Sparse Flow-Sensitive Pointer Analysis For Multithreaded Programs

Yulei Sui, Peng Di and Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
2052 Sydney Australia

March 15, 2016

Contributions

- The first sparse flow-sensitive pointer analysis for unstructured multithreaded programs (C with Pthread)
- A series of static thread interference analyses by reasoning about fork/join, memory accesses, lock/unlock to generate value-flows among threads.
- Significantly faster than non-sparse algorithm and scales to large size multithreaded Pthread programs with up to 100KLOC.

Outline

- Background and Motivation
- Our approach: FSAM
- Evalution

Pointer Analysis

Pointer Analysis is to statically approximate runtime values of a pointer

Pointer Analysis

Pointer Analysis is to statically approximate runtime values of a pointer

A fundamental enabling technology for many other program analyses and optimisations.

- Compiler optimisations (e.g., Auto-Vectorization)
- Memory errors (e.g., Null pointer and use-after-free)
- Concurrency bugs (e.g., Data race, dead lock detection)
- Security (e.g., Control-flow integrity enforcement)
- Accelerating dynamic analysis (e.g., MemSan, TSan)
- ...

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-Insensitive Pointer Analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-Insensitive Pointer Analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Sensitive Pointer Analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-Insensitive Pointer Analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Sensitive Pointer Analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

p = & a

*p = & b

*p = & c

q = *p

Flow-Insensitive Analysis

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-Insensitive Pointer Analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Sensitive Pointer Analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

```
p = & a
*p = & b
*p = & c
q = *p
```

p → a

a → b, c

q → b, c

Flow-Insensitive Analysis

Flow-Insensitive v.s. Flow-Sensitive Analysis

Flow-Insensitive Pointer Analysis:

- **Ignore program execution order**
- **A single solution across whole program**

Flow-Sensitive Pointer Analysis:

- **Respect program control-flow**
- **A separate solution at each program point**

Flow-Insensitive Analysis		Flow-sensitive Analysis	
p = & a	p → a	p = & a	p → a
*p = & b	a → b, c	*p = & b	p → a a → b
*p = & c	q → b, c	*p = & c	p → a a → c
q = *p		q = *p	p → a a → c q → c

Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**

...

p → a   x → m

*p = & b

p → a   a → b   x → m

*p = & c

p → a   a → c   x → m

*x = & d

p → a   a → c   x → m   m → d

y = *x

p → a   a → c   x → m   m → d   y → d

q = *p

p → a   a → c   x → m   m → d   y → d   q → c

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**

...

p → a   x → m

*p = & b

p → a   a → b   x → m

*p = & c

p → a   a → c   x → m

*x = & d

p → a   a → c   x → m   m → d

y = *x

p → a   a → c   x → m   m → d   y → d

q = *p

p → a   a → c   x → m   m → d   y → d   q → c

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**

...

p → a   x → m

*p = & b

p → a   a → b   ~~x → m~~

*p = & c

p → a   a → c   ~~x → m~~

*x = & d

~~p → a~~   ~~a → c~~   x → m   m → d

y = *x

~~p → a~~   ~~a → c~~   x → m   m → d   y → d

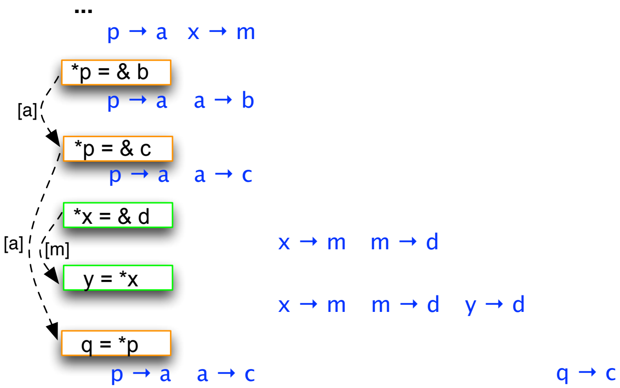
q = *p

p → a   a → c   ~~x → m~~   ~~m → d~~   ~~y → d~~   q → c

Data-flow-based flow-sensitive analysis

Sparse Flow-Sensitive Analysis

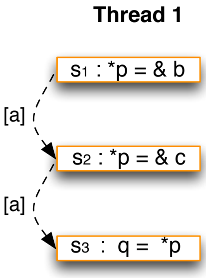
- **Propagate points-to information only along pre-computed def-use chains instead of control-flow**



