Hash Consed Points-To Sets

Mohamad Barbar [1,2] Yulei Sui [1]

[1]University of Technology Sydney, Australia

[2]CSIRO's Data61, Australia

SAS '21

Points-To Analysis

Points-To Analysis

Determine what each pointer points to.

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, \ldots\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, \ldots\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

- Field

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

- Field
- Flow

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

- Field
- Flow
- Context

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, \ldots\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

- Field
- Flow
- Context
- ...

Points-To Analysis

Determine what each pointer points to.

$$pt(p) = \{o_1, o_2, o_3, ...\}$$

(representation: bit-vectors, BDDs, B-trees, etc.)

Why?

- Bug detection
- Optimisation
- Instrumentation

Various sensitivities

- Field
- Flow
- Context
- ...

(generally)

Precision $\Rightarrow$ higher cost

Flow-Insensitive Points-to Analysis

$$[\text{ALLOC}]\ \frac{p = alloc_o}{o \in pt(p)} \qquad [\text{COPY}]\ \frac{p = q}{pt(q) \subseteq pt(p)}$$

$$[\text{LOAD}]\ \frac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)} \qquad [\text{STORE}]\ \frac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)}$$

$$[\text{FIELD}]\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{o.f_i \in pt(p)}$$

Flow-Insensitive Points-to Analysis

$$[\text{ALLOC}]\ \frac{p = alloc_o}{o \in pt(p)} \qquad [\text{COPY}]\ \frac{p = q}{pt(q) \subseteq pt(p)}$$

$$[\text{LOAD}]\ \frac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)} \qquad [\text{STORE}]\ \frac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)}$$

$$[\text{FIELD}]\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{o.f_i \in pt(p)}$$

Flow-Sensitive Points-to Analysis

$pt[\overline{\ell}](o)$: points-to set of o immediately before ℓ. $pt[\underline{\ell}](o)$: points-to set of o immediately after ℓ.

Flow-Sensitive Points-to Analysis

$pt[\overline{\ell}](o)$: points-to set of o immediately before ℓ. $pt[\underline{\ell}](o)$: points-to set of o immediately after ℓ.

$$[\text{ALLOC}]\ \frac{\ell : p = alloc_o}{\{o\} \subseteq pt(p)} \qquad [\text{COPY}]\ \frac{\ell : p = q}{pt(q) \subseteq pt(p)}$$

$$[\text{FIELD}]\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{o.f_i \in pt(p)}$$

$$[\text{LOAD}]\ \frac{\ell : p = *q \quad o \in pt(q)}{pt[\overline{\ell}](o) \subseteq pt(p)} \qquad [\text{STORE}]\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{SU/WU}]\ \frac{\ell : *p = \_ \quad o \in \mathcal{O} \setminus kill(\ell)}{pt[\overline{\ell}](o) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{CONTROL-FLOW}]\ \frac{\ell \rightarrow \ell'}{\forall o \in \mathcal{O}.\ pt[\underline{\ell}](o) \subseteq pt[\overline{\ell'}](o)}$$

$$kill(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \land o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Flow-Sensitive Points-to Analysis

$pt[\overline{\ell}](o)$: points-to set of o immediately before ℓ. $pt[\underline{\ell}](o)$: points-to set of o immediately after ℓ.

$$[\text{ALLOC}]\ \frac{\ell : p = alloc_o}{\{o\} \subseteq pt(p)} \qquad [\text{COPY}]\ \frac{\ell : p = q}{pt(q) \subseteq pt(p)}$$

$$[\text{FIELD}]\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{o.f_i \in pt(p)}$$

$$[\text{LOAD}]\ \frac{\ell : p = *q \quad o \in pt(q)}{pt[\overline{\ell}](o) \subseteq pt(p)} \qquad [\text{STORE}]\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{SU/WU}]\ \frac{\ell : *p = \_ \quad o \in \mathcal{O} \setminus kill(\ell)}{pt[\overline{\ell}](o) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{CONTROL-FLOW}]\ \frac{\ell \rightarrow \ell'}{\forall o \in \mathcal{O}.\ pt[\underline{\ell}](o) \subseteq pt[\overline{\ell'}](o)}$$

$$kill(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Flow-Sensitive Points-to Analysis

$pt[\overline{\ell}](o)$: points-to set of o immediately before ℓ. $pt[\underline{\ell}](o)$: points-to set of o immediately after ℓ.

$$[\text{ALLOC}]\ \frac{\ell : p = alloc_o}{\{o\} \subseteq pt(p)} \qquad [\text{COPY}]\ \frac{\ell : p = q}{pt(q) \subseteq pt(p)}$$

$$[\text{FIELD}]\ \frac{p = \&q \rightarrow f_i \quad o \in pt(q)}{o.f_i \in pt(p)}$$

$$[\text{LOAD}]\ \frac{\ell : p = *q \quad o \in pt(q)}{pt[\overline{\ell}](o) \subseteq pt(p)} \qquad [\text{STORE}]\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{SU/WU}]\ \frac{\ell : *p = \_ \quad o \in \mathcal{O} \setminus kill(\ell)}{pt[\overline{\ell}](o) \subseteq pt[\underline{\ell}](o)}$$

$$[\text{CONTROL-FLOW}]\ \frac{\ell \rightarrow \ell'}{\forall o \in \mathcal{O}.\ pt[\underline{\ell}](o) \subseteq pt[\overline{\ell'}](o)}$$

$$kill(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Motivation

Common points-to sets – why store twice?

Motivation

Common points-to sets – why store twice?

