SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA

Yulei Sui [1], Sen Ye [1], Jingling Xue [1], Pen-Chung Yew [2]

School of Computer Science and Engineering, UNSW, Australia

Department of Computer Science and Engineering, University of Minnesota, USA

Dec 05, 2011

What is Pointer Analysis?

Pointer analysis attempts to **statically** determine the **possible runtime value** of a pointer

What is Pointer Analysis?

Pointer analysis attempts to **statically** determine the **possible runtime value** of a pointer

```
a=100;
p=&a;
if(*p==100){
    ...
}
```

Pointer Analysis

Pointer analysis lies at the heart of many program optimization and analysis problems.

- Software Reliability and Security
- Compiler Optimization
- Verification Task
- Parallelization
- Hardware synthesis from C code
- ...

Related Works

		Flow Insensitive		Flow Sensitive	
		Unification-based	Inclusion-based	Iteration-based	SSA-based
Context Insensitive		**Steensgaard** *PLDI'96 1 MLOC* **Manuvir Das** *PLDI'00 2 MLOC*	**Anderson** *1994 5KLOC* **Fahndrich** *PLDI'98 60KLOC* **Heintze** *PLDI'01 1MLOC* **Berndl** *PLDI'03 500KLOC*	**Landi** *PLDI'92 3KLOC* **Choi** *POPL'93 30KLOC*	**Hardekopf** *POPL'09 400KLOC* **Lhotak** *POPL'11 521KLOC* **Hardekopf** *CGO'11 1MLOC*
Context Sensitive	Cloning	**Lattner** *PLDI'07 350KLOC*	**Whaley** *PLDI'04 200KLOC*	**Emami** *PLDI'94 2KLOC* **Zhu** *DAC'05 200KLOC*	
	Summary	**Fahndrich** *PLDI'00 200KLOC*	**Cheng** *PLDI'00 200KLOC* **Nystrom** SAS'04 200KLOC	**Wilson** *PLDI'95 30KLOC* **Chatterjee** *POPL'99 6KLOC* **Kahlon** *PLDI'08 128KLOC*	**Hongtao** *CGO'10 1MLOC*

Inclusion-based Flow Insensitive Analysis

a = & x;

b = & y;

p = & a;

q = & b;

p = q;

s = *p;

$a \rightarrow x$
$b \rightarrow y$
$p \rightarrow a,b$
$q \rightarrow b$
$s \rightarrow x,y$

Flow Sensitive Analysis

a = & x;

$\mathbf{a} \rightarrow \mathbf{x}$

b = & y;

$\mathbf{a} \rightarrow \mathbf{x} \quad \mathbf{b} \rightarrow \mathbf{y}$

p = & a;

$\mathbf{a} \rightarrow \mathbf{x} \quad \mathbf{b} \rightarrow \mathbf{y} \quad \mathbf{p} \rightarrow \mathbf{a}$

q = & b;

$\mathbf{a} \rightarrow \mathbf{x} \quad \mathbf{b} \rightarrow \mathbf{y} \quad \mathbf{p} \rightarrow \mathbf{a} \quad \mathbf{q} \rightarrow \mathbf{b}$

p = q;

$\mathbf{a} \rightarrow \mathbf{x} \quad \mathbf{b} \rightarrow \mathbf{y} \quad \mathbf{p} \rightarrow \mathbf{b} \quad \mathbf{q} \rightarrow \mathbf{b}$

s = *p;

$\mathbf{a} \rightarrow \mathbf{x} \quad \mathbf{b} \rightarrow \mathbf{y} \quad \mathbf{p} \rightarrow \mathbf{b} \quad \mathbf{q} \rightarrow \mathbf{b} \quad \mathbf{s} \rightarrow \mathbf{y}$

Sparse Flow Sensitive Analysis in SSA Form

$a_1 = \& x;$

$\mathbf{a_1 \rightarrow x}$

$b_1 = \& y;$

$\mathbf{b_1 \rightarrow y}$

$p_1 = \& a;$

$\mathbf{p_1 \rightarrow a}$

$q_1 = \& b;$

$\mathbf{q_1 \rightarrow b}$

$p_2 = q_1;$

$\mathbf{p_2 \rightarrow b}$

$s_1 = *p_2;$

$\mathbf{p_2 \rightarrow b}$ $\mathbf{s_1 \rightarrow y}$

POPL'99 Semi-sparse, CGO10' Level-by-Level, CGO11' Sparse-flow

Goal

- Can we compute a **more precise** analysis than flow sensitive points-to analysis?

