Accelerating Dynamic Data Race Detection Using Static Thread Interference Analysis

Peng Di and Yulei Sui

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

March 12-13, 2016

UNSW THE UNIVERSITY OF NEW SOUTH WALES

Outline

- Motivation
- Analysis phases
- Evalution

Dynamic Data Race Detector: ThreadSanitizer

ThreadSanitizer (TSan) finds data races in multithreaded programs by inserting instrumentations at compile-time to perform runtime checks for all memory accesses.

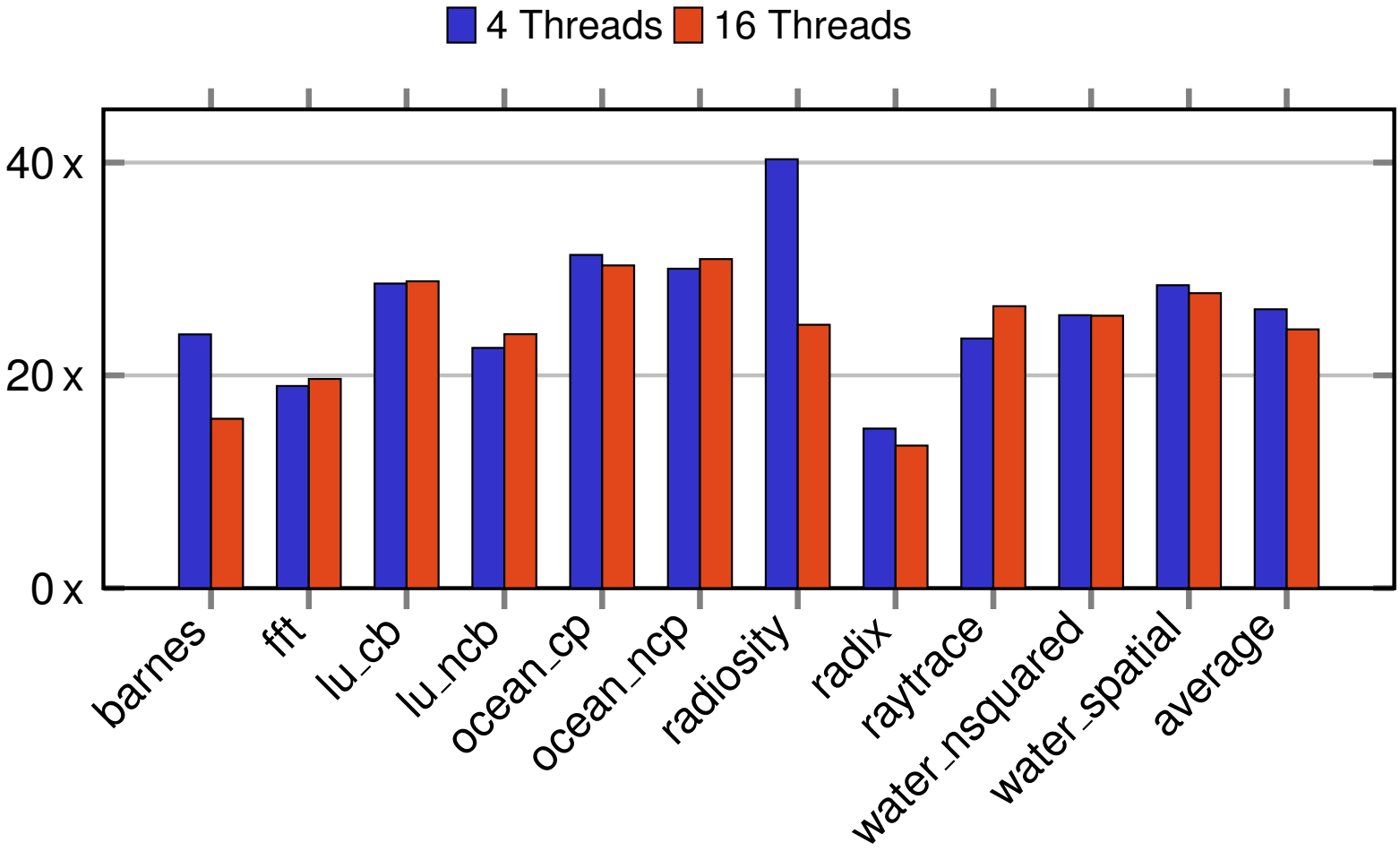
```
void foo(int *p) {
  *p = 42;
}
```

Orignial C code

```
void foo(int *p) {
  __tsan_func_entry(__builtin_return_address(0));
  __tsan_write4(p);
  *p = 42;
  __tsan_func_exit();
}
```

Code after instrumentation

TSan performance slowdown over native code for SPLASH2 benchmarks (under compiler option -O0)

