Points-To Analysis and Memory Disambiguation

Cody Hartwig

Elie Krevat

Background: Pointer Analysis

- Goal: Determine the set of storage locations that a pointer might reference
- Related techniques:
  - Alias Analysis – Determine if 2 pointers alias the same mutable memory location
    - Flow-insensitive vs. flow-sensitive
    - May-alias vs. must-alias
  - Escape Analysis – Determine the dynamic scope and lifetime of a pointer
- Pointer analysis is hard, but essential for enabling compiler optimizations.

Example Optimizations

- CSE needs info on what is read/written:

```
*p = a + b;
 x = a + b;
```

- Reaching definitions and constant propagation:

```
 x = 5;
*p = 42;
 y = x;
```

- Register variable promotion:

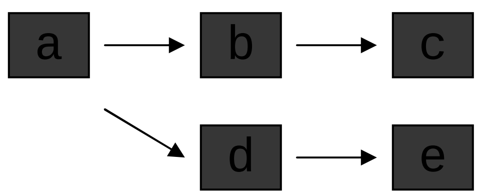
```
   int *p = &s.a;
          s.b = 0;
     *(p + i) = 1;
          ... = s.b;
```

- Scheduling optimizations to hide memory latency
  - Improves IA-64 data speculation

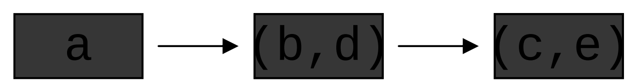
Practical Considerations

- Alias problem is undecidable [Landi 1992]
- Simplest assumption not very useful:
  "Everything *may* alias"
- Andersen vs. Steensgaard: points-to analysis
  - Both are flow-insensitive and context-insensitive
  - Differ in points-to set construction
  - Andersen: many out edges, one variable per node
  - Steensgaard: one out edge, many variables per node

```
a = &b;
b = &c;
a = &d;
d = &e;
```



Andersen



Steensgaard

Memory Disambiguation

- Pointer analysis is just one component of a compiler's memory disambiguator
- Little published on complete framework:
  - How do optimizations interact with and benefit from memory disambiguation?
  - How does this affect program performance?
  - What are the most effective disambiguation techniques?

On the Importance of Points-To Analysis and Other Memory Disambiguation Methods For C Programs

Rakesh Ghiya, Daniel Lavery, and David Sehr

PLDI 2001

Disambiguation Framework

- DISAM: DISambiguation using Abstract Memory locations
  - Maintains high-level symbolic representation
  - LOC represents global/local vars, registers, etc.
    - All LOCs independent
    - Set of possible memory locations → LOC set