Goal

- Can we compute a **more precise** analysis than flow sensitive points-to analysis?
- Can we compute the more precise analysis **almost as fast as** flow sensitive analysis?

Capture Path Correlation for Pointer Analysis

```
void main() {
    int **a,*q;
    int *b,*f,d,e,g;
    if(*){
        a = &b; q = &d;
    }
    else{
        a = &f; q = &e;
    }
    *a = q;
}
```

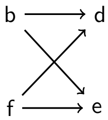
Capture Path Correlation for Pointer Analysis

```
void main() {
    int **a,*q;
    int *b,*f,d,e,g;
    if(*){
        a = &b; q = &d;
    }
    else{
        a = &f; q = &e;
    }
    *a = q;
}
```

*a=q

b=&d   b =&e

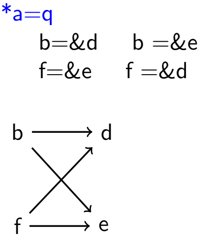
f=&e   f =&d



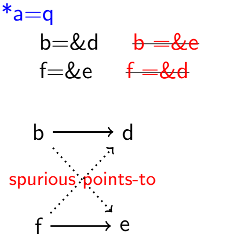
flow sensitive points-to

Capture Path Correlation for Pointer Analysis

```
void main() {
    int **a,*q;
    int *b,*f,d,e,g;
    if(*){
        a = &b; q = &d;
    }
    else{
        a = &f; q = &e;
    }
    *a = q;
}
```



flow sensitive points-to

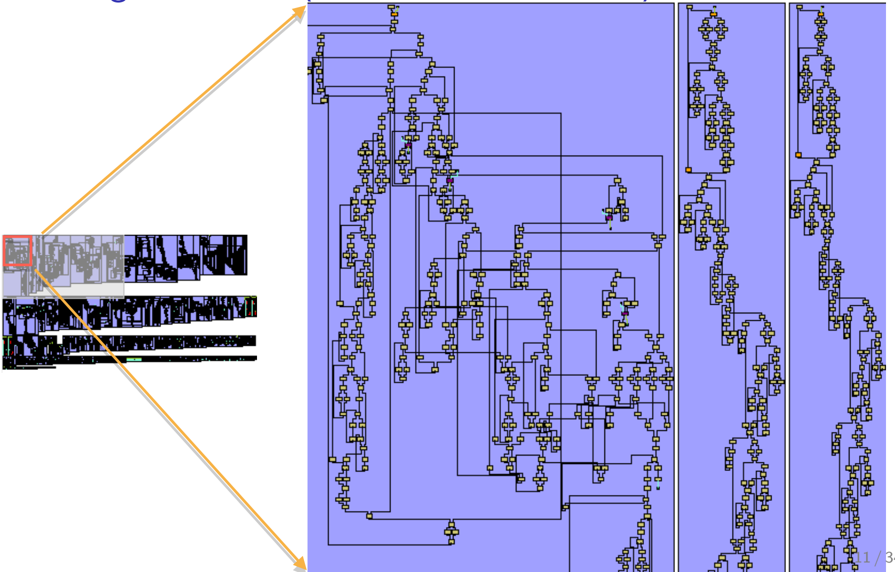


path sensitive points-to

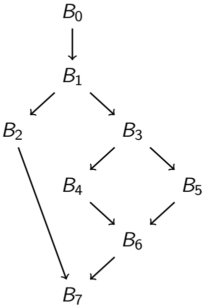
SPAS

- SPAS enables *intra procedural* **path sensitivity** on Flow- and Context- Sensitive(FSCS) Pointer analysis on *full sparse SSA*

Real Program Paths (SPEC2000 300.twolf)