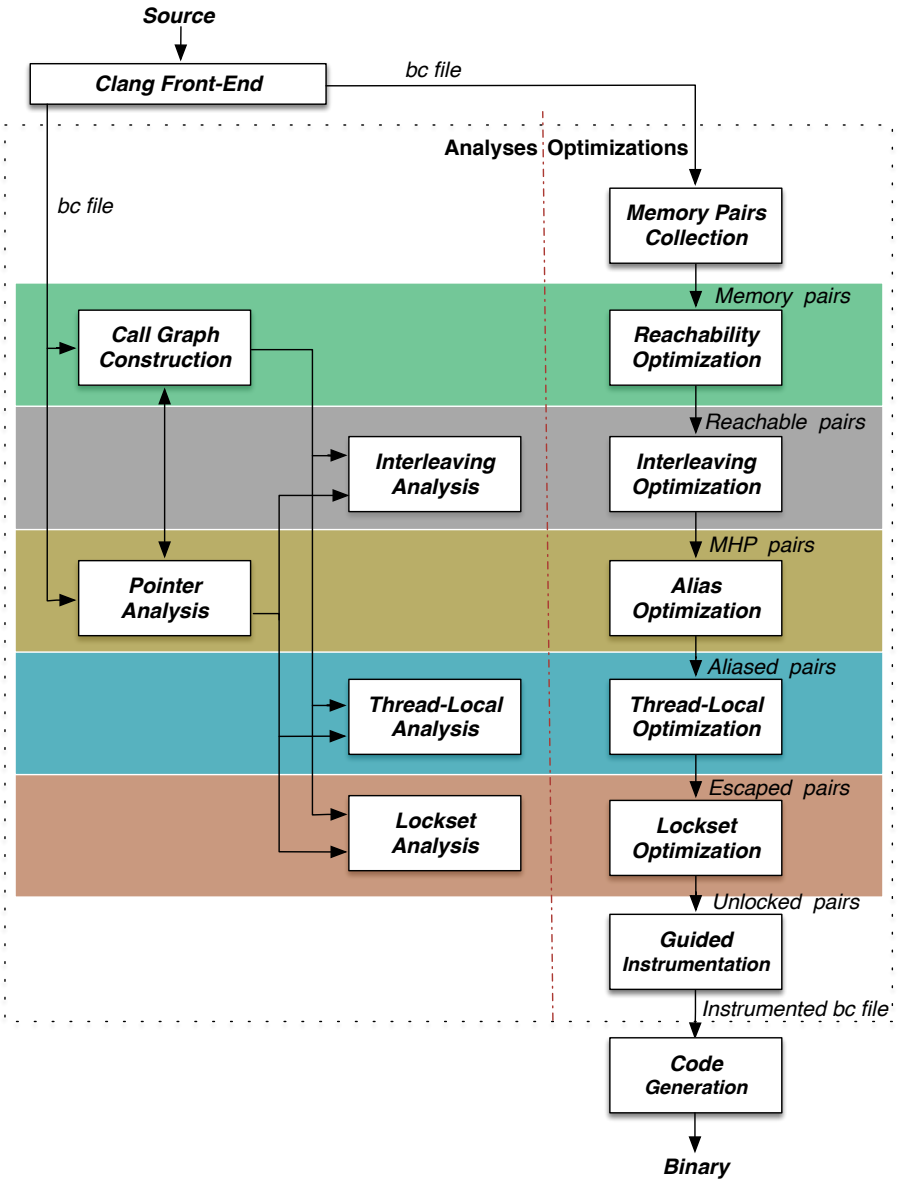


Machine: Ubuntu Linux 3.11.0-15-generic Intel Xeon Quad Core HT, 3.7GHZ, 64GB

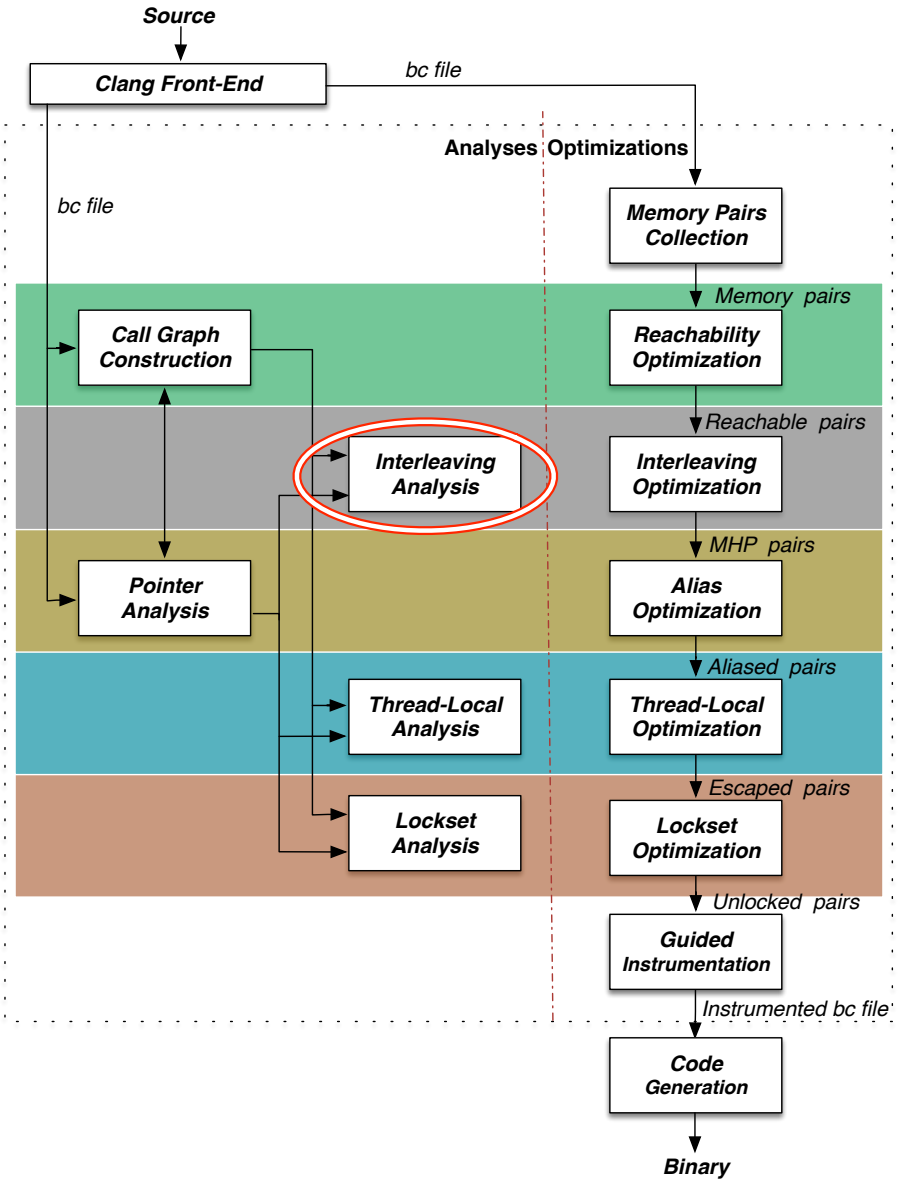
Outline

- Motivation
- Analysis phases
- Evalution

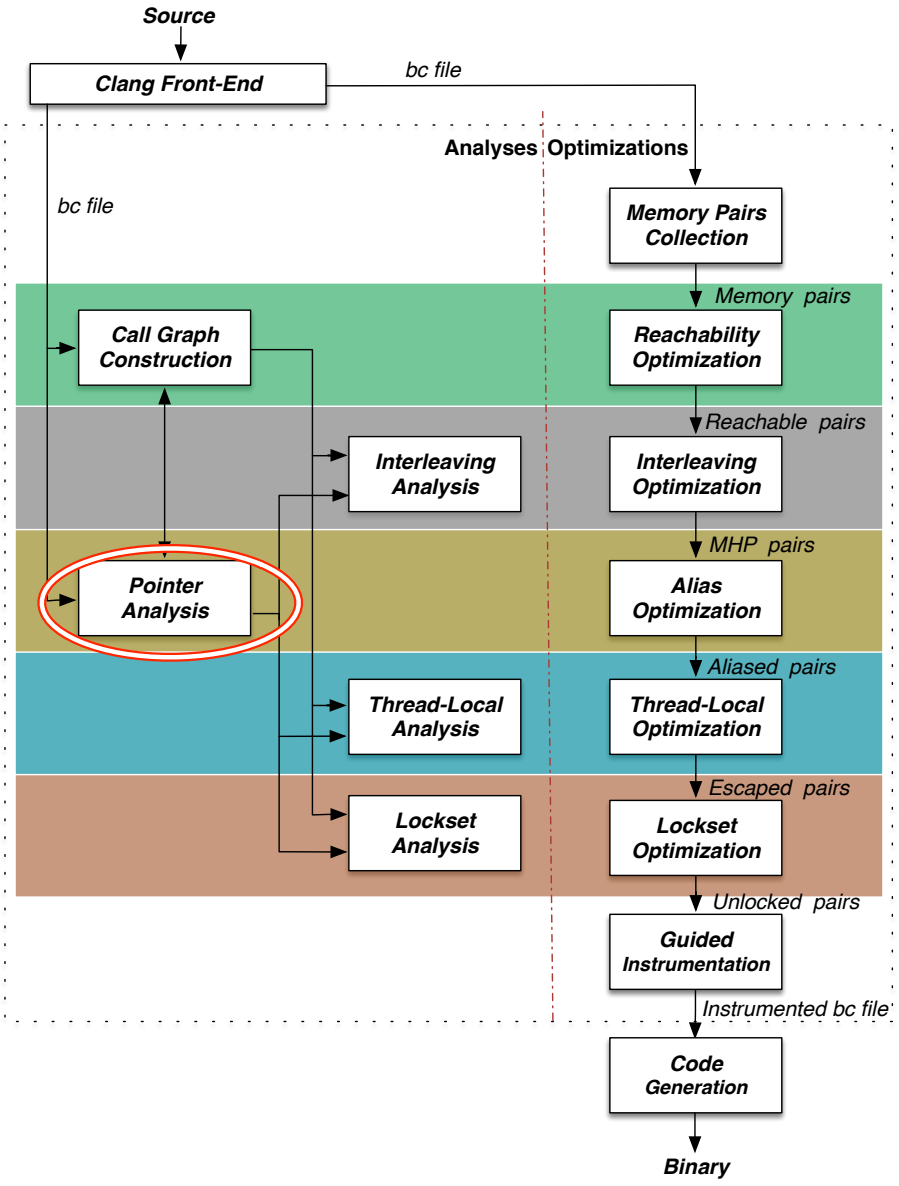
Framework



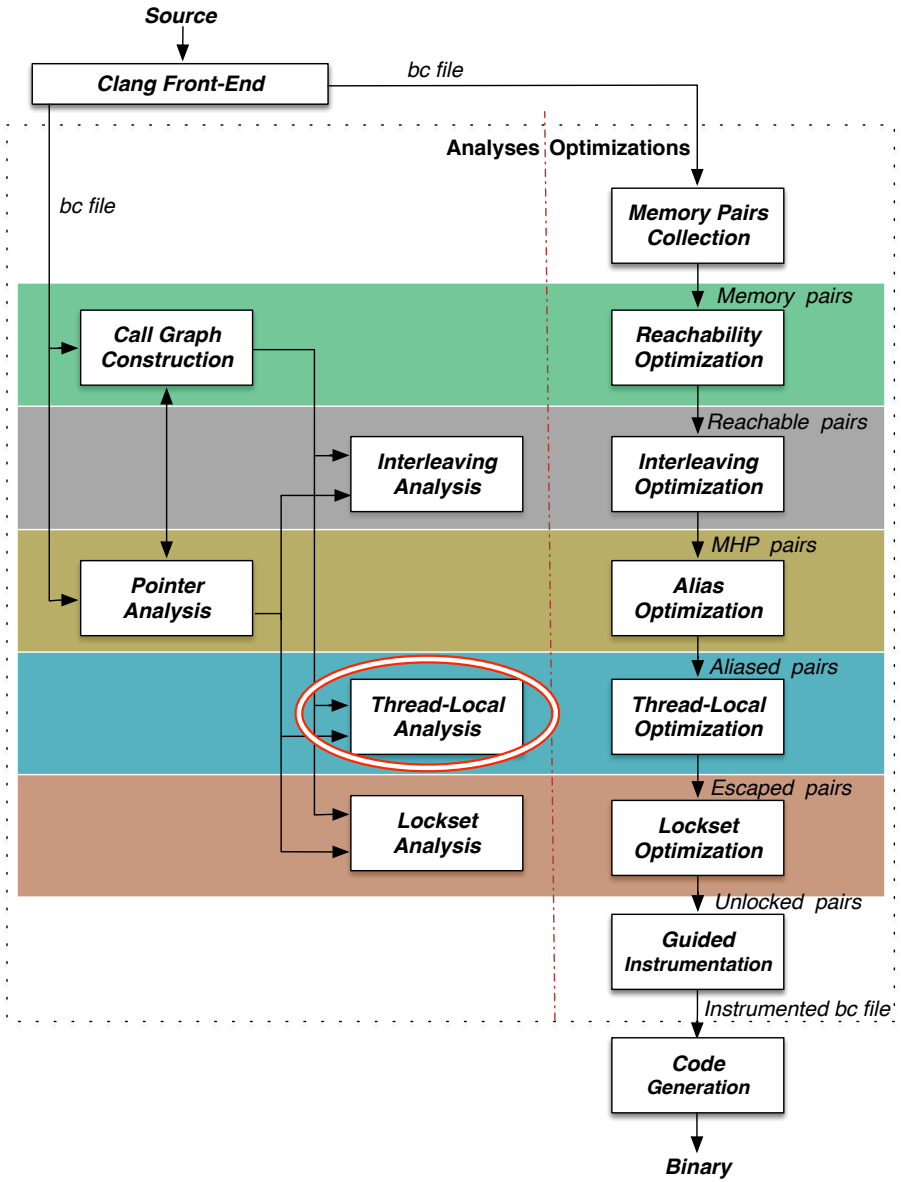