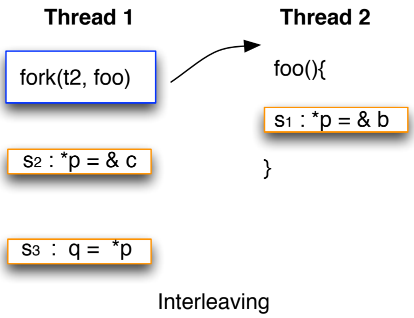
Sparse flow-sensitive analysis

(*Hardekopf and Lin. - CGO'11*) (*Ye, Sui and Xue. - SAS '14*)

Flow-Sensitivity Under Thread Interleaving

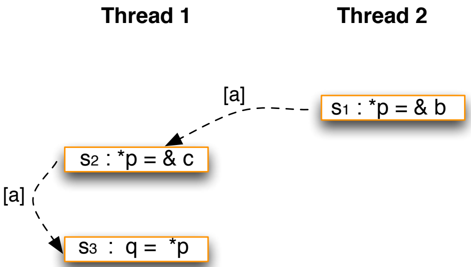


Flow-Sensitivity Under Thread Interleaving



Flow-Sensitivity Under Thread Interleaving

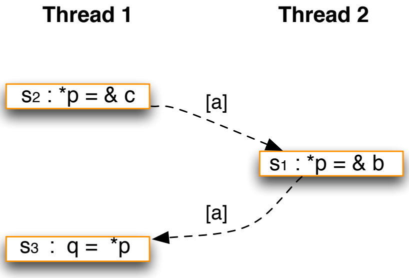
Scenario 1:



execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

Flow-Sensitivity Under Thread Interleaving

Scenario 2:

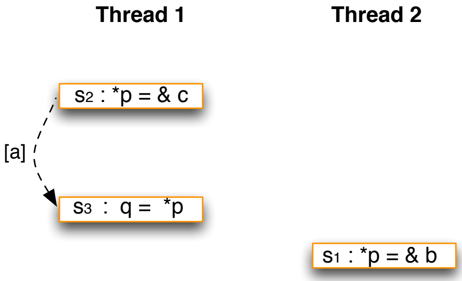


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

execution sequence : s_2, s_1, s_3 points-to of q : pt(q) = {b}

Flow-Sensitivity Under Thread Interleaving

Scenario 3:

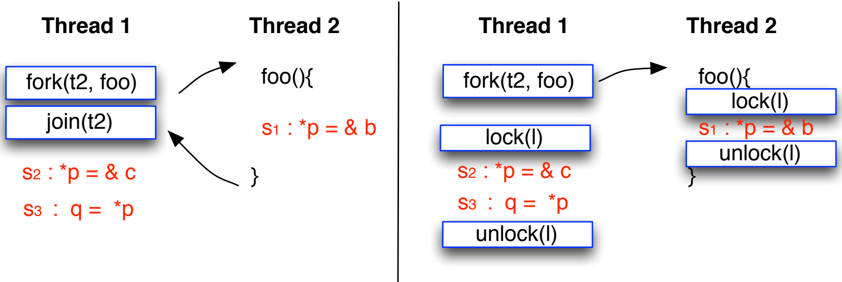


execution sequence : s_1, s_2, s_3 points-to of q : pt(q) = {c}

execution sequence : s_2, s_1, s_3 points-to of q : pt(q) = {b}

execution sequence : s_2, s_3, s_1 points-to of q : pt(q) = {c}

Flow-Sensitivity Under Thread Interleaving



(a) non-interference via join

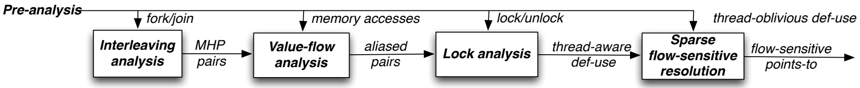
(b) non-interference via lock/unlock

points-to of q: pt(q) = {c}

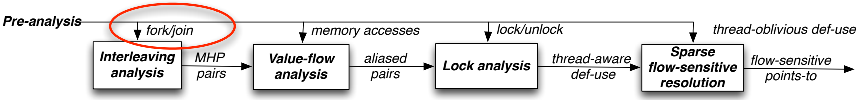
Outline

- Background and Motivation
- Our approach: FSAM
- Evalution

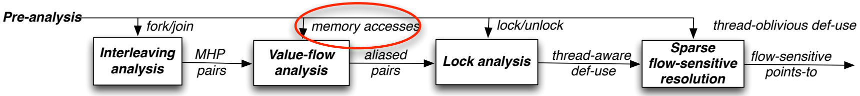
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs



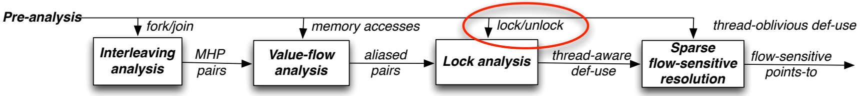
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs



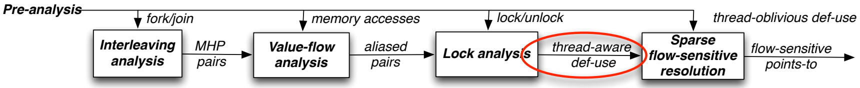
FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs



FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs



FSAM: Sparse Flow-Sensitive Analysis For Multithreaded Programs



Context-Sensitive Abstract Threads

An abstract thread *t* refers to a call of pthread_create() at a context-sensitive fork site during the analysis.

```
void main(){            void foo(){

 cs1: foo();           cs3: fork(t1, bar);
 cs2: foo();              }

}
```

t1 *refers to fork site under context [1,3]*

t1' *refers to fork site under context [2,3]*

t1 and t1' are context-sensitive threads

Context-Sensitive Abstract Threads

An abstract thread t refers to a call of pthread_create() at a context-sensitive fork site during the analysis.

```
void main(){

cs1:  foo();
cs2:  foo();

}
```

```
void foo(){

cs3: fork(t1, bar);
}
```

t1 refers to fork site under context [1,3]

t1' refers to fork site under context [2,3]

t1 and t1' are context-sensitive threads

```
void main(){

   for(i=0;i<10;i++){
      fork(t[i], foo)
   }

}
```

t is multi-forked thread

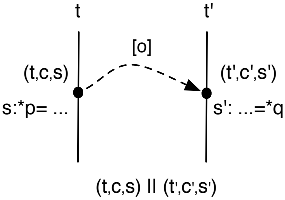
A thread t always refers to a context-sensitive fork site, i.e., a unique runtime thread unless $t \in \mathcal{M}$ is *multi-forked*, in which case, t may represent more than one runtime thread.

Thread-Aware Value-Flows

A thread-aware def-use is added if a pair of statements (t, c, s) and (t', c', s')

- (1) may access same memory using pre-computed results.
- (2) may happen in parallel

$$\frac{s : *p = \_ \qquad s' : \_ = *q \ \text{ or } \ *q = \_ \qquad (t, c, s) \parallel (t', c', s') \qquad o \in Alias(*p, *q)}{s \overset{o}{\hookrightarrow} s'}$$



UNSW THE UNIVERSITY OF NEW SOUTH WALES

Context-sensitive Thread Interleaving Analysis

$(t_1, c_1, s_1) \parallel (t_2, c_2, s_2)$ holds if:

$$\begin{cases} t_2 \in \mathcal{I}(t_1, c_1, s_1) \land t_1 \in \mathcal{I}(t_2, c_2, s_2) & \text{if } t_1 \neq t_2 \\ t_1 \in \mathcal{M} & \text{otherwise} \end{cases}$$

where $\mathcal{I}(t, c, s)$: denotes a set of interleaved threads may run in parallel with s in thread t under calling context c,
$\mathcal{M}$ is the set of multi-forked threads.

