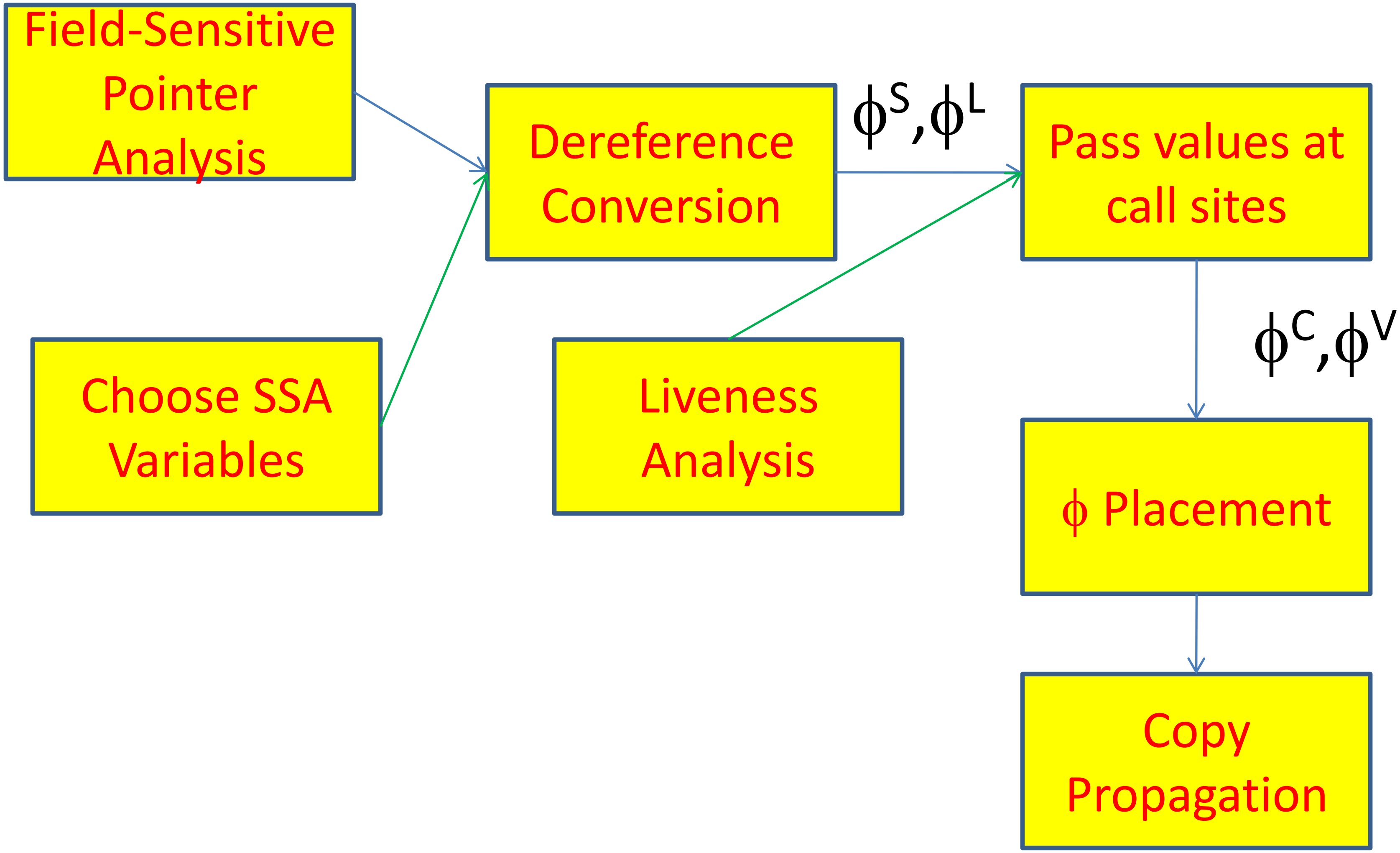
# Example

```
int   y = 5, z = 10,
*x, **g;
C( ) { print( **g ); }
B( ) { *g = &z; }
main( ) {
  g = &x;
  x = &y;
S1:  B( );
  **g = 20;
S2:  C( );
   }
```

```
C( ) {
   x2 = $\phi^{V}_{<x,C>}(CI_2 ,x1)$;
   y2 = $\phi^{V}_{<y,C>}(CI_2 ,y1)$;
   z2 = $\phi^{V}_{<z,C>}(CI_2 ,z1)$;
   print(x2.$\phi^{L}$(<&y, y2>,
                     <&z, z2>) );
}
B( ) {}
main( ) {
CI$_1$:  B( );
  x1 = $\phi^{C}_{<x,CI1>}(B,\&z)$;
  y1 = x1.$\phi^{S}$(&y,5,20);
  z1 = x1.$\phi^{S}$(&z,10,20);
CI$_2$:  C( );
 }
```

```
int   y = 5, z = 10,
     *x, **g;
C( ) {
  print( 20 );
  }
main( ) {
  C( );
  }
```

# ISSA Generation



Field-Sensitive Pointer Analysis

Choose SSA Variables

Dereference Conversion

$\phi^S, \phi^L$

Liveness Analysis

Pass values at call sites

$\phi^C, \phi^V$

$\phi$ Placement

Copy Propagation

# Presentation Overview

- ISSA Generation
  - Copy propagation
  - Liveness analysis
  - Pointer analysis
  - Constant propagation
- Conclusions

# Experimental Setup

- Setup
  - Dual Core 1.66 GHz, L1 32 KB/ L2 2 MB Cache
  - Ubuntu (64-bit OS)
  - 4 GB Memory
- Benchmarks
  - MediaBench
  - SPEC2K with exception of gcc and vortex
- LLVM
  - Passes applied: SSA, Constant propagation (interprocedural), aggressive dead code removal, instruction combining
  - Various analyses (dominator tree, call graph, etc.)