5 most common points-to sets appear as the solution for around

- 60% of pointers (flow-insensitive)
- 90% of pointers (flow-sensitive)

Motivation

Common points-to sets – why store twice?

5 most common points-to sets appear as the solution for around

- 60% of pointers (flow-insensitive)
- 90% of pointers (flow-sensitive)

Common operations – why perform twice?

Motivating Example

Let us analyse the program fragment assuming $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$.

$$
\begin{aligned}
&1 : *p = r; \\
&2 : *q = r; \\
&3 : x = *p; \\
&4 : y = *q;
\end{aligned}
$$

(a) Program fragment.

$$
\begin{aligned}
&1 : \forall o \in pt(p).\ pt(r) \subseteq pt(o) \\
&2 : \forall o \in pt(q).\ pt(r) \subseteq pt(o) \\
&3 : \forall o \in pt(p).\ pt(o) \subseteq pt(x) \\
&4 : \forall o \in pt(q).\ pt(o) \subseteq pt(y)
\end{aligned}
$$

(b) Constraints.

$$
\begin{aligned}
&1 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_1} \\
&2 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_2} \\
&3 : \{o_3, o_4\}_{o_1} \subseteq \{\ \}_x \\
&4 : \{o_3, o_4\}_{o_2} \subseteq \{\ \}_y
\end{aligned}
$$

(c) Operations.

$$
pt(p) = \{o_1\} \quad pt(q) = \{o_2\} \quad pt(r) = \{o_3, o_4\} \quad pt(x) = \{o_3, o_4\}
$$
$$
pt(y) = \{o_3, o_4\} \quad pt(o_1) = \{o_3, o_4\} \quad pt(o_2) = \{o_3, o_4\}
$$
$$
pt(o_3) = \{\ \} \quad pt(o_4) = \{\ \}
$$

(d) Result.

Motivating Example

Let us analyse the program fragment assuming $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$.

$$
\begin{aligned}
&1 : *p = r;\\
&2 : *q = r;\\
&3 : x = *p;\\
&4 : y = *q;
\end{aligned}
$$

(a) Program fragment.

$$
\begin{aligned}
&1 : \forall o \in pt(p).\ pt(r) \subseteq pt(o)\\
&2 : \forall o \in pt(q).\ pt(r) \subseteq pt(o)\\
&3 : \forall o \in pt(p).\ pt(o) \subseteq pt(x)\\
&4 : \forall o \in pt(q).\ pt(o) \subseteq pt(y)
\end{aligned}
$$

(b) Constraints.

$$
\begin{aligned}
&1 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_1}\\
&2 : \{o_3, o_4\}_r \subseteq \{\ \}_{o_2}\\
&3 : \{o_3, o_4\}_{o_1} \subseteq \{\ \}_x\\
&4 : \{o_3, o_4\}_{o_2} \subseteq \{\ \}_y
\end{aligned}
$$

(c) Operations.

$$
pt(p) = \{o_1\} \quad pt(q) = \{o_2\} \quad pt(r) = \{o_3, o_4\} \quad pt(x) = \{o_3, o_4\}
$$
$$
pt(y) = \{o_3, o_4\} \quad pt(o_1) = \{o_3, o_4\} \quad pt(o_2) = \{o_3, o_4\}
$$
$$
pt(o_3) = \{\ \} \quad pt(o_4) = \{\ \}
$$

(d) Result.

Motivating Example

Let us analyse the program fragment assuming $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, and $pt(r) = \{o_3, o_4\}$.

$$
\begin{aligned}
1 &: *p = r; \\
2 &: *q = r; \\
3 &: x = *p; \\
4 &: y = *q;
\end{aligned}
$$

(a) Program fragment.

$$
\begin{aligned}
1 &: \forall o \in pt(p).\ pt(r) \subseteq pt(o) \\
2 &: \forall o \in pt(q).\ pt(r) \subseteq pt(o) \\
3 &: \forall o \in pt(p).\ pt(o) \subseteq pt(x) \\
4 &: \forall o \in pt(q).\ pt(o) \subseteq pt(y)
\end{aligned}
$$

(b) Constraints.

$$
\begin{aligned}
1 &: \{o_3, o_4\}_r \subseteq \{\ \}_{o_1} \\
2 &: \{o_3, o_4\}_r \subseteq \{\ \}_{o_2} \\
3 &: \{o_3, o_4\}_{o_1} \subseteq \{\ \}_x \\
4 &: \{o_3, o_4\}_{o_2} \subseteq \{\ \}_y
\end{aligned}
$$

(c) Operations.

$$pt(p) = \{o_1\} \quad pt(q) = \{o_2\} \quad pt(r) = \{o_3, o_4\} \quad pt(x) = \{o_3, o_4\}$$

$$pt(y) = \{o_3, o_4\} \quad pt(o_1) = \{o_3, o_4\} \quad pt(o_2) = \{o_3, o_4\}$$

$$pt(o_3) = \{\ \} \quad pt(o_4) = \{\ \}$$

(d) Result.

Hash Consing

Hash Consing

- Functional programming technique exploiting immutable data structures.

Hash Consing

- Functional programming technique exploiting immutable data structures.
- Store each *unique* instance of data structure once.
  - Store *references* to the *unique* instance.

Hash Consing

- Functional programming technique exploiting immutable data structures.
- Store each *unique* instance of data structure once.
  - Store *references* to the *unique* instance.
- Example: string interning done by compilers/runtimes.

Hash Consing

- Functional programming technique exploiting immutable data structures.
- Store each *unique* instance of data structure once.
  - Store *references* to the *unique* instance.
- Example: string interning done by compilers/runtimes.
- Benefits: easy memoisation, easy comparisons, storing less, ...