Interleaving Analysis

Computing $\mathcal{I}(t, c, s)$ is formalized as a forward data-flow problem $(V, \sqcap, F)$.

- V: the set of all thread interleaving facts.
- $\sqcap$: meet operator ($\cup$).
- F: $V \rightarrow V$ transfer functions associated with each node in an ICFG.

Interleaving Analysis Rule

$$\text{[I-DESCENDANT]}\quad \frac{t \xrightarrow{(c, fk_i)} t' \quad (t, c, fk_i) \to (t, c, \ell) \quad (c', \ell') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t, c, \ell) \quad \{t\} \subseteq \mathcal{I}(t', c', \ell')}$$

$$\text{[I-SIBLING]}\quad \frac{t \bowtie t' \quad (c, \ell) = Entry(\mathcal{S}_t) \quad (c', \ell') = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t', c', \ell') \quad \{t'\} \subseteq \mathcal{I}(t, c, \ell)}$$

$$\text{[I-JOIN]}\quad \frac{t \xleftarrow{(c, jn_i)} t'}{\mathcal{I}(t, c, jn_i) = \mathcal{I}(t, c, jn_i) \backslash \{t'\}}$$

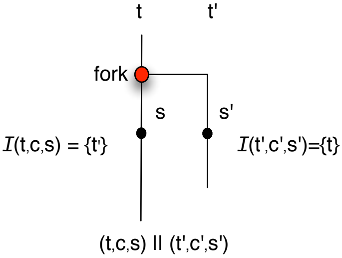
$$\text{[I-CALL]}\quad \frac{(t, c, \ell) \xrightarrow{call_i} (t, c', \ell') \quad c' = c.push(i)}{\mathcal{I}(t, c, \ell) \subseteq \mathcal{I}(t, c', \ell')}$$

$$\text{[I-INTRA]}\quad \frac{(t, c, \ell) \to (t, c, \ell')}{\mathcal{I}(t, c, \ell) \subseteq \mathcal{I}(t, c, \ell')}$$

$$\text{[I-RET]}\quad \frac{(t, c, \ell) \xrightarrow{ret_i} (t, c', \ell') \quad i = c.peek() \quad c' = c.pop()}{\mathcal{I}(t, c, \ell) \subseteq \mathcal{I}(t, c', \ell')}$$

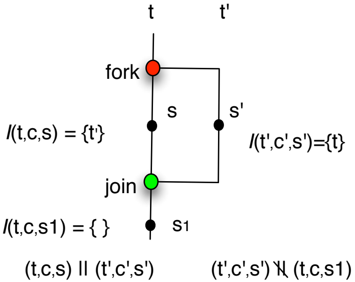
Interleaving Analysis Rule

$$[\texttt{I-DESCENDANT}] \quad \frac{t \xrightarrow{(c, fk_i)} t' \quad (t, c, fk_i) \rightarrow (t, c, \ell) \quad (c', \ell') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t, c, \ell) \quad \{t\} \subseteq \mathcal{I}(t', c', \ell')}$$



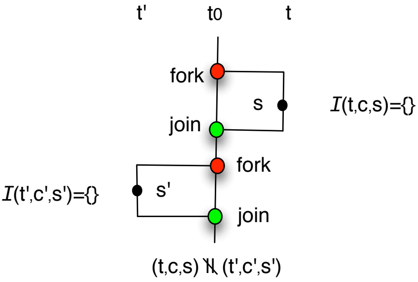
Interleaving Analysis Rule

$$[\texttt{I-JOIN}] \quad \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}}$$