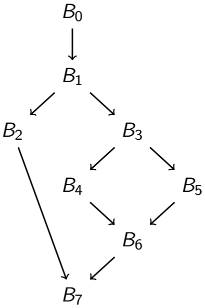


Generating Path Condition for Basic Blocks

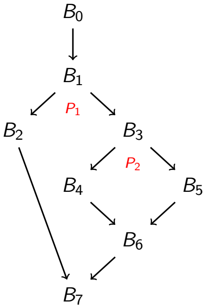


(a) Control Flow Graph

Generating Path Condition for Basic Blocks

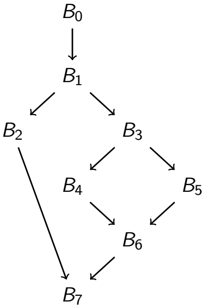


(d) Control Flow Graph

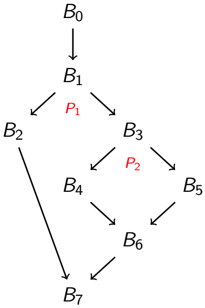


(e) Allocating decision variable

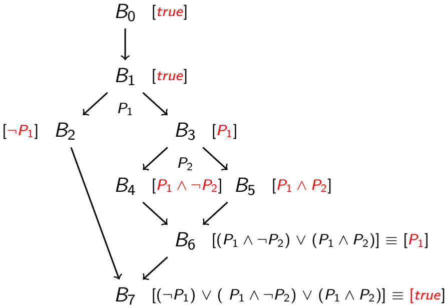
Generating Path Condition for Basic Blocks



(g) Control Flow Graph



(h) Allocating decision variable

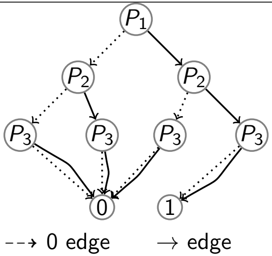


(i) basic block after labeling

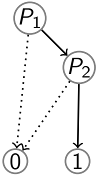
Encoding Path Condition into BDD

The following path condition holds for a points-to relation:

$(P_1 \wedge P_2 \wedge P_3) \vee (P_1 \wedge P_2 \wedge \neg P_3)$



(a) unreduced BDD



(b) after redundancy elimination

Points-to Relation and Conditions

PtrMap(p) = (Loc(p),Dep(p))

- Loc(p) **local method** points-to
- Dep(p) points-to **depends on** formal-in parameters

Points-to Relation and Conditions

PtrMap(p) = (Loc(p),Dep(p))

- Loc(p) **local method** points-to
- Dep(p) points-to **depends on** formal-in parameters

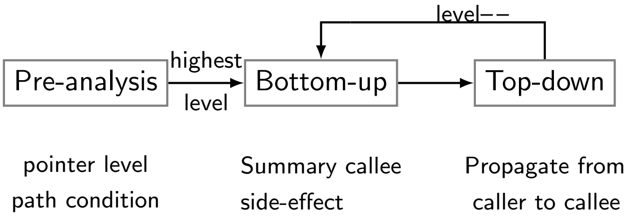
PtrMap(**p**) = $(\{\mathbf{a}, C_a, P_a\}, \{\mathbf{q}, C_q, P_q\})$

- **p** points to local variable **a** under *calling context* C_a along *path* P_a,
- **p** points to what formal parameter **q** points to under *calling context* C_q along *path* P_q.

To be simple, context conditions are true in default in the following talk, if you are interested please refer to our paper

SPAS Analysis Framework

Pointers are analyzed level by level regarding to fast unification-based pre-analysis



Pre-analysis

Pointer level and Path conditions

```
int *q, v, w, z;
void main() {
    int **a, *f = &z;
    a = &f; q = &v;
    foo(a);
}
 void foo(int **x) {
     int *g = &z;
     if (*) {
         x = &g; q = &w;
     }
     *x = q;
 }
```

Pointer level:

2nd level: a, x

1st level: f, q, g

Pre-analysis

Pointer level and Path conditions

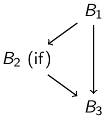
```
int *q, v, w, z;
void main() {
    int **a, *f = &z;
    a = &f; q = &v;
    foo(a);
}
 void foo(int **x) {
     int *g = &z;
     if (*) {
        x = &g; q = &w;
     }
     *x = q;
 }
```

Pointer level:

2nd level: a, x

1st level: f, q, g

Path Condition



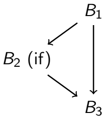
CFG of foo

Path $P_1 : B_1 \to B_2 \to B_3$

Path $\neg P_1 : B_1 \to B_3$

Bottom-up Analyzing **2nd Level** Pointers

```
int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a0 = &f; q = &v;

    foo(a0);

}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)


    int *g = &z;
    if (*) {
        x1 = &g; q = &w;
    }
    x2 = φ(x0, x1);

    *x2 = q;


}
```



