Code analysis *and* transformation

IPA example

Simone Campanoni
simone.campanoni@northwestern.edu

Research paper

Title: Practical and Accurate Low-Level Pointer Analysis

VLLPA

Authors:
Bolei Guo Matthew J. Bridges Spyridon Triantafyllis Guilherme Ottoni Easwaran Raman David I. August

CGO, 2005

The two problems for CATs

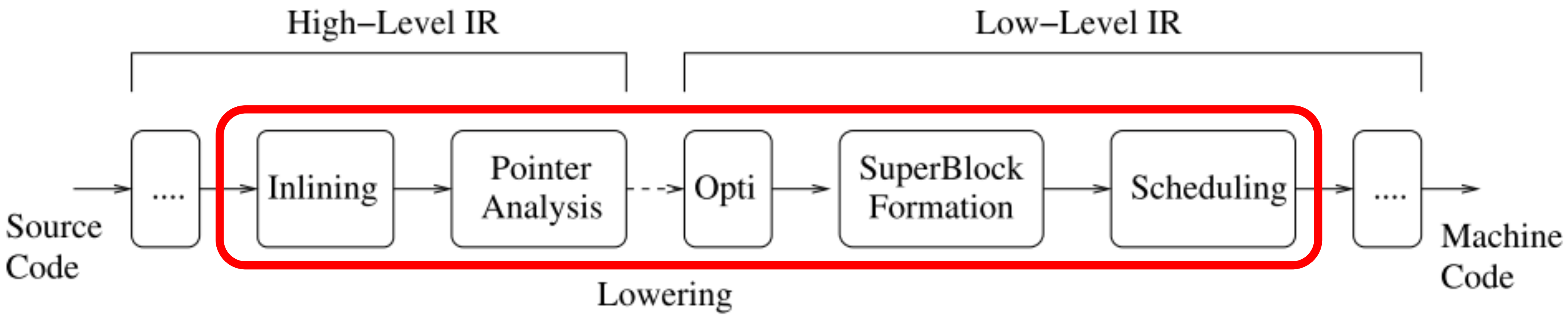
- Problems:
  1. Identifying memory aliases
  2. Identifying callees of indirect calls

- Solutions:
  - Solve conservatively 1 first, and then 2
  - Solve 1 and 2 at the same time

    VLLPA

Alias analysis for C programs

- Usually run once at the source level (the DDG is also computed)



- Compilation passes modify the IR, so they must update the DDG
  - Add complexity to each pass
  - Updates are conservative

Alias analysis for C programs

```
   char A[10],B[10],C[10];
   foo() {
     int i;
     char *p;

     for (i=0;i<10;i++) {
       if (...)
1:       p = A;
       else
2:       p = B;
3:     C[i] = p[i];
4:     A[i] = ...;
     }
   }
```

(a) Source code

Instructions 3 and 4 may access the same memory location

VLLPA:
a low level pointer analysis for C programs

- This paper proposes an alias analysis at the IR level
  - It can be run multiple times
  - No conservative updates
  - Passes are simpler
  - No data type information (not very useful for C anyway)

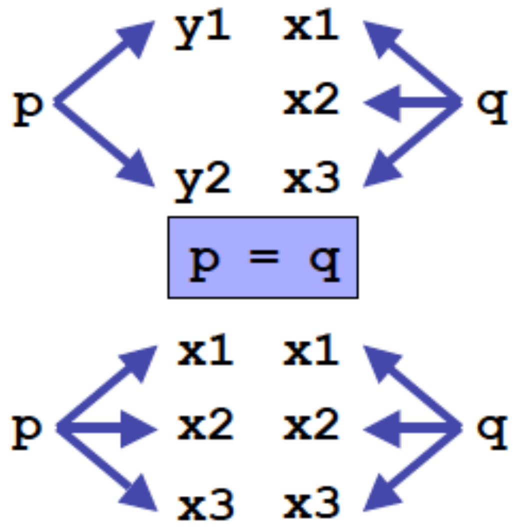
- The first
  context-sensitive and partially flow-sensitive
  low-level points-to analysis algorithm

VLLPA sequence

Pointer points-to sets

i: p = q

- GEN[i] = { }   KILL[i] = { }
  OUT[i] = {(p, z) | (q, z) ∈ IN[i]} U (IN[i] – {(p,x) for all x})



VLLPA sequence

Outline

- Abstractions used
- Data-flow intra-procedural analysis
- Inter-procedural analysis
- Evaluation

Memory abstraction

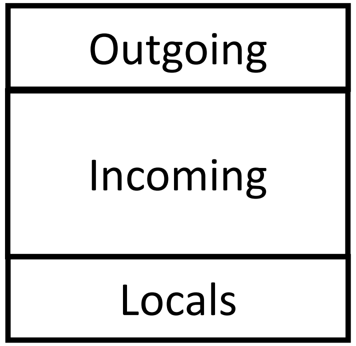
- **Abstract address** = memory location at analysis time =
- **Abstract structure** = contiguous set of abstract addresses