Hash Consed Points-To Sets: Interning

Maintain a **global pool** mapping points-to sets to references.

$$\boxed{\begin{array}{c} \{\ldots\} \mapsto r_1 \\ \{\ldots\} \mapsto r_2 \\ \ldots \\ \{\ldots\} \mapsto r_n \end{array}}$$

Hash Consed Points-To Sets: Interning

Maintain a **global pool** mapping points-to sets to references.

$$\boxed{\begin{array}{c} \{\ldots\} \mapsto r_1 \\ \{\ldots\} \mapsto r_2 \\ \ldots \\ \{\ldots\} \mapsto r_n \end{array}}$$

New points-to set?

Hash Consed Points-To Sets: Interning

Maintain a **global pool** mapping points-to sets to references.

$$\boxed{\begin{array}{c} \{\ldots\} \mapsto r_1 \\ \{\ldots\} \mapsto r_2 \\ \ldots \\ \{\ldots\} \mapsto r_n \end{array}}$$

New points-to set?

Is the points-to set in the global pool?

Hash Consed Points-To Sets: Interning

Maintain a **global pool** mapping points-to sets to references.

$$\boxed{\begin{array}{c} \{\dots\} \mapsto r_1 \\ \{\dots\} \mapsto r_2 \\ \dots \\ \{\dots\} \mapsto r_n \end{array}}$$

New points-to set?

Is the points-to set in the global pool?

Yes $\Rightarrow$ Return corresponding reference.

Hash Consed Points-To Sets: Interning

Maintain a **global pool** mapping points-to sets to references.

$$\boxed{\begin{array}{c} \{\ldots\} \mapsto r_1 \\ \{\ldots\} \mapsto r_2 \\ \ldots \\ \{\ldots\} \mapsto r_n \end{array}}$$

New points-to set?

Is the points-to set in the global pool?

Yes $\Rightarrow$ Return corresponding reference.

No $\Rightarrow$ Add set into pool and return a reference to it.

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{r}
r_{x_1} \cup r_{x_2} \mapsto r_{y_1} \\
r_{x_3} \cup r_{x_4} \mapsto r_{y_2} \\
\dots \\
r_{x_{n-1}} \cup r_{x_n} \mapsto r_{y_m}
\end{array}}$$

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{c} \langle r_{x_1}, r_{x_2} \rangle \mapsto r_{y_1} \\ \langle r_{x_3}, r_{x_4} \rangle \mapsto r_{y_2} \\ \ldots \\ \langle r_{x_{n-1}}, r_{x_n} \rangle \mapsto r_{y_m} \end{array}}$$

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{r} \langle r_{x_1}, r_{x_2} \rangle \mapsto r_{y_1} \\ \langle r_{x_3}, r_{x_4} \rangle \mapsto r_{y_2} \\ \ldots \\ \langle r_{x_{n-1}}, r_{x_n} \rangle \mapsto r_{y_m} \end{array}}$$

Performing an operation on points-to sets (references)?

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{rcl} \langle r_{x_1}, r_{x_2} \rangle & \mapsto & r_{y_1} \\ \langle r_{x_3}, r_{x_4} \rangle & \mapsto & r_{y_2} \\ & \ldots & \\ \langle r_{x_{n-1}}, r_{x_n} \rangle & \mapsto & r_{y_m} \end{array}}$$

Performing an operation on points-to sets (references)?

Is the operation in the operations table?

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{r}
\langle r_{x_1}, r_{x_2}\rangle \mapsto r_{y_1} \\
\langle r_{x_3}, r_{x_4}\rangle \mapsto r_{y_2} \\
\ldots \\
\langle r_{x_{n-1}}, r_{x_n}\rangle \mapsto r_{y_m}
\end{array}}$$

Performing an operation on points-to sets (references)?

- Is the operation in the operations table?
  - Yes $\Rightarrow$ Return corresponding result (reference).

Hash Consed Points-To Sets: Memoisation

Maintain **operations tables** mapping operations to their result, i.e., cache operations.

$$\boxed{\begin{array}{r}
\langle r_{x_1}, r_{x_2}\rangle \mapsto r_{y_1} \\
\langle r_{x_3}, r_{x_4}\rangle \mapsto r_{y_2} \\
\ldots \\
\langle r_{x_{n-1}}, r_{x_n}\rangle \mapsto r_{y_m}
\end{array}}$$

Performing an operation on points-to sets (references)?

Is the operation in the operations table?

Yes $\Rightarrow$ Return corresponding result (reference).

No $\Rightarrow$ Perform operation on the concrete points-to set(s), intern the result, and map operand(s) to that.

Revisiting Our Example

$$\boxed{\begin{aligned} \{o_1\} &\mapsto r_1 \\ \{o_2\} &\mapsto r_2 \\ \{o_3, o_4\} &\mapsto r_3 \\ \{\ \} &\mapsto r_4 \end{aligned}}$$

(a) Global pool mapping points-to sets to references.

$$\boxed{\langle r_3, r_4\rangle \mapsto r_3}$$

(b) Union operations table.

$$pt_r(p) = r_1 \quad pt_r(q) = r_2 \quad pt_r(r) = r_3 \quad pt_r(x) = r_3 \quad pt_r(y) = r_3$$
$$pt_r(o_1) = r_3 \quad pt_r(o_2) = r_3 \quad pt_r(o_3) = r_4 \quad pt_r(o_4) = r_4$$

(c) Result.