Path $P_1 : B_1 \to B_2 \to B_3$

Path $\neg P_1 : B_1 \to B_3$

main:

$$\mathrm{PtrMap}(a_0) = (\{(f, \mathrm{true})\}, \emptyset)$$

foo:

$$\begin{aligned} \mathrm{PtrMap}(x_0) &= (\emptyset, \{(x, \mathrm{true})\}) \\ \mathrm{PtrMap}(x_1) &= (\{(g, P_1)\}, \emptyset) \\ \mathrm{PtrMap}(x_2) &= (\{(g, P_1)\}, \{(x, \neg P_1)\}) \end{aligned}$$

Top-down Analyzing **2nd Level** Pointers

(1)Add Mu/Chi for dereference

```
int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a0 = &f; q = &v;

    foo(a0);

}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)


    int *g = &z;
    if (*) {
        x1 = &g; q = &w;
    }
    x2 = φ(x0, x1);

    *x2 = q;
        f = χ(f, ¬P1); // ¬P1 guard
        g = χ(g, P1); // P1 guard
}
```

(2)Propagate Points-to

foo:

$\text{PtrSet}(x_0) = \{f\}$

Bottom-up Analyzing **1st Level** Pointers

Create Callsite Mu/Chi

```
int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a0 = &f; q = &v;
      μ(q, true);
    foo(a0);
      f = χ(f, true);
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)


    int *g = &z;
    if (∗) {
        x1 = &g; q = &w;
    }
    x2 = φ(x0, x1);

    *x2 = q;
      f = χ(f, ¬P1);
      g = χ(g, P1);
}
```

Bottom-up Analyzing **1st Level** Pointers

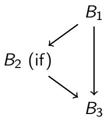
Build SSA for **foo**

```
int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a0 = &f; q = &v;
     μ(q, true);
    foo(a0);
     f = χ(f, true);
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)
    q0 = q; // formal-in q identified as q0 (ver 0)
    f0 = f;  // formal-in f identified as f0 (ver 0)
    int *g0 = &z;
    if (∗) {
        x1 = &g; q1 = &w;
    }
    x2 = φ(x0, x1);
    q2 = φ(q0, q1);
    *x2 = q2;
      f1 = χ(f0, ¬P1);
      g1 = χ(g0, P1);
}
```

Bottom-up Analyzing **1st Level** Pointers

Pointer Inference for **foo**

```
int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a0 = &f; q = &v;
     μ(q, true);
    foo(a0);
     f = χ(f, true);
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)
    q0 = q; // formal-in q identified as q0 (ver 0)
    f0 = f;  // formal-in f identified as f0 (ver 0)
    int *g0 = &z;
    if (*) {
       x1 = &g; q1 = &w;
    }
    x2 = φ(x0, x1);
    q2 = φ(q0, q1);
    *x2 = q2;
      f1 = χ(f0, ¬P1); // ¬P1 guard
      g1 = χ(g0, P1); // P1 guard
}
```



Path $P_1 : B_1 \rightarrow B_2 \rightarrow B_3$

Path $\neg P_1 : B_1 \rightarrow B_3$

foo:

$$\text{PtrMap}(q_2) = (\{(w, P_1)\}, \{(q, \neg P_1)\})$$

$$\text{PtrMap}(f_1) = (\{(w, \mathbf{P_1} \vee \neg \mathbf{P_1})\}, \{(q, \neg P_1), (f, P_1)\})$$

$$\text{PtrMap}(g_1) = (\{(z, \neg P_1), (w, P_1)\}, \{(q, \mathbf{P_1} \vee \neg \mathbf{P_1})\})$$

Bottom-up Analyzing **1st Level** Pointers

Pointer Inference for **foo**

int *q, v, w, z;
void main() {

    int **a, *f = &z;
    a_0 = &f; q = &v;
    μ(q, true);
    foo(a_0);
    f = χ(f, true);
}
void foo(int **x) {
    x_0 = x; // *formal-in x identified as x_0 (ver 0)*
    q_0 = q; // *formal-in q identified as q_0 (ver 0)*
    f_0 = f; // *formal-in f identified as f_0 (ver 0)*
    int *g_0 = &z;
    if (∗) {
        x_1 = &g; q_1 = &w;
    }
    $x_2 = \phi(x_0, x_1)$;
    $q_2 = \phi(q_0, q_1)$;
    *x_2 = q_2;
    $f_1 = \chi(f_0, \neg P_1)$; // $\neg P_1$ guard
    $g_1 = \chi(g_0, P_1)$; // P_1 guard
}

$$
\begin{tikzcd}
 & B_1 \arrow[dl] \arrow[dd] \\
B_2\ (\text{if}) \arrow[dr] & \\
 & B_3
\end{tikzcd}
$$