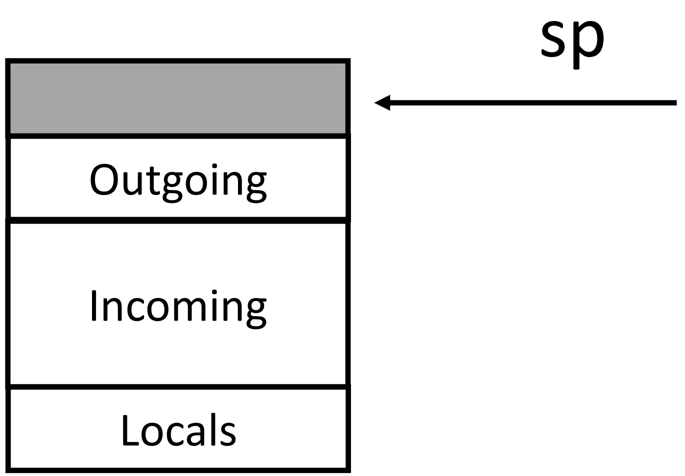
- Memory is divided into a set of abstract structures, each with a unique name
  - A single abstract structure can correspond to multiple blocks at runtime
  - Unbounded set of memory blocks -> finite set of abstract names

- An abstract structure is created for each global variable

Activation frame

```
int myF (int arg0, int arg1){
    int v1, v2, v3;
    ...
    int *p = &v1;
    ...
    ... = *p
    ...
    return v1+v2+v3;
}
```





Memory abstraction

- Activation frame:
  - One abstract structure for each
    - Element in the incoming parameter space
    - Element in the outgoing parameter space
    - Variable in the local variable space

- Heap object allocated:
  - Named according to the context (2 call stack depth)

Abstract structures

- <S,o>
  - S is a structure name
  - o is an offset

```
typedef struct {
  int64_t f1;
  int64_t f2;
} myT;
void myF (void){
  myT *p = (myT *)malloc(sizeof(myT));
  int *q = &(p->f2);
  …
}
```

```
void myF (void){
  p = call malloc(16)
  q = p + 8
  …
}
```

What is the abstract address pointed by p?

<p,0>

What is the abstract address pointed by q?

<p,8>

Abstract structures

- <S,o>
    - S is a structure name
    - o is an offset

- VLLPA merges all array elements
    - myArray[5] is the same location of myArray[42]
    - Conservative assumption
        - More aliases
        - Much faster analysis

Abstract structures, pointer aliases, and dependencies

- Two pointers alias if
  there is an abstract address that they can both point to

- There is a dependence between two instructions if
  the pointers used by them alias

Abstract structures

- <S,o>
  - S is a structure name
  - o is an offset

```
typedef struct {
  int64_t f1;
  int64_t f2;
} myT;
void myF (myT *p){
  int *q = &(p->f2);
  …
}
```

```
void myF (void *p){
  q = p + 8
  …
}
```

What is the abstract address pointed by p?
What is the abstract address pointed by q?

Unknown Initial Values (UIVs)

- They encode the “unknown”

- Represent memory blocks accessible by a function,
  but not created by either that function or its callees

- UIVs are created for memory blocks reachable
  (directly or indirectly)
  through parameters or global variables

Unknown Initial Values (UIVs)

- For a parameter A,
  [A] represents the memory block pointed by A

```
void myF (void *P0){
        Var1 = P0
        …
}
```

What is the abstract address pointed by Var1?
<[P0],0>

Unknown Initial Values (UIVs)

- If [A] has a field at offset o, which is a pointer, then the following new UIV is created: [A]@o

```
void myF (void *P0){
        Var1 = P0
        …
        Var2 = Mem[Var1+4]
        …
}
```

Abstract structure pointed by Var1: <[P0],0>

What is the abstract structure pointed by Var2?
<[P0]@4,0>

- UIVs are created lazily

Outline

- Abstractions used
- Data-flow intra-procedural analysis
- Inter-procedural analysis
- Evaluation

Main challenge

- Common memory operations (array and field accesses) are not explicit in the code

```
Vx = Vy + 10
Vz = Mem[Vx]
```

```
my_struct_t *Vy = ...
int64_t Vz = Vy->myField;
```

- The analysis has to infer whether a memory operation “looks like” a field and/or array access

Intra-procedural analysis

- Assume SSA
  - One assignment per variable. Therefore
  - For each variable, we need to maintain a single points-to set

  R(var) = mapping from a variable to a set of abstract addresses that might point to

```
void myF (void){
    int v1, v2;
    int *p, *q;
    int *p = &v1;
    int *q = &v2;
    if (rand()) p = q;
}
```

R(v1) = { }
R(v2) = { }
R(p) = {v1,v2 }
R(q) = {v2 }

Intra-procedural analysis

- Assume SSA
  - One assignment per variable. Therefore
  - For each variable, we need to maintain a single points-to set

  R(var) = mapping from a variable to a set of abstract addresses
  that might point to

- Not flow-sensitive for pointers in memory
  - Single points-to set for each abstract memory location

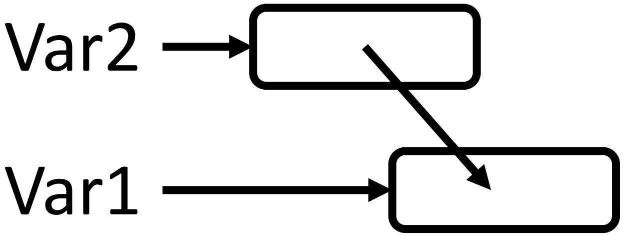
  M(addr) = mapping from an abstract address to a set of abstract addresses
  that might point to

- UIVs of the function analyzed
  - **I(f)** = set of UIVs of function f

Intra-procedural analysis

- Modify R, M, and I with a data-flow analysis
- Var1 = Mem[Var2]