Revisiting Our Example

$$\begin{array}{|r|}\hline \{o_1\} \mapsto r_1 \\ \{o_2\} \mapsto r_2 \\ \{o_3, o_4\} \mapsto r_3 \\ \{\ \} \mapsto r_4 \\ \hline \end{array}$$

(a) Global pool mapping points-to sets to references.

$$\boxed{\langle r_3, r_4 \rangle \mapsto r_3}$$

(b) Union operations table.

$$pt_r(p) = r_1 \quad pt_r(q) = r_2 \quad pt_r(r) = r_3 \quad pt_r(x) = r_3 \quad pt_r(y) = r_3$$
$$pt_r(o_1) = r_3 \quad pt_r(o_2) = r_3 \quad pt_r(o_3) = r_4 \quad pt_r(o_4) = r_4$$

(c) Result.

Revisiting Our Example

$$\begin{array}{r}
\{o_1\} \mapsto r_1 \\
\{o_2\} \mapsto r_2 \\
\{o_3, o_4\} \mapsto r_3 \\
\{\,\} \mapsto r_4
\end{array}$$

(a) Global pool mapping points-to sets to references.

$$\boxed{\langle r_3, r_4 \rangle \mapsto r_3}$$

(b) Union operations table.

$$pt_r(p) = r_1 \quad pt_r(q) = r_2 \quad pt_r(r) = r_3 \quad pt_r(x) = r_3 \quad pt_r(y) = r_3$$
$$pt_r(o_1) = r_3 \quad pt_r(o_2) = r_3 \quad pt_r(o_3) = r_4 \quad pt_r(o_4) = r_4$$

(c) Result.

Exploiting Set Properties

Exploiting Set Properties: Commutative Operations

- Performing $x \cup y$ and $y \cup x$ is wasteful; one suffices.

Exploiting Set Properties: Commutative Operations

- Performing $x \cup y$ and $y \cup x$ is wasteful; one suffices.
- Commutative operations should always be performed in some deterministic order.

Exploiting Set Properties: Commutative Operations

- Performing $x \cup y$ and $y \cup x$ is wasteful; one suffices.
- Commutative operations should always be performed in some deterministic order.
- $r_x < r_y$? Perform/lookup/cache $r_x \cup r_y$, otherwise $r_y \cup r_x$
  - Comparison is trivial due to hash consing.

Exploiting Set Properties: Property Operations

With hash-consed points-to sets, equality tests are trivial

Exploiting Set Properties: Property Operations

With hash-consed points-to sets, equality tests are trivial

Assume:
e is a reference to the empty points-to set and less than all other references.
r would be the (reference) result of any example operation.

Exploiting Set Properties: Property Operations

With hash-consed points-to sets, equality tests are trivial

Assume:
e is a reference to the empty points-to set and less than all other references.
r would be the (reference) result of any example operation.

Unions
Given an ordered union between references x and y ($x \cup y$),

$$x = e \Rightarrow r = y$$

$$x = y \Rightarrow r = x$$

Exploiting Set Properties: Property Operations

With hash-consed points-to sets, equality tests are trivial

Assume:
e is a reference to the empty points-to set and less than all other references.
r would be the (reference) result of any example operation.

Intersections
Given an ordered intersection between references x and y ($x \cap y$),

$$x = e \Rightarrow r = e$$

$$x = y \Rightarrow r = x$$

Exploiting Set Properties: Property Operations

With hash-consed points-to sets, equality tests are trivial

Assume:
e is a reference to the empty points-to set and less than all other references.
r would be the (reference) result of any example operation.

Differences
Given the difference between references x and y ($x - y$),

$$x = e \Rightarrow r = e$$

$$y = e \Rightarrow r = x$$

$$x = y \Rightarrow r = e$$

Exploiting Set Properties: Preemptive Memoisation

The result of one operation leads to the results of others.

Exploiting Set Properties: Preemptive Memoisation

The result of one operation leads to the results of others.

Unions

Given a union between references x and y with result r ($x \cup y = r$), we can infer and cache that,

$$x \cup r = r$$

$$x \cap r = x$$

$$y \cup r = r$$

$$y \cap r = y$$

Exploiting Set Properties: Preemptive Memoisation

The result of one operation leads to the results of others.

Intersections
Given an intersection between references x and y with result r ($x \cap y = r$), we can infer and cache that,

$$x \cap r = r$$

$$x \cup r = x$$

$$y \cap r = r$$

$$y \cup r = y$$

Exploiting Set Properties: Preemptive Memoisation

The result of one operation leads to the results of others.

Differences

Given a difference between references x and y with result r ($x - y = r$), we can infer and cache that,

$$x \cup r = x$$

$$x \cap r = r$$

$$y \cap r = e$$

$$y - r = y$$

$$r - y = r$$

Evaluation

Evaluation: Implementation

Evaluation: Implementation

- Implemented in LLVM-based points-to analysis framework SVF.
  - No algorithmic changes.

Evaluation: Implementation

- Implemented in LLVM-based points-to analysis framework SVF.
  - No algorithmic changes.
- **Flow-insensitive analysis**: field-sensitive Andersen's analysis with wave propagation with cycle detection.
- **Flow-sensitive analysis**: base field-sensitive staged flow-sensitive analysis.

Evaluation: Implementation

- Implemented in LLVM-based points-to analysis framework SVF.
  - No algorithmic changes.
- **Flow-insensitive analysis**: field-sensitive Andersen's analysis with wave propagation with cycle detection.
- **Flow-sensitive analysis**: base field-sensitive staged flow-sensitive analysis.
- Concrete points-to sets are represented with LLVM's sparse bit-vectors.
- References are 32-bit integers.