Framework



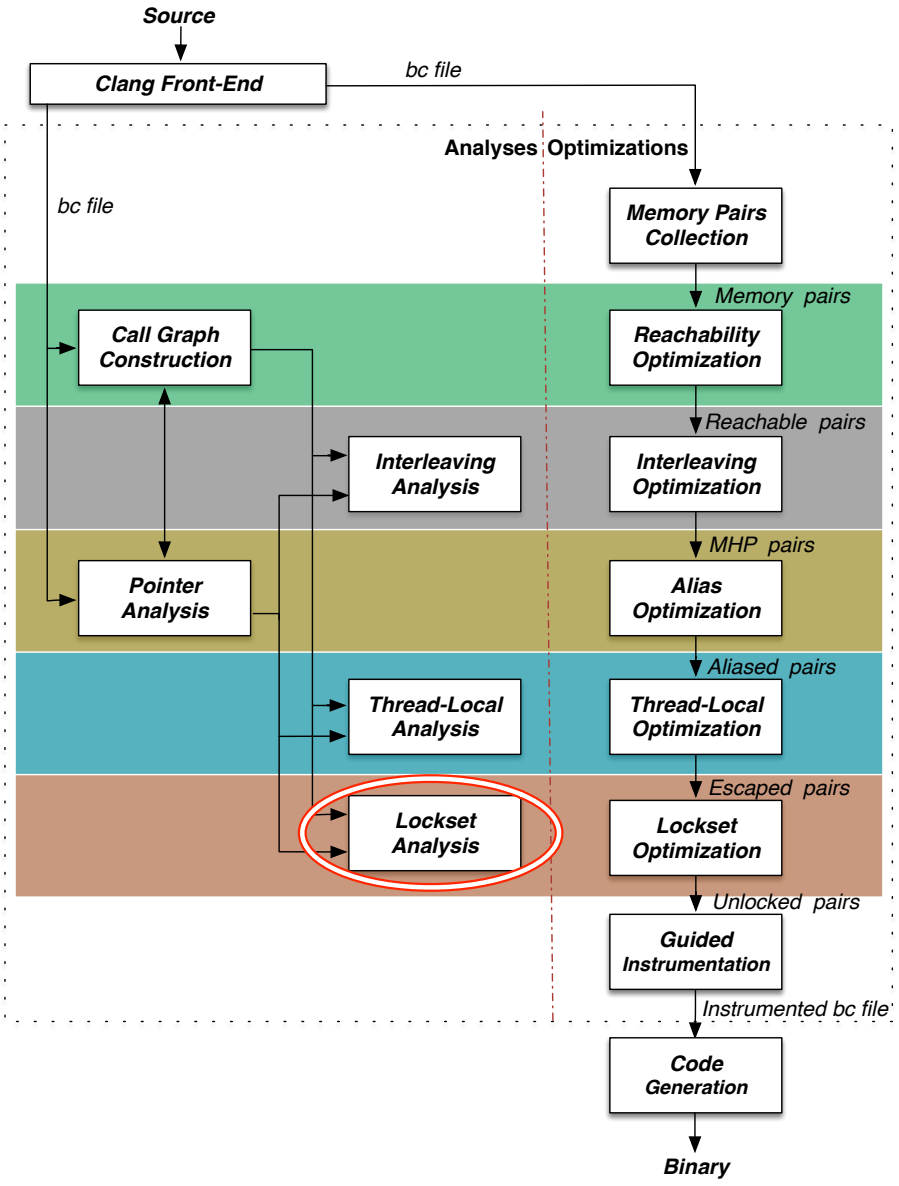
Framework



Framework

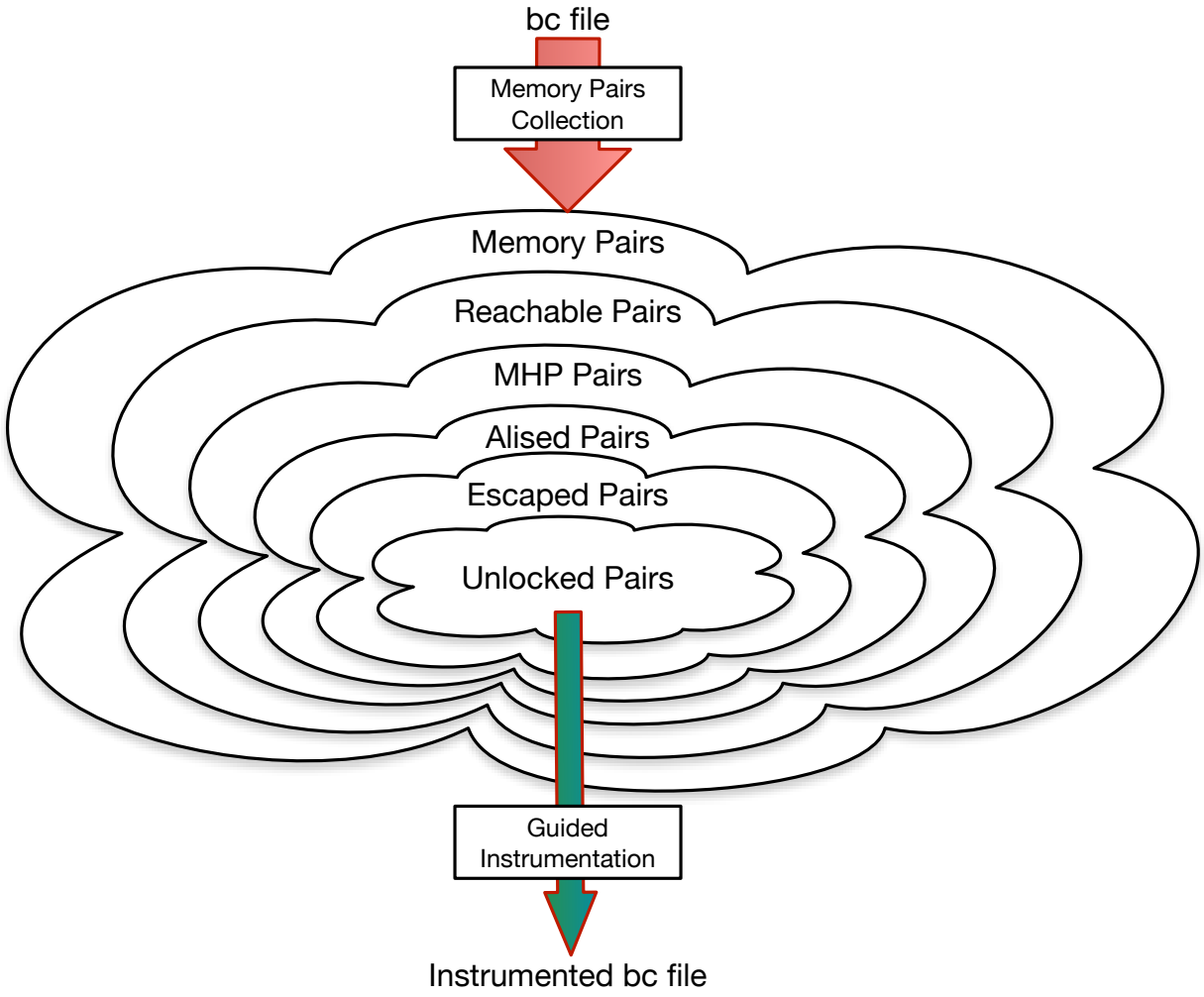


Framework



Refining Pairs

Only statements in the paris that are not filtered are instrumented for runtime check.



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);
   }
```

```
void main(){