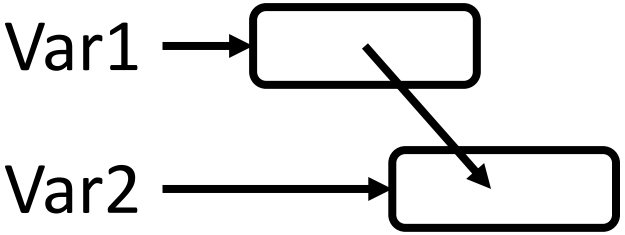
  R(var1) = { M(<S,o>) | <S,o> ∈R(var2) }



- Mem[Var1] = Var2

  For each <S,o> ∈ R(Var1):

    M(<S,o>) ∪= R(Var2)



- Var1 = Var2 + c

  R(Var1) = { <S,o+c> | <S,o> ∈ R(Var2)}

Intra-procedural analysis

- Var1 = Var2 + Var3

  R(Var1) = { <S,o+c> | <S,o> ∈ R(Var2) and c = infer_offset(Var3)} ∪
  { <S,o+c> | <S,o> ∈ R(Var3) and c = infer_offset(Var2)}

  Offset assumed to follow i x l + c
  l is the size of array elements
  c is a constant displacement
  (non-zero if the array is a structure field)

- VarX = PHI(Var1, Var2, … VarN)
  - R(VarX) = R(Var1) ∪ R(Var2) ∪ … ∪ R(VarN)

Termination

- Data-flow analysis can only add new elements in R, M, and I
  - They increase monotonically
- To ensure termination: we need an upper bound to R, M, and I
  - Finite number of abstract addresses
- Do we have these upper bounds?

Termination: unbounded UIVs?

R(r1) ={<[P0],0>,
<[P0]@4,0>,
<[P0]@4@4,0>

```
typedef struct T {
  int data; T* next;
} T;

f(T* l) {
  while (l != NULL)
    ...
    l = l->next;
}
```

UIV: P0

(a) List: source

```
f:
LOOP:
  r1 = φ (param0, r2)
  br r1 == 0 EXIT
  ...
  r2 = mem[r1+4]
  jump LOOP
EXIT:
```

(b) List: low-level

If <[UIV],c> ∈ R and <[UIV]@N,c> ∈ R , then remove the latter

Termination: what about the offsets?

R(r2) ={ <[P0],0>,
<[P0],4>, <[P0],8>, …

```
int A[100];

g() {
  int *a = A;
  while (...) {
    ... = *a;
    ...
    a++;
  }
}
```

(c) Array: source

```
A:
  reserve 400
g:
  r1 = A
LOOP:
  r2 = φ (r1, r4)
  r3 = mem[r2]
  ...
  r4 = r2 + 4
  br (...)  LOOP
```

(d) Array: low-level

If <S,o1> ∈ R and <S,o2> ∈ R and o1 < o2 then remove <S,o2>

Intra-procedural analysis

- In all equations:
  - If <[UIV],c> ∈ R and
    <[UIV]@N,c> ∈ R then remove the latter

Elements of the same list are represented as a single abstract address

  - If <S,o1> ∈ R and
    <S,o2> ∈ R and
    o1 < o2 then remove <S,o2>

Elements of the same array are represented as a single abstract address

Outline

- Abstractions used
- Data-flow intra-procedural analysis
- Inter-procedural analysis
- Evaluation

Intra-procedural analysis

- Assume SSA
  - One assignment per variable. Therefore
  - For each variable, we need to maintain a single points-to set

  R(var) = mapping from a variable to a set of abstract addresses that might point to

- Not flow-sensitive for pointers in memory
  - Single points-to set for each abstract memory location

  M(addr) = mapping from an abstract address to a set of abstract addresses that might point to

- UIVs of the function analyzed
  - **I(f)** = set of UIVs of function f

Propagated

VLLPA main blocks

- Intra-procedural analysis:
  - Compute R, M, I for every function in isolation
- Inter-procedural analysis:
  - Propagate M, I through the call graph
  - Map abstract addresses to UIVs
  - Update the call graph

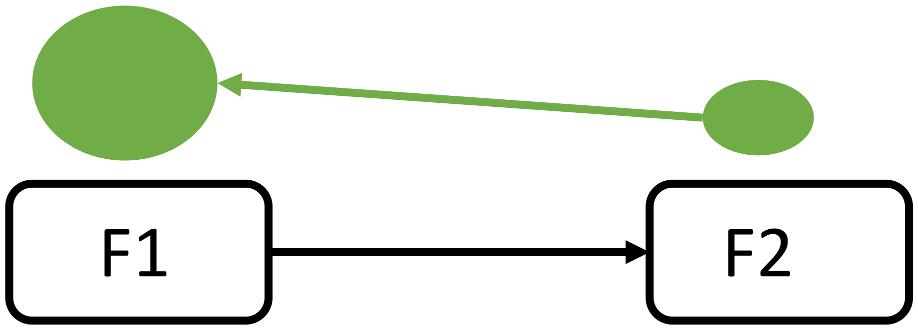
VLLPA summary

- Summary: M, I

M(addr) = mapping from an abstract address to a set of abstract addresses that might point to