Evaluation: Flow-Insensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	4.52	0.30	3.58	0.28	1.26×	1.07×
nsd	9.32	0.55	7.23	0.51	1.29×	1.07×
tmux	18.86	0.59	14.11	0.56	1.34×	1.05×
gawk	19.92	0.64	14.15	0.58	1.41×	1.10×
bash	10.93	0.64	7.29	0.58	1.50×	1.11×
mutt	41.79	1.01	20.09	0.95	2.08×	1.06×
lynx	61.09	1.11	44.51	1.03	1.37×	1.08×
xpdf	179.52	1.94	111.80	1.88	1.61×	1.03×
python3	5509.52	4.13	1779.64	3.51	3.10×	1.18×
svn	5869.05	4.24	1829.20	2.82	3.21×	1.50×
emacs	5082.81	13.63	2651.32	13.05	1.92×	1.05×
git	5905.84	6.73	2499.55	6.79	2.36×	0.99×
kakoune	673.88	3.07	263.08	3.26	2.56×	0.94×
ruby	67.32	2.74	32.08	2.58	2.10×	1.06×
squid	2752.84	6.30	949.33	5.03	2.90×	1.25×
wireshark	271.60	6.42	211.42	6.21	1.28×	1.03×
Geo. Mean					1.85×	1.09×

Evaluation: Flow-Insensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	4.52	0.30	3.58	0.28	1.26×	1.07×
nsd	9.32	0.55	7.23	0.51	1.29×	1.07×
tmux	18.86	0.59	14.11	0.56	1.34×	1.05×
gawk	19.92	0.64	14.15	0.58	1.41×	1.10×
bash	10.93	0.64	7.29	0.58	1.50×	1.11×
mutt	41.79	1.01	20.09	0.95	2.08×	1.06×
lynx	61.09	1.11	44.51	1.03	1.37×	1.08×
xpdf	179.52	1.94	111.80	1.88	1.61×	1.03×
python3	5509.52	4.13	1779.64	3.51	3.10×	1.18×
svn	5869.05	4.24	1829.20	2.82	**3.21**×	1.50×
emacs	5082.81	13.63	2651.32	13.05	1.92×	1.05×
git	5905.84	6.73	2499.55	6.79	2.36×	0.99×
kakoune	673.88	3.07	263.08	3.26	2.56×	0.94×
ruby	67.32	2.74	32.08	2.58	2.10×	1.06×
squid	2752.84	6.30	949.33	5.03	2.90×	1.25×
wireshark	271.60	6.42	211.42	6.21	1.28×	1.03×
Geo. Mean					1.85×	1.09×

Evaluation: Flow-Insensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	4.52	0.30	3.58	0.28	1.26×	1.07×
nsd	9.32	0.55	7.23	0.51	1.29×	1.07×
tmux	18.86	0.59	14.11	0.56	1.34×	1.05×
gawk	19.92	0.64	14.15	0.58	1.41×	1.10×
bash	10.93	0.64	7.29	0.58	1.50×	1.11×
mutt	41.79	1.01	20.09	0.95	2.08×	1.06×
lynx	61.09	1.11	44.51	1.03	1.37×	1.08×
xpdf	179.52	1.94	111.80	1.88	1.61×	1.03×
python3	5509.52	4.13	1779.64	3.51	3.10×	1.18×
svn	5869.05	4.24	1829.20	2.82	3.21×	1.50×
emacs	5082.81	13.63	2651.32	13.05	1.92×	1.05×
git	5905.84	6.73	2499.55	6.79	2.36×	0.99×
kakoune	673.88	3.07	263.08	3.26	2.56×	0.94×
ruby	67.32	2.74	32.08	2.58	2.10×	1.06×
squid	2752.84	6.30	949.33	5.03	2.90×	1.25×
wireshark	271.60	6.42	211.42	6.21	1.28×	1.03×
Geo. Mean					**1.85×**	1.09×

Evaluation: Flow-Insensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	4.52	0.30	3.58	0.28	1.26×	1.07×
nsd	9.32	0.55	7.23	0.51	1.29×	1.07×
tmux	18.86	0.59	14.11	0.56	1.34×	1.05×
gawk	19.92	0.64	14.15	0.58	1.41×	1.10×
bash	10.93	0.64	7.29	0.58	1.50×	1.11×
mutt	41.79	1.01	20.09	0.95	2.08×	1.06×
lynx	61.09	1.11	44.51	1.03	1.37×	1.08×
xpdf	179.52	1.94	111.80	1.88	1.61×	1.03×
python3	5509.52	4.13	1779.64	3.51	3.10×	1.18×
svn	5869.05	4.24	1829.20	2.82	3.21×	1.50×
emacs	5082.81	13.63	2651.32	13.05	1.92×	1.05×
git	5905.84	6.73	2499.55	6.79	2.36×	**0.99**×
kakoune	673.88	3.07	263.08	3.26	2.56×	**0.94**×
ruby	67.32	2.74	32.08	2.58	2.10×	1.06×
squid	2752.84	6.30	949.33	5.03	2.90×	1.25×
wireshark	271.60	6.42	211.42	6.21	1.28×	1.03×
Geo. Mean					1.85×	1.09×