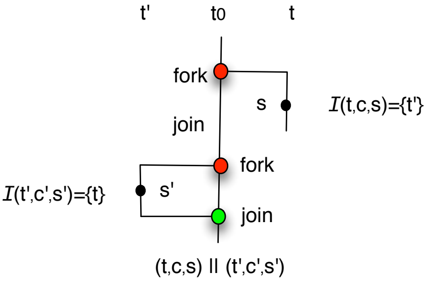
Interleaving Analysis Rule

$$[\texttt{I-SIBLING}]\quad \frac{t \bowtie t' \quad (c,\ell) = Entry(\mathcal{S}_t) \quad (c',\ell') = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',\ell') \quad \{t'\} \subseteq \mathcal{I}(t,c,\ell)}$$



Interleaving Analysis Rule

$$[\texttt{I-SIBLING}] \quad \frac{t \bowtie t' \quad (c, \ell) = \mathit{Entry}(\mathcal{S}_t) \quad (c', \ell') = \mathit{Entry}(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t', c', \ell') \quad \{t'\} \subseteq \mathcal{I}(t, c, \ell)}$$



Interleaving Analysis Rule

$$\texttt{[I-DESCENDANT]}\ \frac{t \xrightarrow{(c,fk_i)} t' \quad (t,c,fk_i) \to (t,c,\ell) \quad (c',\ell') = Entry(\mathcal{S}_{t'})}{\{t'\} \subseteq \mathcal{I}(t,c,\ell) \quad \{t\} \subseteq \mathcal{I}(t',c',\ell')}$$

$$\texttt{[I-SIBLING]}\ \frac{t \bowtie t' \quad (c,\ell) = Entry(\mathcal{S}_t) \quad (c',\ell') = Entry(\mathcal{S}_{t'}) \quad t \not\succ t' \wedge t' \not\succ t}{\{t\} \subseteq \mathcal{I}(t',c',\ell') \quad \{t'\} \subseteq \mathcal{I}(t,c,\ell)}$$

$$\texttt{[I-JOIN]}\ \frac{t \xleftarrow{(c,jn_i)} t'}{\mathcal{I}(t,c,jn_i) = \mathcal{I}(t,c,jn_i) \backslash \{t'\}} \qquad \texttt{[I-CALL]}\ \frac{(t,c,\ell) \xrightarrow{call_i} (t,c',\ell') \quad c' = c.push(i)}{\mathcal{I}(t,c,\ell) \subseteq \mathcal{I}(t,c',\ell')}$$

$$\texttt{[I-INTRA]}\ \frac{(t,c,\ell) \to (t,c,\ell')}{\mathcal{I}(t,c,\ell) \subseteq \mathcal{I}(t,c,\ell')} \qquad \texttt{[I-RET]}\ \frac{(t,c,\ell) \xrightarrow{ret_i} (t,c',\ell') \quad i = c.peek() \quad c' = c.pop()}{\mathcal{I}(t,c,\ell) \subseteq \mathcal{I}(t,c',\ell')}$$

Lock Analysis

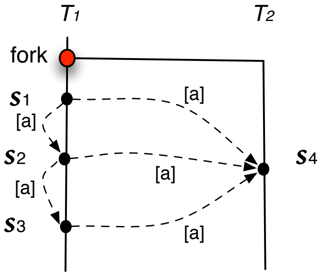
Statements from different mutex regions are interference-free if these regions are protected by a common lock.

Thread 1

```
main(){
fork(t2, foo)
s1 : *p = & c
s2 : *p = & d
s3 : *p = & e
}
```

Thread 2

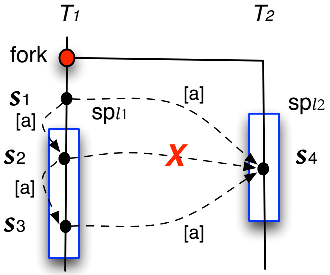
```
foo(){
s4 : q = *p
}
```



Lock Analysis

Statements from different mutex regions are interference-free if these regions are protected by a common lock.