I(f) = set of UIVs of function f

- Transfer function

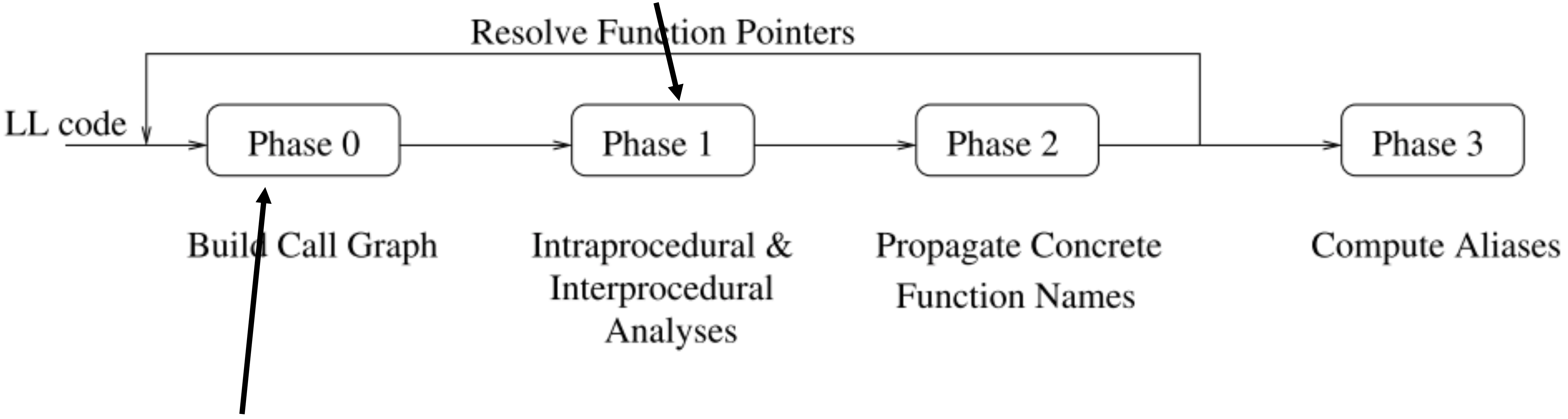


SCCDAG

- We compute SCCs of the call graph
- This SCCDAG is the graph where nodes are either functions or SCCs
- An SCCDAG has no cycles

Algorithm outline

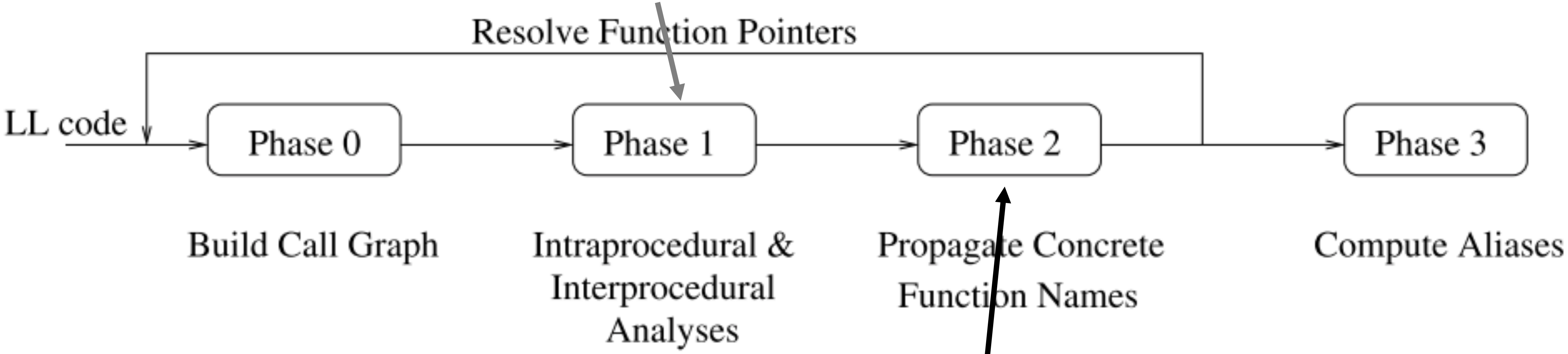
SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed



First iteration: indirect calls have no target
Call graph is augmented with later iterations
SCCDAG is computed from the call graph