Path $P_1 : B_1 \rightarrow B_2 \rightarrow B_3$

Path $\neg P_1 : B_1 \rightarrow B_3$

foo:

$\mathrm{PtrMap}(q_2) = (\{(w, P_1)\}, \{(q, \neg P_1)\})$

$\mathrm{PtrMap}(f_1) = (\{\cancel{(w, \mathbf{P_1} \wedge \neg \mathbf{P_1})}\}, \{(q, \neg P_1), (f, P_1)\})$

path conflict !

$\mathrm{PtrMap}(g_1) = (\{(z, \neg P_1), (w, P_1)\}, \{\cancel{(q, \mathbf{P_1} \wedge \neg \mathbf{P_1})}\})$

Bottom-up Analyzing **1st Level** Pointers

Build SSA / Pointer Inference for **main**

```
int *q, v, w, z;
void main() {
    q0 = q; // formal-in q identified as q0 (ver 0)
    int **a, *f0 = &z;
    a0 = &f; q1 = &v;
     μ(q1, true);
    foo(a0);
     f1= χ(f0, true);
    //More precised side-effect from callee
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)
    q0 = q; // formal-in q identified as q0 (ver 0)
    f0 = f;  // formal-in f identified as f0 (ver 0)
    int *g0 = &z;
    if (*) {
        x1 = &g; q1 = &w;
    }
    x2 = φ(x0, x1);
    q2 = φ(q0, q1);
    *x2 = q2;
     f1 = χ(f0, ¬P1);
     g1 = χ(g0, P1);
}
```

main:

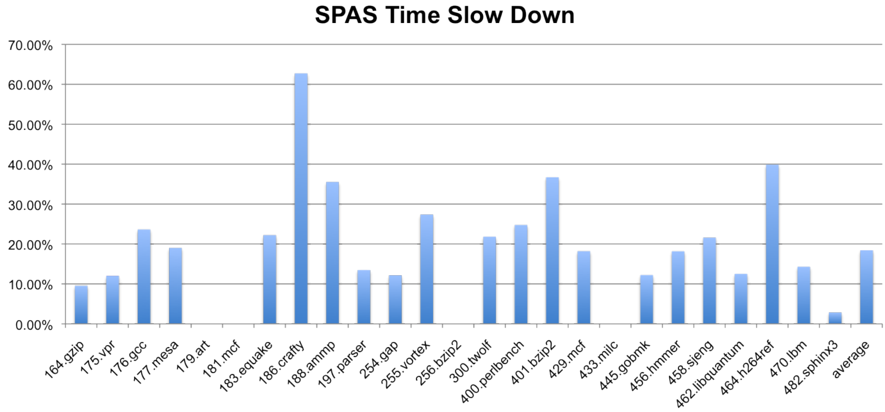
$$\mathrm{PtrMap}(f_0) = (\{(z, \mathrm{true})\}, \emptyset)$$
$$\mathrm{PtrMap}(q_1) = (\{(v, \mathrm{true})\}, \emptyset)$$
$$\mathrm{PtrMap}(f_1) = (\{(z, \mathrm{true}), (v, \mathrm{true})\}, \emptyset)$$