Disambiguation Framework

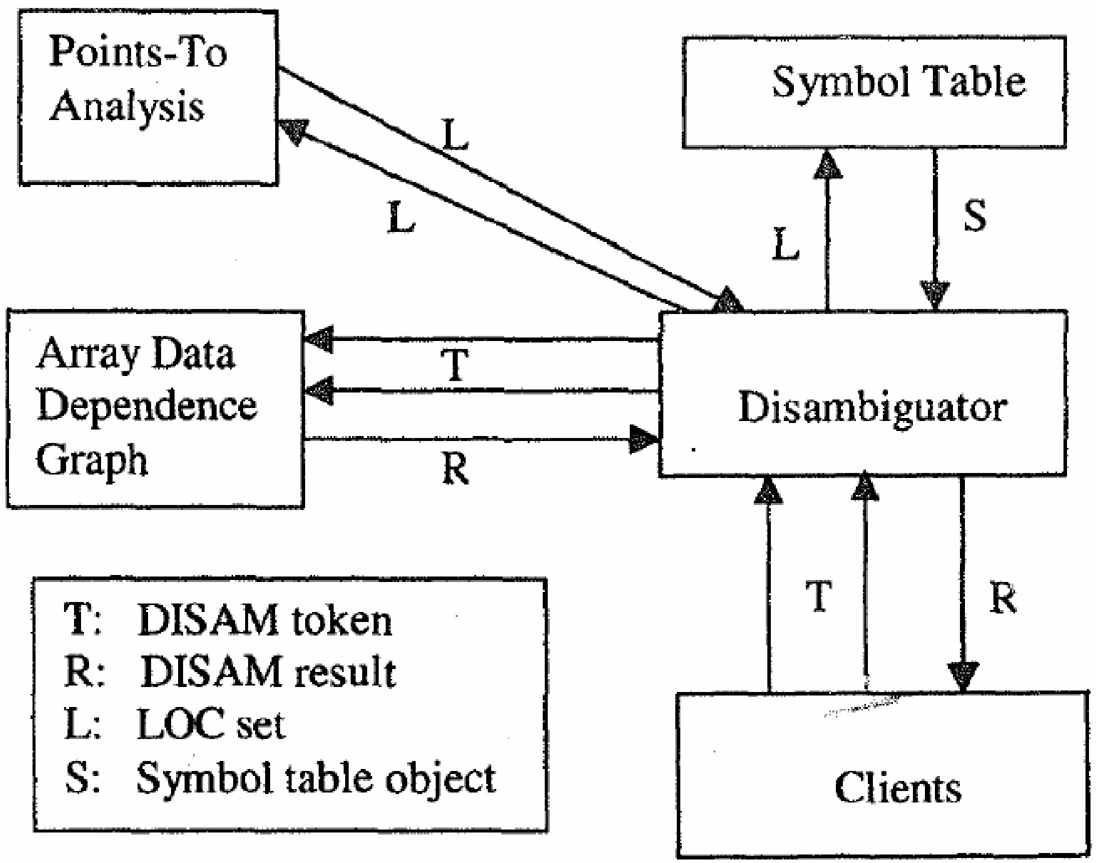


Figure 1 Disambiguation System

Disambiguation Methods

- Intraprocedural (local) methods
  - Direct memory analysis (direct)
    - Includes symbol structure type analysis
  - Simple base and offset analysis (sbo)
  - Indirect memory analysis (indirect)
  - Local points-to analysis (lpt)
  - Array data-dependence analysis (array)
- Interprocedural methods
  - Global address-taken analysis (global)
  - Whole-program points-to analysis (wpt)
- Requiring user assertion for compliance with ANSI C
  - Type-based disambiguation (type)

WPT Framework

- WPT implements Andersen algorithm
- Standard optimizations to reduce cubic complexity:
- More precise structure handling to distinguish fields, but not instances

```
struct foo (int *p; int *q;} s1, s2;
int x,y;
s1.p = &x;
s2.q = &y;
```

s1.p & s2.p point to x
s1.q & s2.q point to y

- Identification of malloc-like functions
  - Determine if malloc is unconditional
  - Determine that address is not taken/stored elsewhere
- Assignment statements visited in sorted topological order