Algorithm outline

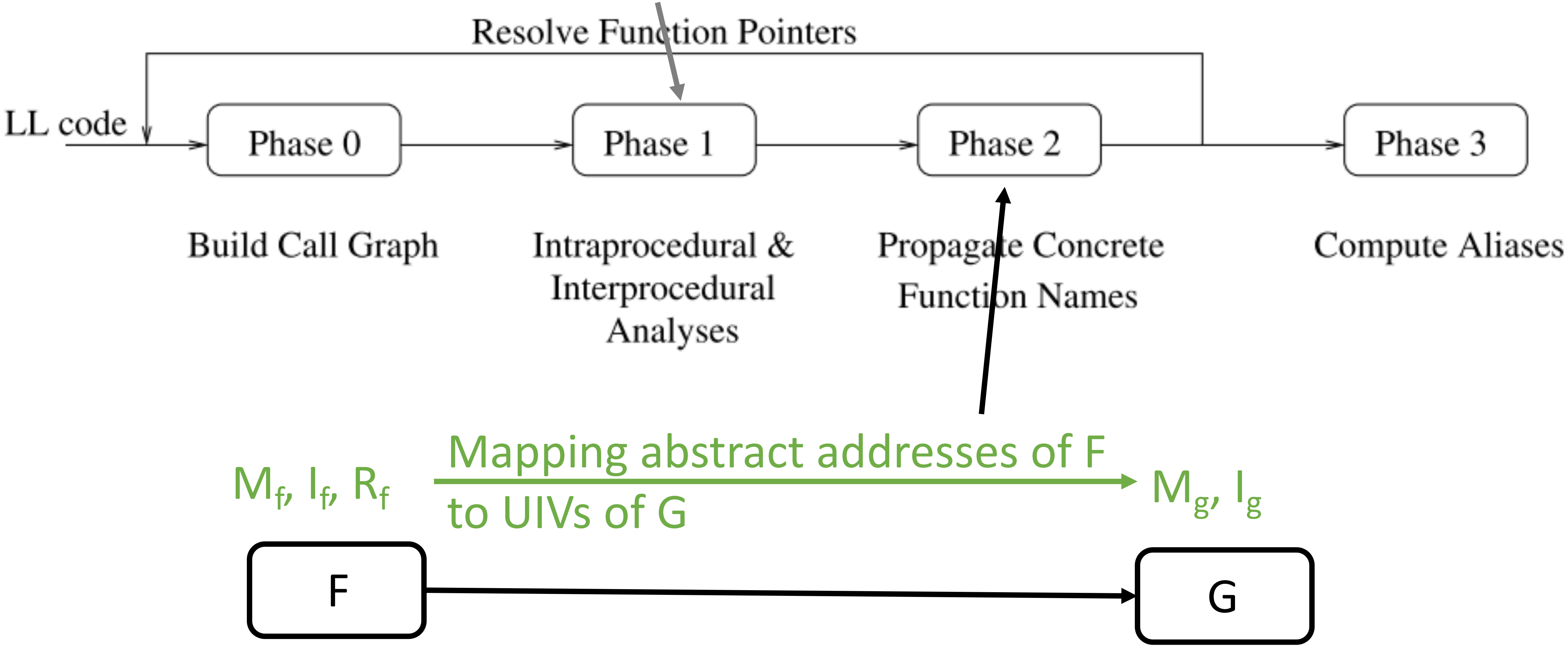
SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed



SCCDAG traversed in topological order to resolve UIVs and indirect calls

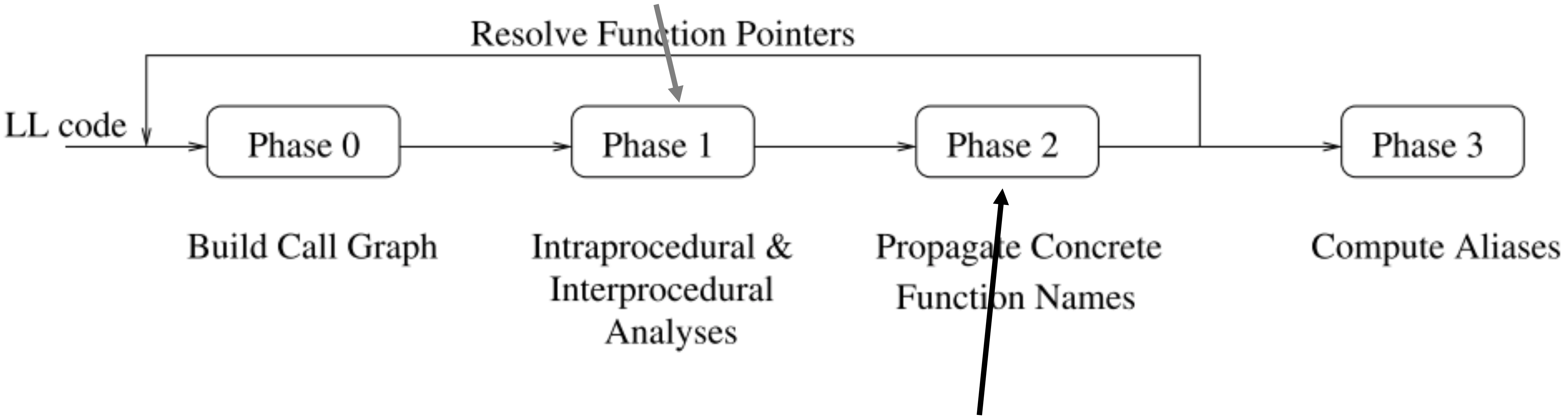
Algorithm outline

SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed



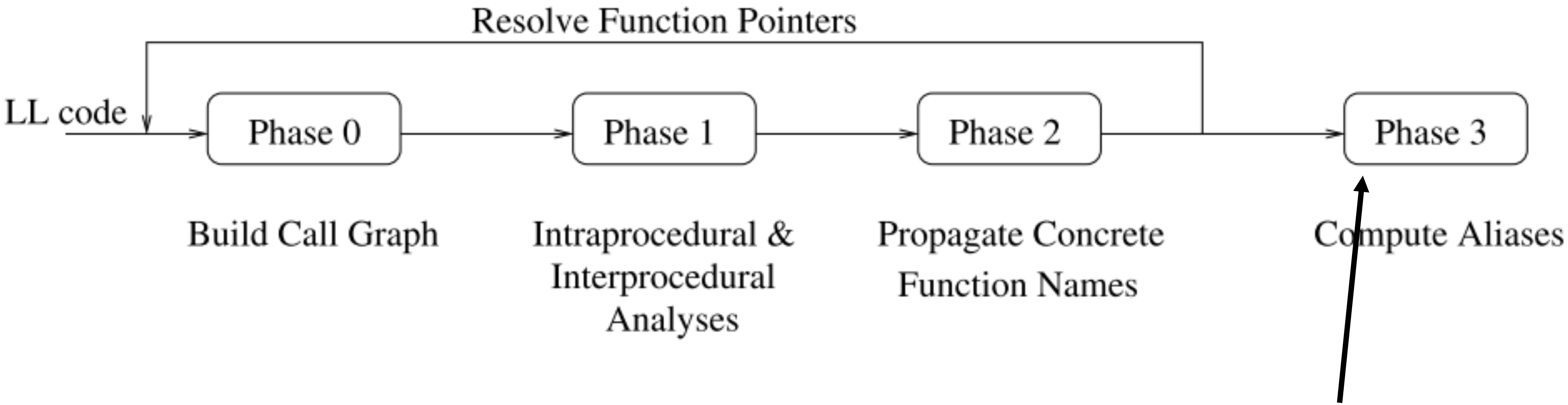
Algorithm outline

SCCDAG is traversed in reverse topological order
Unknown initial values (UIV) assumed



SCCDAG traversed in topological order to resolve UIVs and indirect calls

Algorithm outline



The now complete SCCDAG is traversed once more
in topological order
to compute aliases and dependences

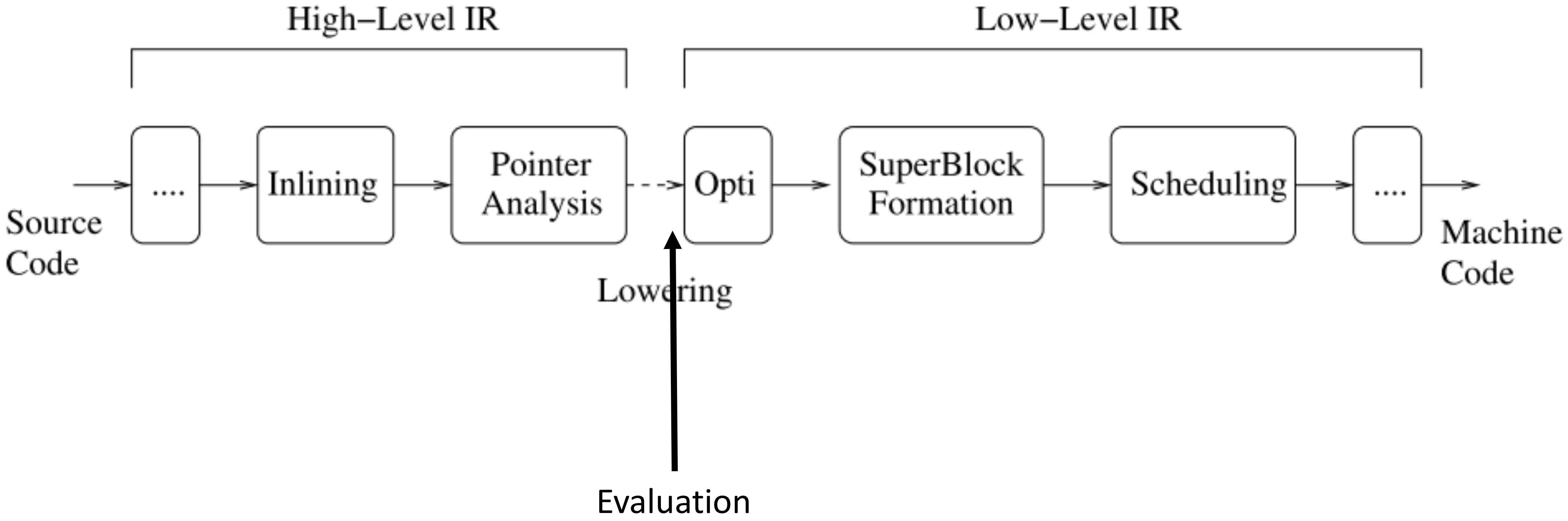
Outline

- Abstractions used
- Data-flow intra-procedural analysis
- Inter-procedural analysis
- Evaluation

VLLPA evaluation

- Comparing against high-level language alias analysis
- Analysis time
- Accuracy of the analysis
- Performance of the generated binary