Thread 1	Thread 2
main(){	foo(){
fork(t2, foo)	lock(l)
s_1 : *p = & c	s_4 : q = *p
lock(l)	unlock(l)
s_2 : *p = & d	}
s_3 : *p = & e	
unlock(l)	
}	



Outline

- Background and Motivation
- Our approach: FSAM
- Evalution

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.

[1] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.
- Methodology
  - FSAM v.s. NONSPARSE iterative flow-sensitive analysis following RR algorithm[1]

[1] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/) (CC '16)
  - Around 4,000 LOC core source code
  - Field-sensitivity: each field instance of a struct is treated as a separate object, arrays are considered monolithic.
  - On-the-fly call graph construction.
- Methodology
  - FSAM v.s. NONSPARSE iterative flow-sensitive analysis following RR algorithm[1]
- Benchmarks:
  - Two largest C benchmarks from `Phoenix-2.0`
  - Five largest C benchmarks from `Parsec-3.0`
  - Three open-source applications
- Machine setup:
  - Ubuntu Linux 3.11 Intel Xeon Quad Core, 3.7GHZ, 64GB

[1] *Radu Rugina and Martin Rinard*, Pointer Analysis for Multithreaded Programs PLDI '99

Benchmarks

Table: Program statistics.

Benchmark	Description	LOC
word_count	Word counter based on map-reduce	6330
kmeans	Iterative clustering of 3-D points	6008
radiosity	Graphics	12781
automount	Manage autofs mount points	13170
ferret	Content similarity search server	15735
bodytrack	Body tracking of a person	19063
httpd_server	Http server	52616
mt_daapd	Multi-threaded DAAP Daemon	57102
raytrace	Real-time raytracing	84373
x264	Media processing	113481
Total		380,659

RR only evaluated their analysis with benchmarks with up to 4500 lines of Cilk code.

Analysis Time and Memory Usage

Table: Analysis time and memory usage.

Program	Time (Secs)		Memory (MB)	
	FSAM	NONSPARSE	FSAM	NONSPARSE
word_count	3.04	17.40	13.79	53.76
kmeans	2.50	18.19	18.27	53.19
radiosity	6.77	29.29	38.65	95.00
automount	8.66	83.82	27.56	364.67
ferret	13.49	87.10	52.14	934.57
bodytrack	128.80	2809.89	313.66	12410.16
httpd_server	191.22	2079.43	55.78	6578.46
mt_daapd	90.67	2667.55	37.92	3403.26
raytrace	284.61	OOT	135.06	OOT
x264	531.55	OOT	129.58	OOT

FSAM is 12x faster and uses 28x less memory.

Impact of FSAM's three thread interference analysis

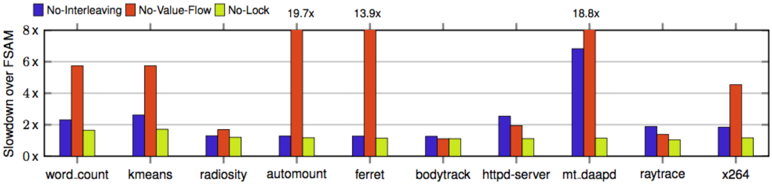


Figure: Impact of FSAM's three thread interference analysis phases on the performance of flow-sensitive points-to resolution.

Conclusion

- The first sparse flow-sensitive pointer analysis for unstructured multithreaded programs (C with Pthread)
- A series of context-sensitive thread interference analyses by reasoning about fork/join, memory accesses, lock/unlock.
- Significantly faster than non-sparse algorithm and scales to large size multithreaded Pthread programs with up to 100KLOC.

Open source and publicly available online:
`http://www.cse.unsw.edu.au/~corg/fsam/`

Thanks!

Q & A