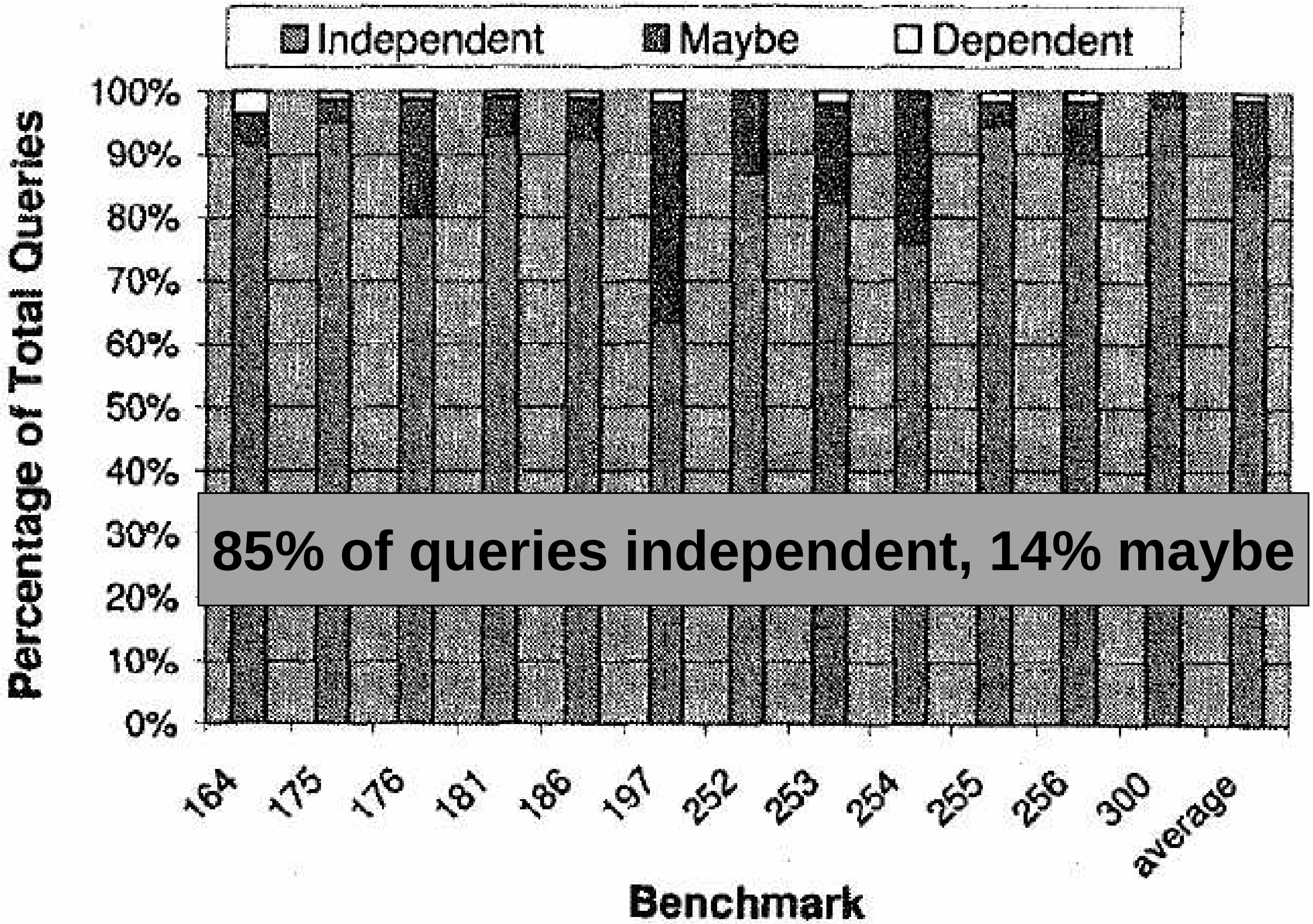
LPT Framework

- Uses same analysis engine as WPT
- Conservative assumptions necessary for global vars and function call effects
  - Symbolic location *nloc* represents all address-escaped vars