Benchmark Characteristics

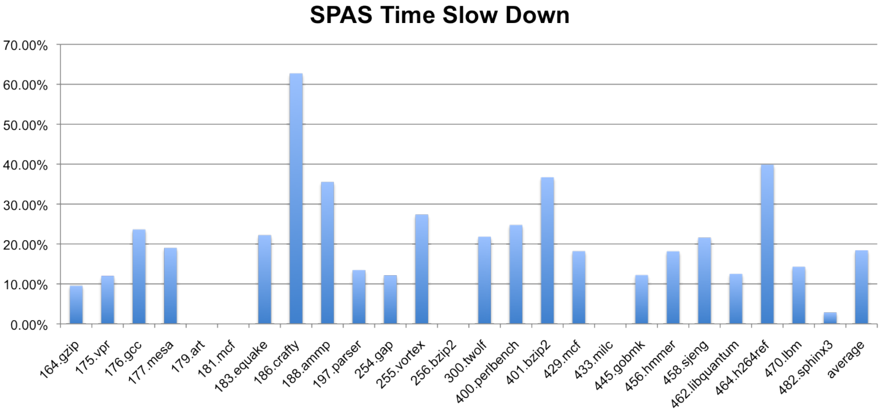
Benchmark	Characteristics						
	KLOC	#Pointers	#Methods	#Callsites	#Indirect CallSites	#Recur. Cycles	#Max Recur. Size
164.gzip	8.6	617	96	418	2	0	0
175.vpr	17.8	2202	275	1995	2	0	0
176.gcc	230.4	17380	2171	22353	140	**179**	**398**
177.mesa	61.3	9504	1108	3611	671	1	1
179.art	1.2	215	29	163	0	0	0
181.mcf	2.5	230	29	82	0	1	1
183.equake	1.5	246	30	215	0	0	0
186.crafty	21.2	1341	111	4046	0	5	2
188.ammp	13.4	2427	182	1201	24	6	2
197.parser	11.4	1580	327	1782	0	42	3
253.perlbmk	87.1	7488	1075	8470	58	12	**322**
254.gap	71.5	6162	857	5980	1275	33	20
255.vortex	67.3	10029	926	8522	15	12	38
256.bzip2	4.7	469	77	402	0	0	0
300.twolf	20.5	2470	194	2074	0	5	1
400.perlbench	169.9	13283	1830	14593	140	**30**	**461**
401.bzip2	8.3	765	103	430	20	0	0
403.gcc	521.1	35697	5250	50673	461	**317**	**436**
429.mcf	2.7	225	27	78	0	1	1
433.milc	15.0	1861	238	1592	4	0	0
445.gobmk	197.2	16789	929	6147	44	26	22
456.hmmer	36.0	3668	545	4085	10	6	4
458.sjeng	13.9	1122	147	1257	1	6	1
462.libquantum	4.4	603	118	508	0	2	5
464.h264ref	51.6	5068	597	3382	369	2	34
470.lbm	1.2	170	26	74	0	0	0
482.sphinx3	25.1	2558	376	2771	8	8	1

Benchmark	Analysis Overhead				SPAS								
	Time (secs)		Memory (MBs)		D-Vars	Time Breakdown (secs)							
	LevPA	SPAS(%)	LevPA	SPAS(%)		Comp. Levels	Gen. Paths	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
164.gzip	0.42	9.52	20.97	9.31	269	0.09	0.02	0.07	0.22	0.02	0.00	0.04	0.00
175.vpr	1.11	12.00	55.78	9.62	639	0.34	0.03	0.09	0.50	0.15	0.01	0.08	0.04
176.gcc	1230.76	23.62	6576.05	9.91	16043	4.05	1.02	129.99	346.20	926.12	15.70	95.73	2.71
177.mesa	8.21	19.01	247.29	12.41	6242	2.84	0.25	0.48	2.90	0.83	0.11	0.23	2.13
179.art	0.08	0.00	5.28	10.51	47	0.03	0.00	0.00	0.03	0.01	0.00	0.00	0.01
181.mcf	0.13	0.00	5.74	7.52	34	0.03	0.00	0.00	0.05	0.01	0.00	0.01	0.03
183.equake	0.09	22.20	5.62	10.47	50	0.04	0.01	0.00	0.05	0.00	0.01	0.00	0.00
186.crafty	3.65	62.73	136.44	11.33	1517	0.55	0.06	0.48	3.06	1.55	0.11	0.07	0.06
188.ammp	2.28	35.53	58.94	5.93	804	0.03	0.09	1.11	1.54	0.03	0.06	0.20	0.03
197.parser	15.31	13.46	133.60	10.60	570	0.27	0.03	0.23	1.99	13.44	0.04	1.20	0.17
254.gap	21.71	12.16	440.50	4.90	7482	1.91	0.31	4.92	7.43	4.20	0.51	4.23	0.84
255.vortex	19.37	27.36	624.01	5.24	6019	1.91	0.33	4.88	8.69	3.67	0.44	4.30	0.45
256.bzip2	0.20	0.00	13.29	10.45	144	0.06	0.00	0.03	0.07	0.01	0.02	0.01	0.00
300.twolf	1.65	21.82	64.22	7.94	520	0.52	0.03	0.09	0.80	0.37	0.00	0.08	0.12
400.perlbench	971.20	24.75	4111.17	9.11	13218	2.98	0.87	105.84	277.65	680.99	13.01	125.60	4.67
401.bzip2	0.79	36.71	24.52	16.68	530	0.17	0.02	0.03	0.66	0.07	0.01	0.01	0.11
429.mcf	0.11	18.18	4.95	24.45	37	0.03	0.00	0.00	0.03	0.03	0.00	0.00	0.04
433.milc	0.87	0.00	45.05	10.23	469	0.32	0.02	0.17	0.19	0.08	0.01	0.04	0.04
445.gobmk	14.66	12.21	682.00	16.64	3680	1.45	0.23	3.14	5.83	2.95	0.22	2.26	0.37
456.hmmer	2.71	18.15	45.97	14.00	1673	0.86	0.05	0.12	1.30	0.50	0.03	0.10	0.86
458.sjeng	1.39	21.58	55.78	11.93	1060	0.27	0.06	0.24	0.65	0.34	0.01	0.10	0.02
462.libquantum	0.32	12.50	0.00	10.41	141	0.07	0.02	0.10	0.11	0.03	0.00	0.02	0.01
464.h264ref	5.77	39.86	247.29	11.06	2457	1.44	0.16	0.80	2.97	1.37	0.16	0.47	0.70
470.lbm	0.07	14.29	5.28	11.52	19	0.04	0.00	0.01	0.02	0.00	0.00	0.00	0.01
482.sphinx3	1.76	2.84	5.74	11.98	835	0.53	0.07	0.13	0.65	0.16	0.02	0.08	0.17