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

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.

Outline

- Motivation
- Analysis phases
  - Thread Interleaving Analysis
  - Alias Analysis
  - Thread Local Analysis
  - Lock Analysis
- Evalution

Context-sensitive Thread Interleaving Analysis

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

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

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: $V \to V$ transfer functions associated with each node in an ICFG.

Interleaving Analysis Rule

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

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

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

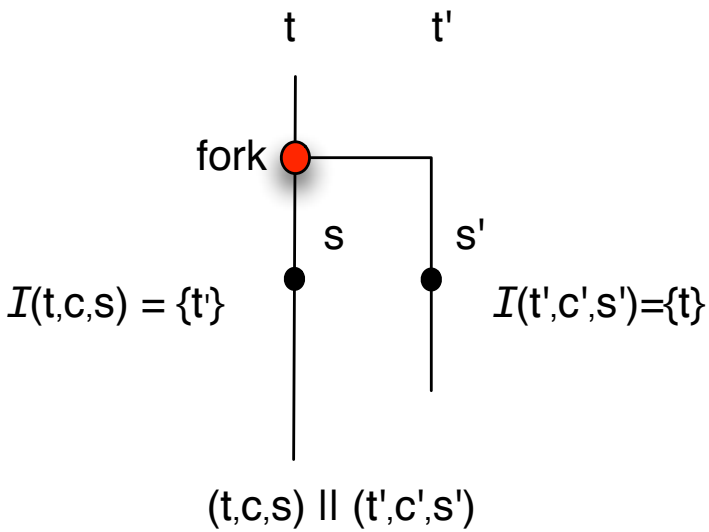
$$\texttt{[I-CALL]}\quad \frac{(t,c,\ell) \xrightarrow{call_i} (t,c',\ell') \quad c' = c.\mathit{push}(i)}{\mathcal{I}(t,c,\ell) \subseteq \mathcal{I}(t,c',\ell')}$$