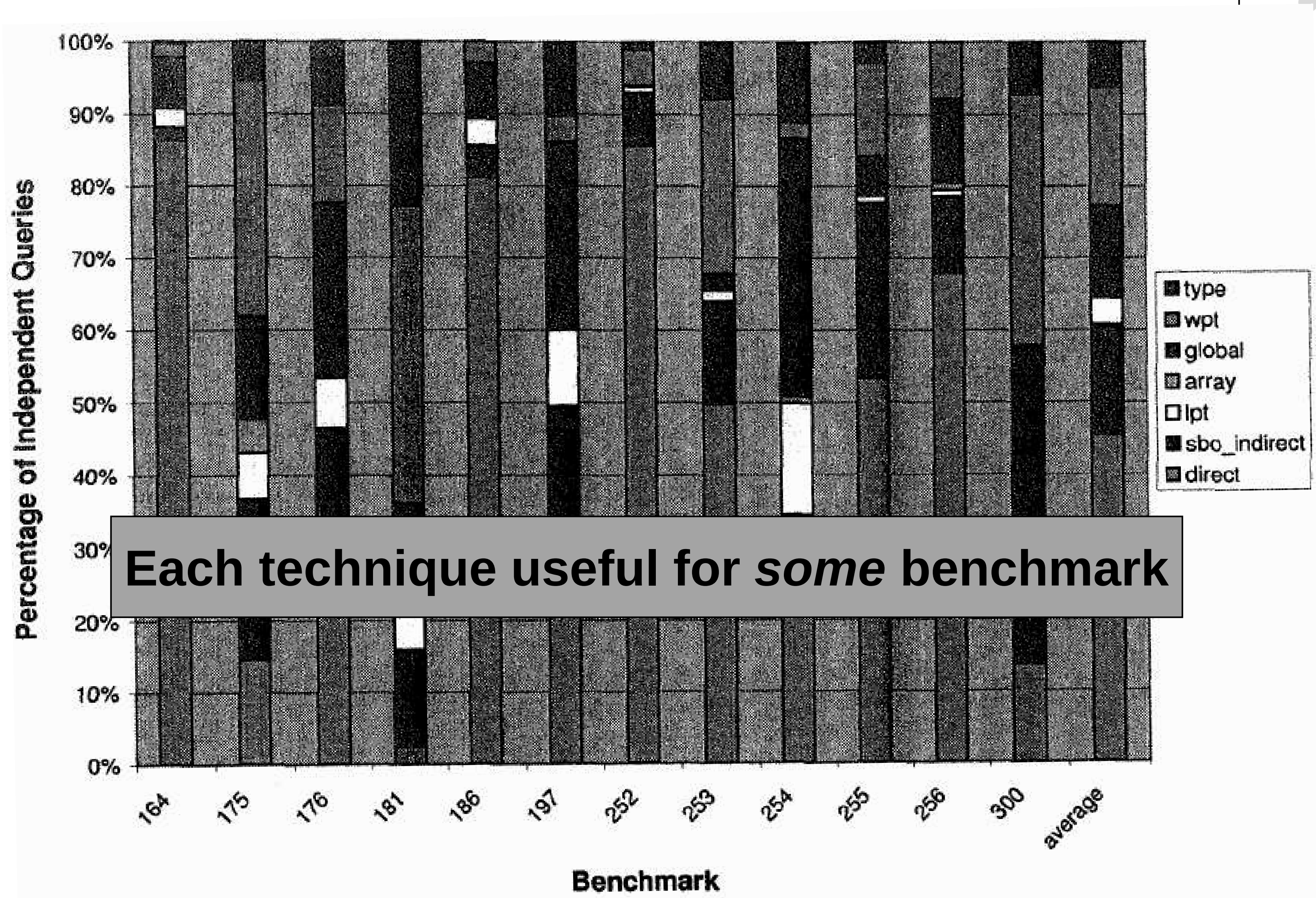
Experiments

- 12 C/C++ benchmarks run on IA-64 hardware
- Highest-optimization level compilation
- Data speculation turned off!
- Data collected:
  - Memory reference characteristics and points-to sets
  - Disambiguation queries (number and type)
  - Hash duplicate disambiguation queries
  - Incremental results for each disambiguation method