Evaluation: Flow-Insensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	4.52	0.30	3.58	0.28	1.26×	1.07×
nsd	9.32	0.55	7.23	0.51	1.29×	1.07×
tmux	18.86	0.59	14.11	0.56	1.34×	1.05×
gawk	19.92	0.64	14.15	0.58	1.41×	1.10×
bash	10.93	0.64	7.29	0.58	1.50×	1.11×
mutt	41.79	1.01	20.09	0.95	2.08×	1.06×
lynx	61.09	1.11	44.51	1.03	1.37×	1.08×
xpdf	179.52	1.94	111.80	1.88	1.61×	1.03×
python3	5509.52	4.13	1779.64	3.51	3.10×	1.18×
svn	5869.05	4.24	1829.20	2.82	3.21×	1.50×
emacs	5082.81	13.63	2651.32	13.05	1.92×	1.05×
git	5905.84	6.73	2499.55	6.79	2.36×	0.99×
kakoune	673.88	3.07	263.08	3.26	2.56×	0.94×
ruby	67.32	2.74	32.08	2.58	2.10×	1.06×
squid	2752.84	6.30	949.33	5.03	2.90×	1.25×
wireshark	271.60	6.42	211.42	6.21	1.28×	1.03×
Geo. Mean					1.85×	**1.09**×

Evaluation: Flow-Insensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	3766 (3.10%)	58 424 (48.14%)	59 185 (48.76%)	121 375
nsd	2900 (1.76%)	98 068 (59.45%)	64 001 (38.80%)	164 969
tmux	5651 (1.58%)	102 511 (28.58%)	250 552 (69.85%)	358 714
gawk	6378 (2.26%)	155 251 (55.09%)	120 172 (42.64%)	281 801
bash	1358 (0.93%)	126 307 (86.61%)	18 167 (12.46%)	145 832
mutt	8135 (3.07%)	145 604 (55.02%)	110 881 (41.90%)	264 620
lynx	10 750 (3.19%)	188 602 (56.04%)	137 205 (40.77%)	336 557
xpdf	29 622 (4.32%)	249 768 (36.41%)	406 582 (59.27%)	685 972
python3	33 274 (3.16%)	560 048 (53.25%)	458 319 (43.58%)	1 051 641
svn	22 879 (1.45%)	808 308 (51.13%)	749 564 (47.42%)	1 580 751
emacs	92 677 (4.61%)	809 938 (40.27%)	1 108 850 (55.13%)	2 011 465
git	124 333 (9.03%)	684 809 (49.73%)	567 897 (41.24%)	1 377 039
kakoune	86 364 (8.72%)	394 225 (39.81%)	509 693 (51.47%)	990 282
ruby	11 090 (3.15%)	195 495 (55.47%)	145 827 (41.38%)	352 412
squid	55 792 (3.23%)	796 024 (46.09%)	875 241 (50.68%)	1 727 057
wireshark	47 856 (3.02%)	592 647 (37.34%)	946 580 (59.64%)	1 587 083
Geo. Mean	– (2.98%)	– (48.40%)	– (44.24%)	

Evaluation: Flow-Insensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	3766 (3.10%)	58 424 (48.14%)	59 185 (48.76%)	121 375
nsd	2900 (1.76%)	98 068 (59.45%)	64 001 (38.80%)	164 969
tmux	5651 (1.58%)	102 511 (28.58%)	250 552 (69.85%)	358 714
gawk	6378 (2.26%)	155 251 (55.09%)	120 172 (42.64%)	281 801
bash	1358 (0.93%)	126 307 (86.61%)	18 167 (12.46%)	145 832
mutt	8135 (3.07%)	145 604 (55.02%)	110 881 (41.90%)	264 620
lynx	10 750 (3.19%)	188 602 (56.04%)	137 205 (40.77%)	336 557
xpdf	29 622 (4.32%)	249 768 (36.41%)	406 582 (59.27%)	685 972
python3	33 274 (3.16%)	560 048 (53.25%)	458 319 (43.58%)	1 051 641
svn	22 879 (1.45%)	808 308 (51.13%)	749 564 (47.42%)	1 580 751
emacs	92 677 (4.61%)	809 938 (40.27%)	1 108 850 (55.13%)	2 011 465
git	124 333 (**9.03%**)	684 809 (49.73%)	567 897 (41.24%)	1 377 039
kakoune	86 364 (8.72%)	394 225 (39.81%)	509 693 (51.47%)	990 282
ruby	11 090 (3.15%)	195 495 (55.47%)	145 827 (41.38%)	352 412
squid	55 792 (3.23%)	796 024 (46.09%)	875 241 (50.68%)	1 727 057
wireshark	47 856 (3.02%)	592 647 (37.34%)	946 580 (59.64%)	1 587 083
Geo. Mean	– (2.98%)	– (48.40%)	– (44.24%)	

Evaluation: Flow-Insensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	3766 (3.10%)	58 424 (48.14%)	59 185 (48.76%)	121 375
nsd	2900 (1.76%)	98 068 (59.45%)	64 001 (38.80%)	164 969
tmux	5651 (1.58%)	102 511 (28.58%)	250 552 (69.85%)	358 714
gawk	6378 (2.26%)	155 251 (55.09%)	120 172 (42.64%)	281 801
bash	1358 (0.93%)	126 307 (86.61%)	18 167 (12.46%)	145 832
mutt	8135 (3.07%)	145 604 (55.02%)	110 881 (41.90%)	264 620
lynx	10 750 (3.19%)	188 602 (56.04%)	137 205 (40.77%)	336 557
xpdf	29 622 (4.32%)	249 768 (36.41%)	406 582 (59.27%)	685 972
python3	33 274 (3.16%)	560 048 (53.25%)	458 319 (43.58%)	1 051 641
svn	22 879 (1.45%)	808 308 (51.13%)	749 564 (47.42%)	1 580 751
emacs	92 677 (4.61%)	809 938 (40.27%)	1 108 850 (55.13%)	2 011 465
git	124 333 (9.03%)	684 809 (49.73%)	567 897 (41.24%)	1 377 039
kakoune	86 364 (8.72%)	394 225 (39.81%)	509 693 (51.47%)	990 282
ruby	11 090 (3.15%)	195 495 (55.47%)	145 827 (41.38%)	352 412
squid	55 792 (3.23%)	796 024 (46.09%)	875 241 (50.68%)	1 727 057
wireshark	47 856 (3.02%)	592 647 (37.34%)	946 580 (59.64%)	1 587 083
Geo. Mean	– (2.98%)	– **(48.40%)**	– **(44.24%)**	