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

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

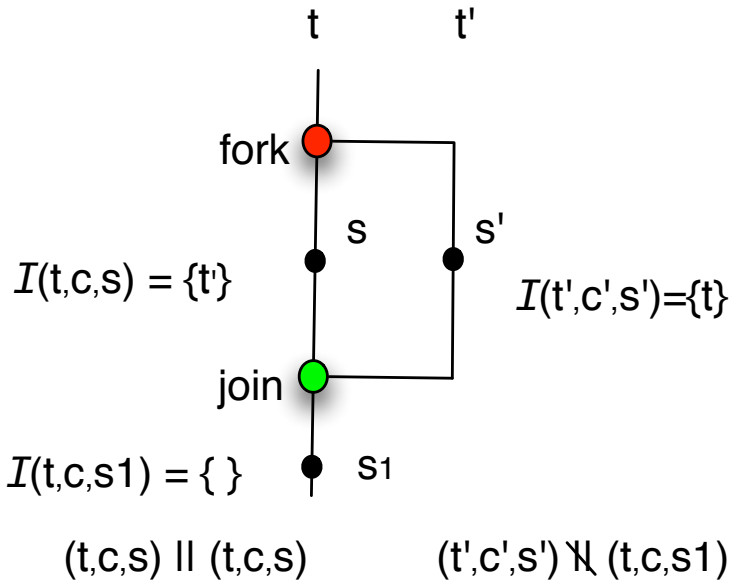
Interleaving Analysis Rule

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



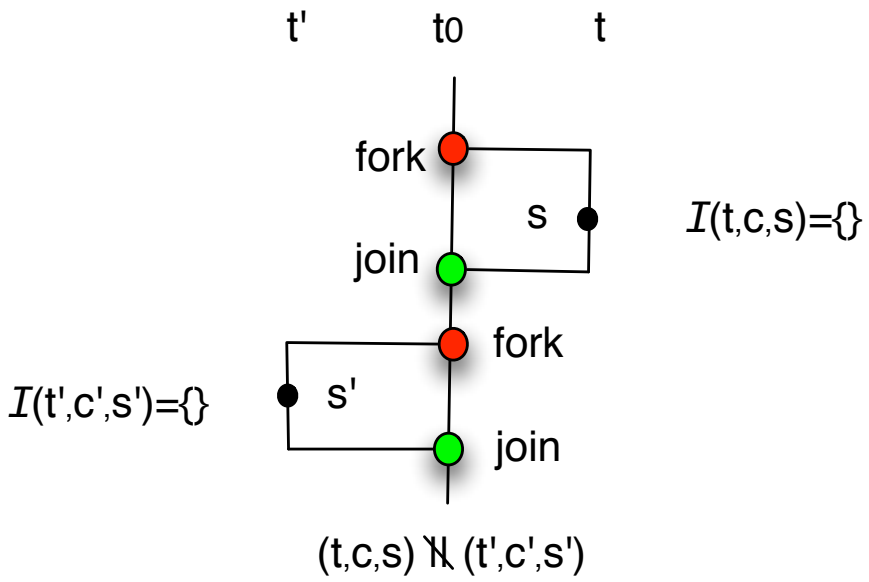
Interleaving Analysis Rule

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



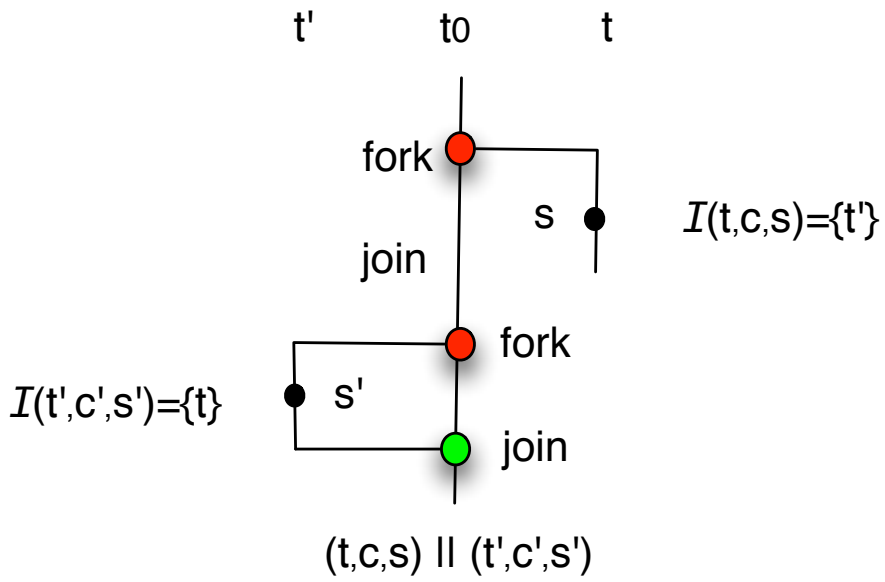
Interleaving Analysis Rule

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



Interleaving Analysis Rule

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



Interleaving Analysis Rule

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

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

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

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

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

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

Outline

- Motivation
- Analysis phases
  - Thread Interleaving Analysis
  - Alias Analysis
  - Thread Local Analysis
  - Lock Analysis
- Evalution

Alias Analysis

Obtain aliasing pairs by refining MHP store-load and store-store pairs $[(t, c, s), (t', c', s')]$, where $Alias(*p, *q)$ is the set of objects pointed to by both p and q.

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

```
   int x,y;
   int *p,*q,*r;
   p=&x;
   q=&x;
   r=&y;

   void main(){
    fork(t, foo);
s1:  *p=...;
s2:  *r=...;

   }

   void foo(){
s3: ...=*q;
   }
```

Outline

- Motivation
- Analysis phases
  - Thread Interleaving Analysis
  - Alias Analysis
  - Thread Local Analysis
  - Lock Analysis
- Evalution

Thread-Local Analysis

An object is not *thread-local*, i.e. *escaping*, if it escapes via

- arguments at a forksite
- global pointers

```
    void main(){
      int x,y;
      fork(t,foo,&x);
s1:   x=...;
      join(t);
s3:   y=...;
    }

    void foo(int* p){
s2:   ...=*p;
    }
```

Outline

- Motivation
- Analysis phases
  - Thread Interleaving Analysis
  - Alias Analysis
  - Thread Local Analysis
  - Lock Analysis
- Evalution

Lockset Analysis

Statements from different lock-unlock spans, are interference-free if these spans are protected by a common lock.
Our framework does this by performing a flow- and context-sensitive analysis for lock/unlock operations

```
    int x;
    mutex m;
    void main(){
      fork(t,foo);
s1:   x=...;
      lock(m);
s3:   x=...;
      unlock(m);
      join(t);
    }

    void foo(){
      lock(m);
s2:   ...=x;
      unlock(m);
    }
```