Query Results

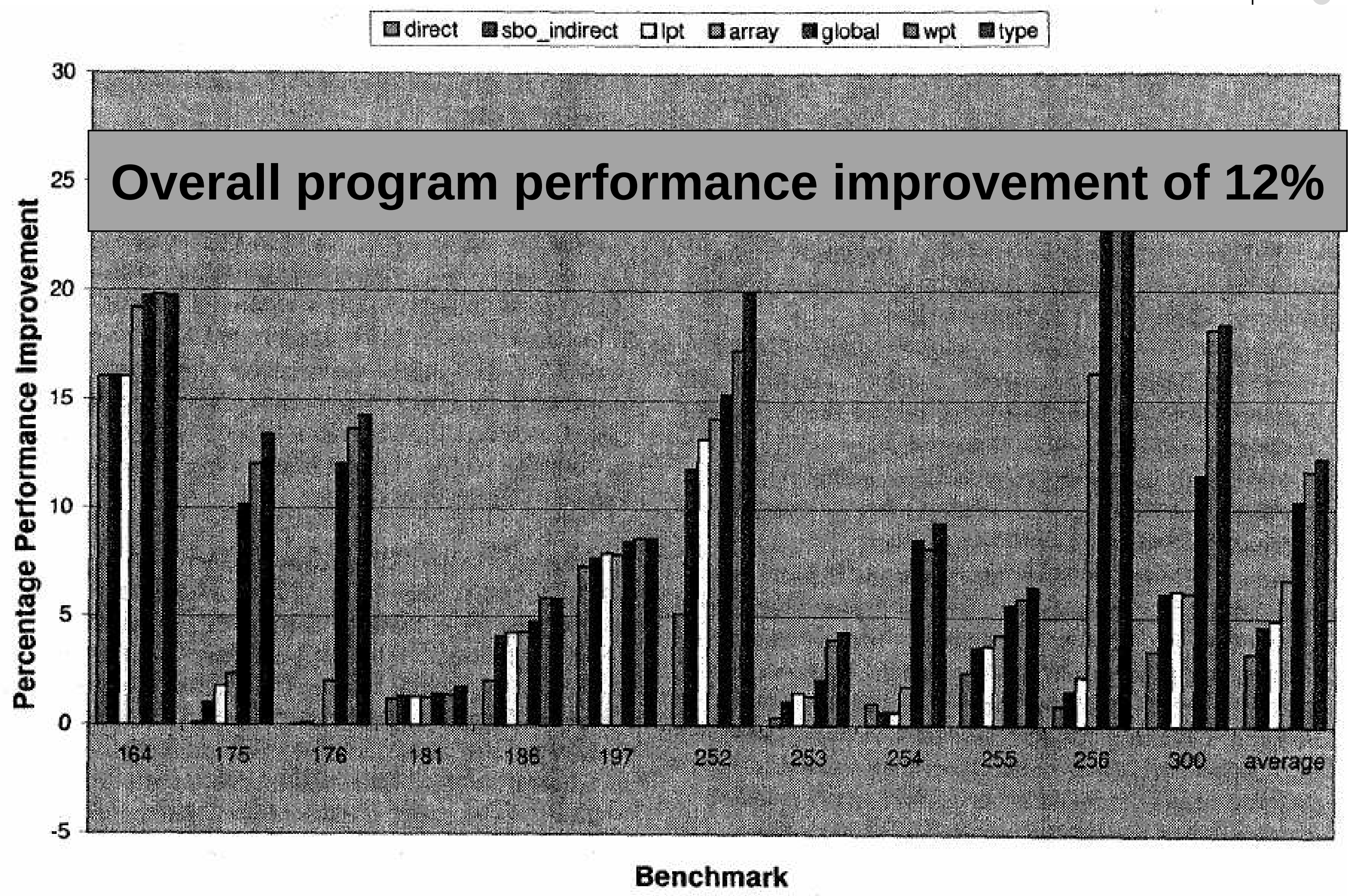


Independent queries by method



Each technique useful for *some* benchmark

Perf Improvement per method



Overall program performance improvement of 12%

Conclusions

- Suite of disambiguation methods provides different tools effective in different situations
- Optimizations to Andersen points-to analysis make algorithm runtime acceptable in practice
- 85% of queries found independent with disambiguation framework
- 14% *maybe* independent references leave some room for improvement
  - unrecognizable malloc wrappers
  - indirect calls with many possible targets

Ultra-fast Aliasing Analysis using CLA: A Million Lines of C a Second

Nevin Heintze
Olivier Tardieu

Motivation

- "…given a million+ lines of C code, and a proposed change of the form 'change the type of this object from type1 to type2', find all the other objects whose type may need to be changed to ensure 'type consistency'…"

Example

```
short x;
short y, z;
short *p, v, w;
y = x;
z = y+1;
p = &v;
*p = z;
w = 1;
```