Evaluation:
Comparing alias analyses



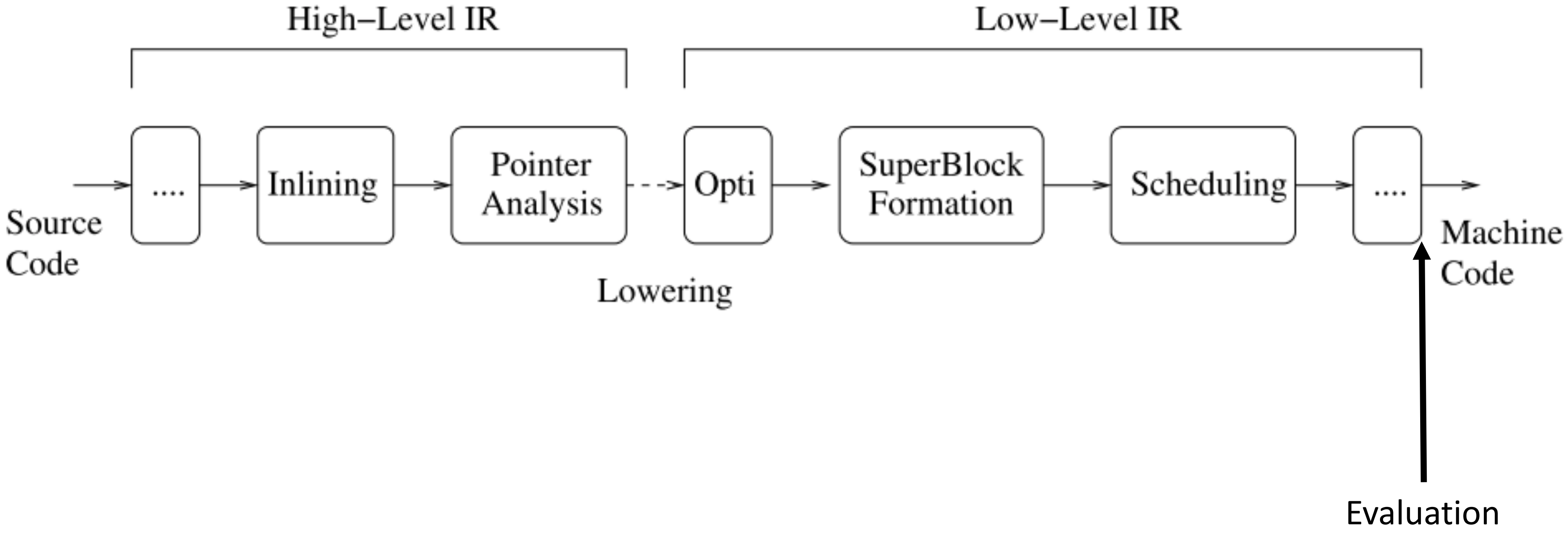
Evaluation: analysis time

Benchmark	# Procs	# Opers	# Indirect Calls	Time (s) VLLPA	Time (s) IMPACT
epicdec	34	3998	0	0.770	0.116
g721dec	26	2396	1	0.035	0.150
g721enc	26	2395	1	0.036	0.091
gsmdec	94	11869	6	0.129	0.645
gsmenc	94	11869	6	0.146	0.472
mpeg2dec	114	10223	0	2.150	0.537
adpcmenc	3	288	0	0.071	0.061
adpcmdec	3	284	0	0.055	0.030
rasta	436	42500	7	3.880	2.428
099.go	372	55879	0	2.087	1.765
124.m88ksim	239	26663	3	4.584	1.357
129.compress	18	1211	0	0.268	0.0759
130.li	357	11953	4	14.843	73.340
132.ijpeg	473	33780	644	2.484	13.899
164.gzip	62	7346	2	0.764	0.339
175.vpr	255	25111	0	1.328	1.743
176.gcc	2220	463462	197	1495.318	1706.950
181.mcf	24	2157	0	0.285	0.1383
186.crafty	110	41370	0	1.543	0.694
197.parser	324	22686	0	2.835	3.388
254.gap	854	145017	1281	643.734	950.64
255.vortex	923	91864	15	12.107	42.330
256.bzip2	63	6725	0	0.485	0.2746
300.twolf	167	53950	0	1.567	1.136

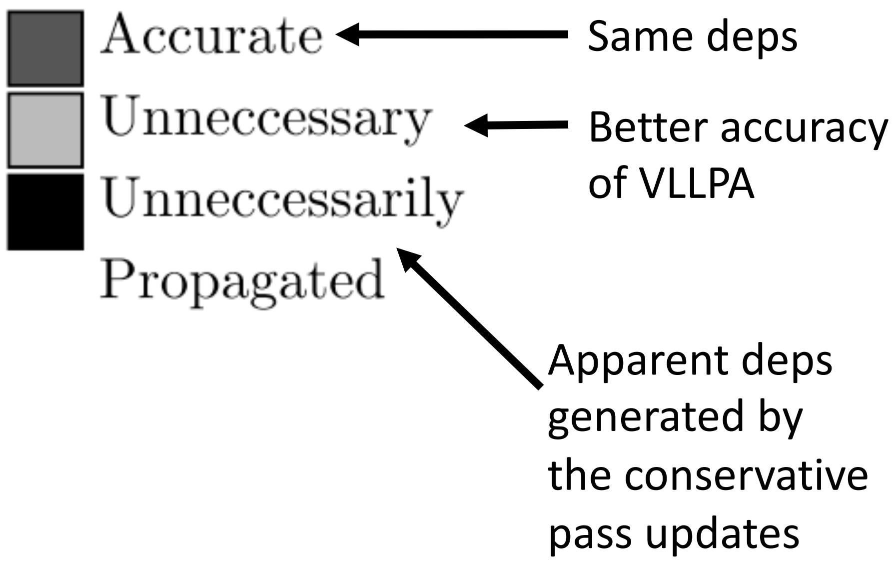
Evaluation: accuracy

Benchmark	# Opers w/ Arcs	VLLPA Arcs	
		More	
099.go	13232		
124.m88ksim	7161		
129.compress	329		
130.li	3762		
164.gzip	1953		
175.vpr	8166		
181.mcf	705		
186.crafty	12026		
256.bzip2	1535		