Evaluation: Flow-Sensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	77.27	1.08	73.04	0.66	1.06×	1.65×
nsd	113.39	2.97	75.76	0.74	1.50×	4.02×
tmux	280.09	3.33	212.25	1.14	1.32×	2.93×
gawk	1526.61	12.13	685.78	2.42	2.23×	5.02×
bash	337.01	8.55	165.28	1.51	2.04×	5.65×
mutt	797.92	13.95	400.08	2.15	1.99×	6.49×
lynx	3256.47	26.71	1594.90	3.65	2.04×	7.32×
xpdf	OOM	OOM	7210.36	6.44	–	≥15.52×
python3	OOM	OOM	23534.00	16.72	–	≥5.98×
svn	OOM	OOM	14000.10	22.61	–	≥4.42×
emacs	OOM	OOM	51367.00	44.50	–	≥2.25×
git	OOM	OOM	49264.50	39.59	–	≥2.53×
kakoune	OOM	OOM	12845.40	9.49	–	≥10.53×
ruby	OOM	OOM	4250.19	9.77	–	≥10.24×
squid	OOM	OOM	72733.50	37.53	–	≥2.66×
wireshark	OOM	OOM	24820.20	14.50	–	≥6.90×
Geo. Mean					1.69×	≥4.93×

Evaluation: Flow-Sensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	77.27	1.08	73.04	0.66	1.06×	1.65×
nsd	113.39	2.97	75.76	0.74	1.50×	4.02×
tmux	280.09	3.33	212.25	1.14	1.32×	2.93×
gawk	1526.61	12.13	685.78	2.42	2.23×	5.02×
bash	337.01	8.55	165.28	1.51	2.04×	5.65×
mutt	797.92	13.95	400.08	2.15	1.99×	6.49×
lynx	3256.47	26.71	1594.90	3.65	2.04×	7.32×
xpdf	OOM	OOM	7210.36	6.44	–	≥15.52×
python3	OOM	OOM	23534.00	16.72	–	≥5.98×
svn	OOM	OOM	14000.10	22.61	–	≥4.42×
emacs	OOM	OOM	51367.00	44.50	–	≥2.25×
git	OOM	OOM	49264.50	39.59	–	≥2.53×
kakoune	OOM	OOM	12845.40	9.49	–	≥10.53×
ruby	OOM	OOM	4250.19	9.77	–	≥10.24×
squid	OOM	OOM	72733.50	37.53	–	≥2.66×
wireshark	OOM	OOM	24820.20	14.50	–	≥6.90×
Geo. Mean					**1.69**×	≥4.93×

Evaluation: Flow-Sensitive

Program	Baseline		Hash consed		Time diff.	Memory diff.
	Time	Memory	Time	Memory		
dhcpcd	77.27	1.08	73.04	0.66	1.06×	1.65×
nsd	113.39	2.97	75.76	0.74	1.50×	4.02×
tmux	280.09	3.33	212.25	1.14	1.32×	2.93×
gawk	1526.61	12.13	685.78	2.42	2.23×	5.02×
bash	337.01	8.55	165.28	1.51	2.04×	5.65×
mutt	797.92	13.95	400.08	2.15	1.99×	6.49×
lynx	3256.47	26.71	1594.90	3.65	2.04×	7.32×
xpdf	OOM	OOM	7210.36	6.44	–	≥15.52×
python3	OOM	OOM	23534.00	16.72	–	≥5.98×
svn	OOM	OOM	14000.10	22.61	–	≥4.42×
emacs	OOM	OOM	51367.00	44.50	–	≥2.25×
git	OOM	OOM	49264.50	39.59	–	≥2.53×
kakoune	OOM	OOM	12845.40	9.49	–	≥10.53×
ruby	OOM	OOM	4250.19	9.77	–	≥10.24×
squid	OOM	OOM	72733.50	37.53	–	≥2.66×
wireshark	OOM	OOM	24820.20	14.50	–	≥6.90×
Geo. Mean					1.69×	≥**4.93**×

Evaluation: Flow-Sensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	858 019 (0.95%)	60 000 495 (66.20%)	29 772 284 (32.85%)	90 630 798
nsd	106 236 (0.06%)	131 385 659 (70.44%)	55 032 002 (29.50%)	186 523 897
tmux	265 726 (0.05%)	515 202 000 (89.33%)	61 282 554 (10.63%)	576 750 280
gawk	2 568 240 (0.11%)	1 674 478 472 (72.11%)	645 178 369 (27.78%)	2 322 225 081
bash	27 701 (0.01%)	435 565 965 (84.61%)	79 195 559 (15.38%)	514 789 225
mutt	319 829 (0.02%)	1 033 079 848 (78.53%)	282 194 806 (21.45%)	1 315 594 483
lynx	788 833 (0.02%)	3 836 871 871 (79.72%)	975 346 188 (20.26%)	4 813 006 892
xpdf	2 375 069 (0.02%)	9 475 061 599 (76.93%)	2 838 361 665 (23.05%)	12 315 798 333
python3	1 125 561 (0.00%)	27 494 110 299 (83.29%)	5 516 560 498 (16.71%)	33 011 796 358
svn	9 536 154 (0.04%)	15 950 564 702 (73.53%)	5 731 542 295 (26.42%)	21 691 643 151
emacs	40 525 287 (0.04%)	62 746 471 959 (67.68%)	29 925 669 621 (32.28%)	92 712 666 867
git	15 868 477 (0.03%)	36 002 062 086 (75.28%)	11 805 253 473 (24.69%)	47 823 184 036
kakoune	833 730 (0.00%)	21 708 874 709 (81.56%)	4 907 142 103 (18.44%)	26 616 850 542
ruby	1 219 328 (0.01%)	11 142 763 302 (83.49%)	2 202 254 328 (16.50%)	13 346 236 958
squid	3 080 598 (0.00%)	117 192 828 125 (85.99%)	19 097 056 263 (14.01%)	136 292 964 986
wireshark	9 219 867 (0.06%)	7 534 330 653 (50.97%)	7 237 949 555 (48.97%)	14 781 500 075
Geo. Mean	– (0.02%)	– (75.61%)	– (22.10%)	