Outline

- Motivation
- Analysis phases
- **Evalution**

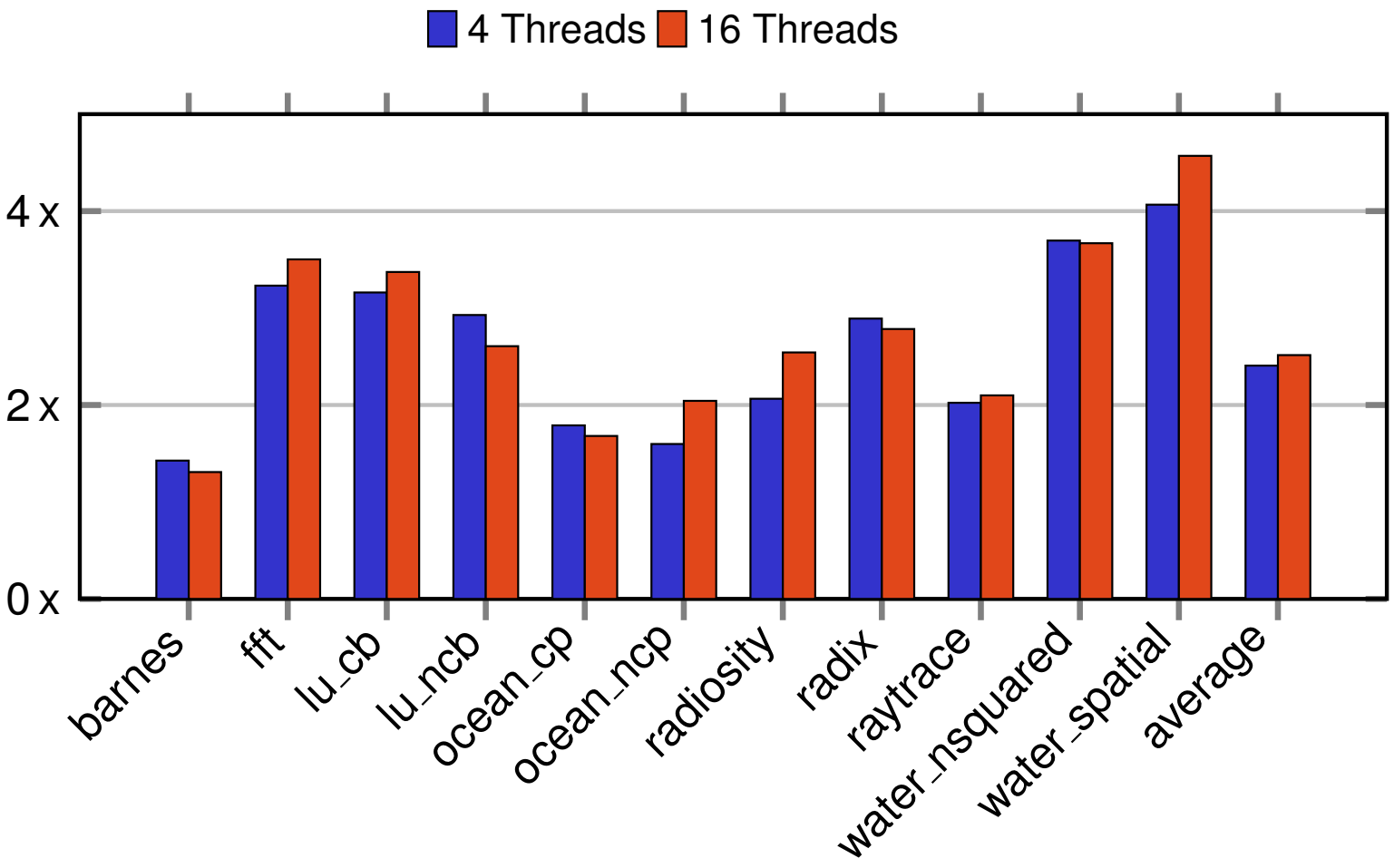
Evaluation

- Implementation:
  - On top of our previous open-source tool SVF (http://unsw-corg.github.io/SVF/)
  - Based on our previous papers CGO ’16 and ICPP ’15
- Benchmarks:
  - 11 SPLASH2 Pthread benchmarks
- Machine setup:
  - Ubuntu Linux 3.11.0-15-generic Intel Xeon Quad Core HT, 3.7GHZ, 64GB

Instrumentation Statistics (under Option -O0)

	Pthread API				TSan		Our Approach	
	Fork	Join	Lock	Unlock	Read	Write	Read	Write
barnes	1	1	12	12	2188	1222	982	601
fft	1	1	8	8	1048	387	576	261
lu_cb	1	1	6	6	1097	408	396	199
lu_ncb	1	1	6	6	840	303	392	182
ocean_cp	1	1	24	24	9531	2301	5722	1809
ocean_ncp	1	1	23	23	5465	1381	2909	1103
radiosity	3	3	38	50	5250	1917	3006	1500
radix	1	1	13	13	777	354	329	208
raytrace	1	1	13	16	6049	2368	2865	1057
water_nsquared	1	1	18	18	2188	703	1313	510
water_spatial	1	1	19	19	2437	833	1168	440

Speedups over Original TSan (under Option -O0)



Thanks!

Q & A