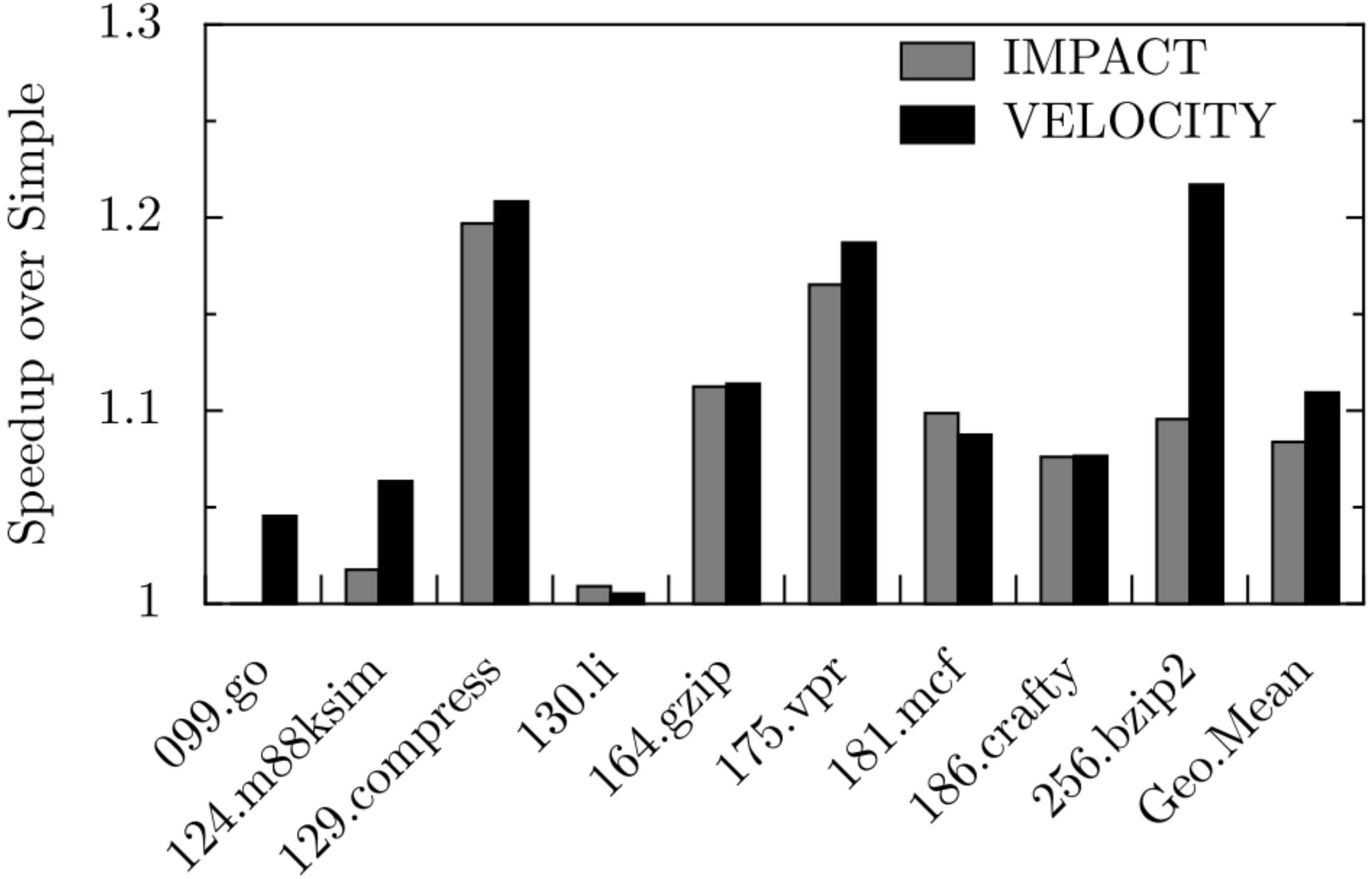
Evaluation: problem of alias analysis at the source language



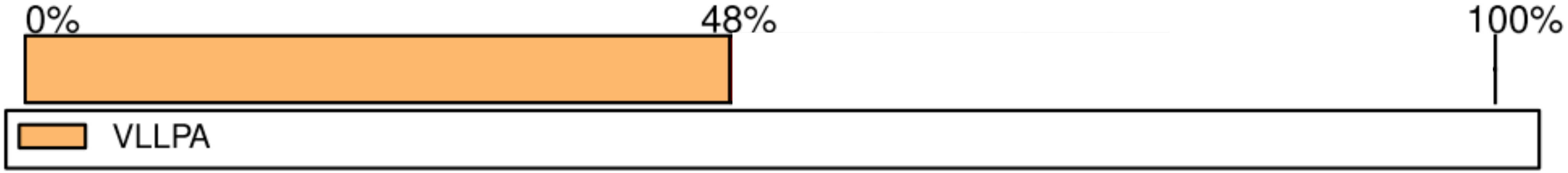
Evaluation: problem of alias analysis at the source language



Evaluation: performance of the generated binary



Improved VLLPA in HELIX-RC (ISCA 2014)



After 2014

- **Approximating Flow-Sensitive Pointer Analysis Using Frequent Itemset Mining**
  Vaivaswatha Nagaraj and R. Govindarajan
  CGO 2015

- … many others

- **A Collaborative Dependence Analysis Framework**
  Nick P. Johnson, Jordan Fix, Taewook Oh, Stephen R. Beard, Thomas B. Jablin, and David I. August
  CGO 2017