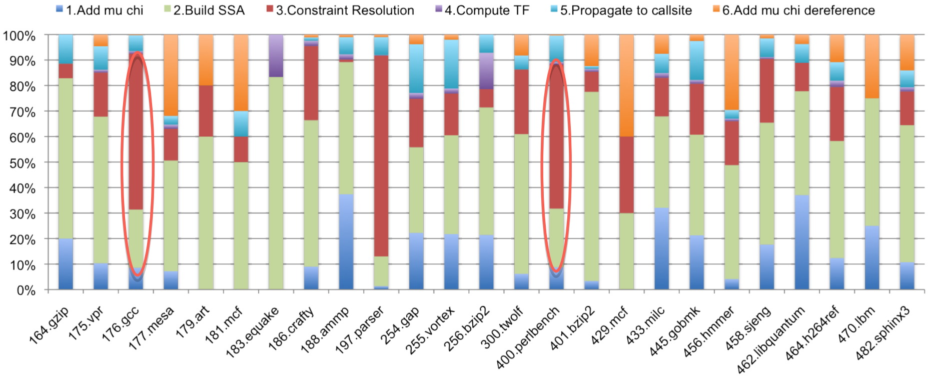
Analysis Time



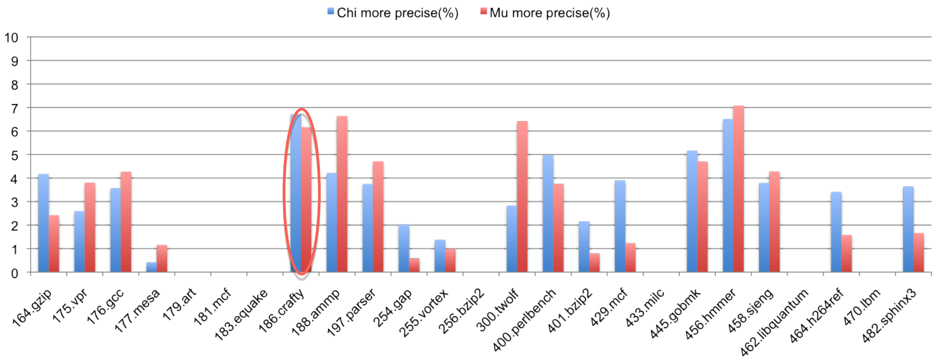
Analysis Time



Time Break Down



More Precise Points-to at Stores/Loads



Conclusions and Future work

- SPAS can be extended simply and efficiently on FSCS analysis with little analysis overhead.
- SPAS analysis computes more precise points-to information
- SPAS analysis performs on full sparse SSA form
- Future work
  - Perform inter-procedural path sensitive pointer analysis
  - Improve points-to analysis precision by eliminating infeasible paths