# Copy Propagation

- Replaces target of an assignment with value at usage points
- Fold $\phi^S$, $\phi^L$, $\phi^V$, $\phi^C$, and $\phi$ instructions
- Copy propagation helps by:
  - Reducing IR size
  - Correlating definitions with uses while removing false merge points
- How do we interpret an interprocedural value?

# Interprocedural Value

- Definition for instruction *I* in procedure *P*
  - Value of *I* in the last call frame of *P* on the stack, or otherwise (*P* is not on the stack) value of *I* in the last invocation of *P*
- Benefits
  - Value in SSA equals a value in ISSA
    - Can directly construct ISSA on IR in SSA form
  - Folding of $\phi^V$ instructions is trivial
    - Folded whenever it merges the same value

# Problem of Propagating through $\phi^C$

- Folding $\phi^C$ instructions not as simple as $\phi^V$ instructions
  - Some $\phi^C$ instructions maintain values of overwritten expressions
  - In traditional SSA form
    - $\phi$ instructions are inserted at entries to cycles
    - Merge different values
  - Different
    - $\phi^C$ instruction can merge the same value but we might NOT be allowed to fold them
    - Depends on usage point
  - Violate our definition for interprocedural value

# Problem of Propagating through $\phi^C$

```
int Sum(int a,int b, int c)  {
   tmp=a+b+c; return tmp;
}
void main( ) {
   int e = Sum(1,2,3);
   int f = Sum(20,21,315);
   printf(f,e);
}
```

$e=\phi^C$ of first call. value=tmp=a+b+c where a=1,b=2,c=3

$f=\phi^C$ of second call. value=tmp=a+b+c where a=20,b=21,c=315

```
StructPtr recursiveProc(StructPtr a, StructPtr b) {

  resA = recursiveProc(a->right,b->right);

  resB = recursiveProc(a->left,b->left);
  if (resA==resB)  {
       ....
     }
  }
```

Pointer produced a few invocations ago

Pointer produced in the last invocation

If indiscriminate propagation of $\phi^C$, we come to wrong conclusion that branch is always taken. Can't sub any $\phi^C$ as it violates our definition.

# Propagating through $\phi^C$

- Can do so when $\phi^C$ merges a constant
- Can do so when $\phi^C$ merges the same instruction *V* in procedure *P*
  - *P* and current procedure not in the same maximal Strongly Connected Component
  - At usage point, $\phi^C$ corresponds to last invocation of *P*
- Copy propagation
  - Reduced $\phi^C$ and $\phi^V$ instructions by 44.5%
  - Folded 30% of $\phi^V$ instructions with multiple operands

# Liveness Analysis

- Pruned SSA does not insert $\phi$ instructions that will not be used
  - Using liveness analysis to determine where a $\phi$ is redundant
- Our liveness analysis constrains insertion of $\phi^C$ and $\phi^V$ instructions
  - Insert $\phi^V$ instructions only for variables that may be written prior to some invocation of a procedure
  - Insert $\phi^C$ instructions only for variables that may be read after some invocation of the target procedure
  - We apply these conditions by :
    - Identifying the set of variables that may be written before a procedure
    - Identifying the set of variables that may be read after a procedure
- Reduced $\phi^C$ and $\phi^V$ instructions by 23.3%

# Pointer Analysis

- Evaluated the impact of the pointer analysis on the input and output sets (number of variables) that must be propagated at call sites
- Comparison between
  - Field-Sensitive and Field-Insensitive pointer analysis (LLVM infrastructure)
- Showed the Field-Sensitive pointer analysis reduces number of variables propagated across call sites by a factor of 12.1

# Constant Propagation

- Applied constant propagation on the ISSA form
  - Sparse Conditional Constant Propagation
  - Constant folding and branch resolution
- Demonstrate benefit
  - Folded an additional 11.8% instructions over the LLVM infrastructure
  - Removed 5.6 additional basic blocks as a result

# Constant Propagation Extension

- Context-Sensitive
  - Keep track of context-specific values in a map
  - Restricted to one-level context-sensitivity
- Identify and apply preconditions
  - Conditions that must be true when reaching a program point
  - Restricted to indirect call sites
- Runtime of algorithm was in milliseconds
- Constant folded an additional 15.3% of instructions over  the LLVM infrastructure

# Comparison with other ISSA

- Greater scope
  - SSA variables consist of singular heap locations and elements of records
- Interprocedural value
- Higher resolution since each SSA variable corresponds to one memory location
  - No may-def/use relation
  - We replace 1.7 times more load instructions with the corresponding definition

# Related Work

- Cytron and Gershbein. Efficient accommodation of may-alias information in SSA form. (PLDI’93)
- Liao, S.W. SUIF Explorer: An interactive and interprocedural parallelizer. (Ph.D. theis)
- Staiger et al. Interprocedural Static Single Assignment Form. (WCRE’07)
  - Compared ISSA construction using Steensgaard’s and Andersen’s pointer analysis
  - ISSA form stored in separate data structure
  - May-def/use relations

# Conclusions

- Proposed ISSA construction, which includes
  - Copy propagation, liveness analysis, handling singleton heap variables
  - Showed benefit to constant propagation
- MediaBench benchmarks performed better
  - Folded $\phi$ instructions and runtime
  - No recursion or complex abstract data structures