Evaluation: Flow-Sensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	858 019 (**0.95%**)	60 000 495 (66.20%)	29 772 284 (32.85%)	90 630 798
nsd	106 236 (0.06%)	131 385 659 (70.44%)	55 032 002 (29.50%)	186 523 897
tmux	265 726 (0.05%)	515 202 000 (89.33%)	61 282 554 (10.63%)	576 750 280
gawk	2 568 240 (0.11%)	1 674 478 472 (72.11%)	645 178 369 (27.78%)	2 322 225 081
bash	27 701 (0.01%)	435 565 965 (84.61%)	79 195 559 (15.38%)	514 789 225
mutt	319 829 (0.02%)	1 033 079 848 (78.53%)	282 194 806 (21.45%)	1 315 594 483
lynx	788 833 (0.02%)	3 836 871 871 (79.72%)	975 346 188 (20.26%)	4 813 006 892
xpdf	2 375 069 (0.02%)	9 475 061 599 (76.93%)	2 838 361 665 (23.05%)	12 315 798 333
python3	1 125 561 (0.00%)	27 494 110 299 (83.29%)	5 516 560 498 (16.71%)	33 011 796 358
svn	9 536 154 (0.04%)	15 950 564 702 (73.53%)	5 731 542 295 (26.42%)	21 691 643 151
emacs	40 525 287 (0.04%)	62 746 471 959 (67.68%)	29 925 669 621 (32.28%)	92 712 666 867
git	15 868 477 (0.03%)	36 002 062 086 (75.28%)	11 805 253 473 (24.69%)	47 823 184 036
kakoune	833 730 (0.00%)	21 708 874 709 (81.56%)	4 907 142 103 (18.44%)	26 616 850 542
ruby	1 219 328 (0.01%)	11 142 763 302 (83.49%)	2 202 254 328 (16.50%)	13 346 236 958
squid	3 080 598 (0.00%)	117 192 828 125 (85.99%)	19 097 056 263 (14.01%)	136 292 964 986
wireshark	9 219 867 (0.06%)	7 534 330 653 (50.97%)	7 237 949 555 (48.97%)	14 781 500 075
Geo. Mean	– (0.02%)	– (75.61%)	– (22.10%)	

Evaluation: Flow-Sensitive Unions

Program	Concrete	Property	Lookup	Total
dhcpcd	858 019 (0.95%)	60 000 495 (66.20%)	29 772 284 (32.85%)	90 630 798
nsd	106 236 (0.06%)	131 385 659 (70.44%)	55 032 002 (29.50%)	186 523 897
tmux	265 726 (0.05%)	515 202 000 (89.33%)	61 282 554 (10.63%)	576 750 280
gawk	2 568 240 (0.11%)	1 674 478 472 (72.11%)	645 178 369 (27.78%)	2 322 225 081
bash	27 701 (0.01%)	435 565 965 (84.61%)	79 195 559 (15.38%)	514 789 225
mutt	319 829 (0.02%)	1 033 079 848 (78.53%)	282 194 806 (21.45%)	1 315 594 483
lynx	788 833 (0.02%)	3 836 871 871 (79.72%)	975 346 188 (20.26%)	4 813 006 892
xpdf	2 375 069 (0.02%)	9 475 061 599 (76.93%)	2 838 361 665 (23.05%)	12 315 798 333
python3	1 125 561 (0.00%)	27 494 110 299 (83.29%)	5 516 560 498 (16.71%)	33 011 796 358
svn	9 536 154 (0.04%)	15 950 564 702 (73.53%)	5 731 542 295 (26.42%)	21 691 643 151
emacs	40 525 287 (0.04%)	62 746 471 959 (67.68%)	29 925 669 621 (32.28%)	92 712 666 867
git	15 868 477 (0.03%)	36 002 062 086 (75.28%)	11 805 253 473 (24.69%)	47 823 184 036
kakoune	833 730 (0.00%)	21 708 874 709 (81.56%)	4 907 142 103 (18.44%)	26 616 850 542
ruby	1 219 328 (0.01%)	11 142 763 302 (83.49%)	2 202 254 328 (16.50%)	13 346 236 958
squid	3 080 598 (0.00%)	117 192 828 125 (85.99%)	19 097 056 263 (14.01%)	136 292 964 986
wireshark	9 219 867 (0.06%)	7 534 330 653 (50.97%)	7 237 949 555 (48.97%)	14 781 500 075
Geo. Mean	– (0.02%)	– **(75.61%)**	– (22.10%)	

Thank you

Implementation available at
https://github.com/SVF-tools/SVF/wiki/Hash-Consed-Points-To-Sets