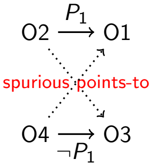
Backup Slides (Path Correlations Captured by SPAS)

SPEC 2000, 186.crafty file store.c (simplfied version)

```
trans_ref_wa=malloc(16*hash_table_size); //(O1)
trans_ref_wb=malloc(16*2*hash_table_size); //(O2)
trans_ref_ba=malloc(16*hash_table_size); //(O3)
trans_ref_bb=malloc(16*2*hash_table_size); //(O4)

if (wtm) {
    htablea=trans_ref_wa;
    htableb=trans_ref_wb;
}
else {
    htablea=trans_ref_ba;
    htableb=trans_ref_bb;
}
......
htableb→word1=htablea→word1;
```

PtrMap(htablea) = $(\{(P_1, O1), (\neg P_1, O3)\}, \emptyset)$

PtrMap(htablea) = $(\{(P_1, O2), (\neg P_1, O4)\}, \emptyset)$



Backup Slides (Path Correlations Captured by SPAS)

SPEC 2000, 188.ammp file variable.c (simplfied version)

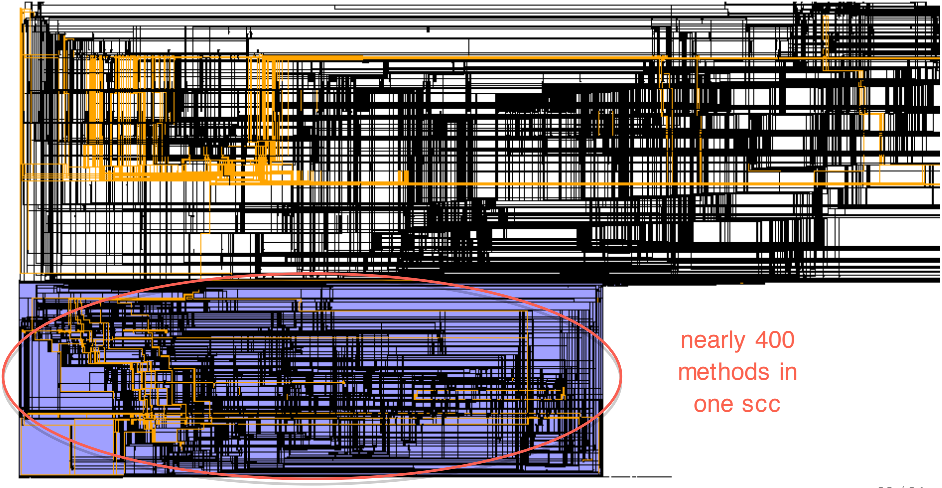
```
// ptr(variableLAST) (O1)
new = match_variable( name); // (O2)
if( new == NULL)
{
    new = malloc( variableLONG ) // (O3)
......
    variableLAST = new;
}
......
variableLAST→ next = new;
```

PtrMap(htablea) =
$(\{(\neg P_1, O1), (P_1, O3)\}, \emptyset)$

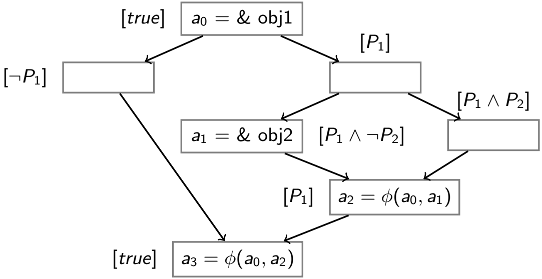
PtrMap(htablea) =
$(\{(P_1, O2), (\neg P_1, O3)\}, \emptyset)$

$$\begin{tikzcd} O1 \arrow[r, "\neg P_1"] \arrow[dr, dotted, "\text{spurious points-to}"'] & O2 \\ & O3 \end{tikzcd}$$

Backup Slides SPEC 2000 176.gcc Call Graph



Points-to relation with path conditions



$Ptr(a_0) = \{\text{obj1}, \text{true}\}$

$Ptr(a_1) = \{(\text{obj2}, P_1 \wedge \neg P_2)\}$

$Ptr(a_2) = \{(\text{obj1}, P_1 \wedge P_2), (\text{obj2}, P_1 \wedge \neg P_2)\}$

$Ptr(a_3) = \{(\text{obj1}, (P_1 \wedge P_2) \vee (\neg P_1)), (\text{obj2}, P_1 \wedge \neg P_2)\}$