Example

```
int x;
short y, z;
short *p, v, w;
y = x;
z = y+1;
p = &v;
*p = z;
w = 1;
```

Complications

- Requires analysis of pointers
  - Based on points-to analysis of Andersen[1]
- Must deal with vast amounts of code
  - Modular analysis
  - Defer work to preserve memory and time

Points-to Analysis

- Unification based (Steensgaard)
  - Assignment unifies graph nodes
    - x = y; // unifies nodes for x and y
  - Less accurate, faster
- Subset-relationship based (Andersen)
  - Assignment creates subset relationship
    - x = y; // creates constraint $x \supseteq y$
  - More accurate, slower

Deduction Rules

$\mathbf{Exp} ::= x \mid *x \mid \&x \quad \mathbf{Asn} ::= x = \text{Exp} \mid *x = \text{Exp}$

$\mathbf{Program} ::= \text{Asn} \mid p; \text{Asn}$

When $P \in \mathbf{Program}$ and $e, e_1, e_2 \in \mathbf{Exp}$:

$$\frac{x \to \&y}{y \to e} \text{(if } *x = e \text{ in } P)$$

$$\frac{x \to \&y}{e \to y} \text{(if } e = *x \text{ in } P)$$

$$\frac{}{e_1 \to e_2} \text{ if } (e_1 = e_2 \text{ in } P)$$

$$\frac{e_1 \to e_2 \quad e_2 \to e_3}{e_1 \to e_3}$$

x can point to y if we can derive $x \to y$

`struct` handling

- Field-independent
  - `struct` is treated as unstructured memory region
- Field-based
  - `struct` is treated as separate variables

`struct` Example

Statement	Field-independent	Field-based
`A.x=&z;`	assign to A	assign to x
`p=A.x;`	p gets &z	p gets &z
`q=A.y;`	q gets &z	---
`r=B.x;`	---	r gets &z
`s=B.y;`	---	---

How to scale?

- These analyses are easy to implement for small programs
- Large programs are considerably more difficult
  - Time constraints
  - Memory constraints

Naïve Approach

- Paste all source files together
- Load information from large pasted file
- Analyze information

- Doesn’t scale beyond few thousand LOC

3-phase approach

- Compile
  - parse source files
  - extract assignments, function calls/returns/defns
  - write object file (database)
- Link
  - Merge all object files
- Analyze
  - Use points-to analysis contained in merged object file

Analysis

- Graph algorithm for Andersen's method
- Graph contains node for every variable in program
- Edges of graph show possible sources (dependence) between nodes.
  - If x = y appears, then there is an edge $x \rightarrow y$
- Graph remains in pre-transitive form!
  - Edge $x \rightarrow z$ may not appear, even if $x \rightarrow y$ and $y \rightarrow z$ do appear.

Results

- Uncovered many serious new errors in existing Lucent code. (original goal)
- Capable of analyzing over 1M lines of code in less than 1 second.
  - Misleading: lines of code are not a good indicator of runtime or space
    - Lucent code: 1.3M LOC :: 0.38s 8.8MB
    - GIMP: 440K LOC :: 1.00s 12MB
- Adaptable framework capable of different analyses

Questions?

References

1. L. Anderson, “Program Analysis and Specialization for the C Programming Language”, PhD. thesis, DIKU report 94/19, 1994

Extra Slides

Memory Characteristics

Table 1 Program Memory Reference Characteristics

Program	Local %	Global %	Ind %	Avg Set Size	Total Queries
164.gzip	7	84	9	2.4	26118
175.vpr	16	39	45	1.3	40093
176.gcc	8	31	61	22.1	1237456
181.mcf	8	11	80	1.3	10195
186.crafty	4	87	9	3.7	321026
197.parser	7	39	5	6.9	67642
252.eon	27	40	33	147.7	507662
253.perl	6	36	58	427.3	1192815
254.gap	4	22	74	196.3	286053
255.vortex	34	22	44	39.3	405790
256.bzip2	15	67	18	1.00	13544
300.twolf	2	46	52	3.4	443028