Inspector

Data Provenance using Intel Processor Trace (PT)

Jörg Thalheim
Pramod Bhatotia & Christof Fetzer

Technical University of Dresden

Data Provenance

…records transformations made on data to explain how the computation was performed

Motivation: Use-case examples

Dependability:

Debugging programs

Security:

Dynamic Information Flow Tracking (DIFT)

Efficiency:

Memory management for NUMA

Research gap

- Currently limited either to sequential programs

For parallel programs:

- Require manual annotations w/ new type systems
- Restrictive programming model & synchronization primitives

Design goals

- Transparency
    - Unmodified multithreaded programs
- Generality
    - Shared-memory model w/ POSIX sync. primitives
- Efficiency
    - Low overheads using a parallel provenance algorithm

Inspector: Easy to use!

```
$ inspector -- ./<program> <arguments>
```

1. Preloads the Inspector library

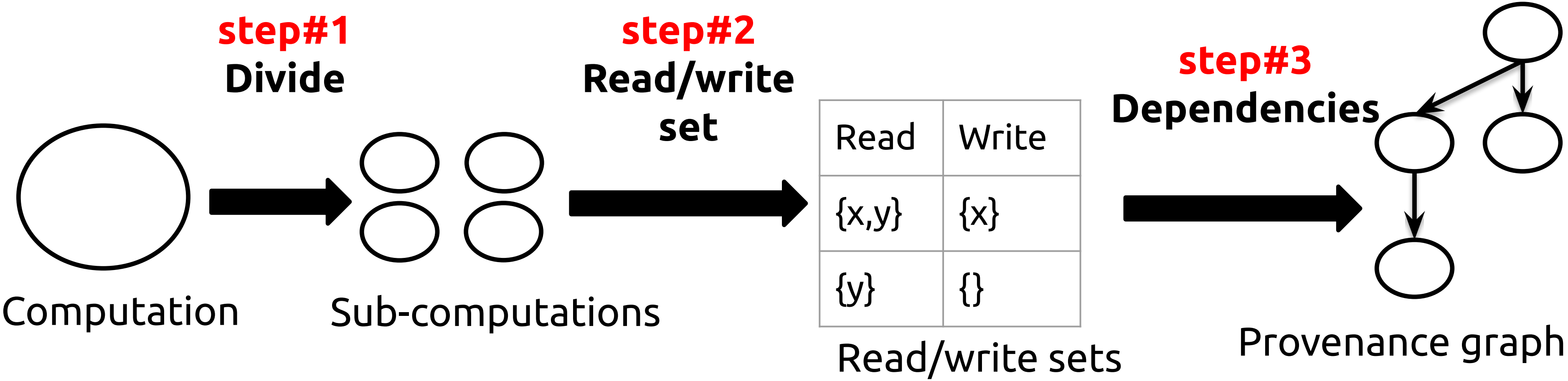
   LD_PRELOAD=libinspector.so

2. Executes “existing binaries” w/o re-compilation
3. Writes the provenance log to ./perf.data

Agenda

✓ Motivation
❏ Design
❏ Implementation
❏ Evaluation

Behind the scenes



step#1
Divide

Computation

Sub-computations

step#2
Read/write set

Read	Write
{x,y}	{x}
{y}	{}

Read/write sets

step#3
Dependencies

Provenance graph

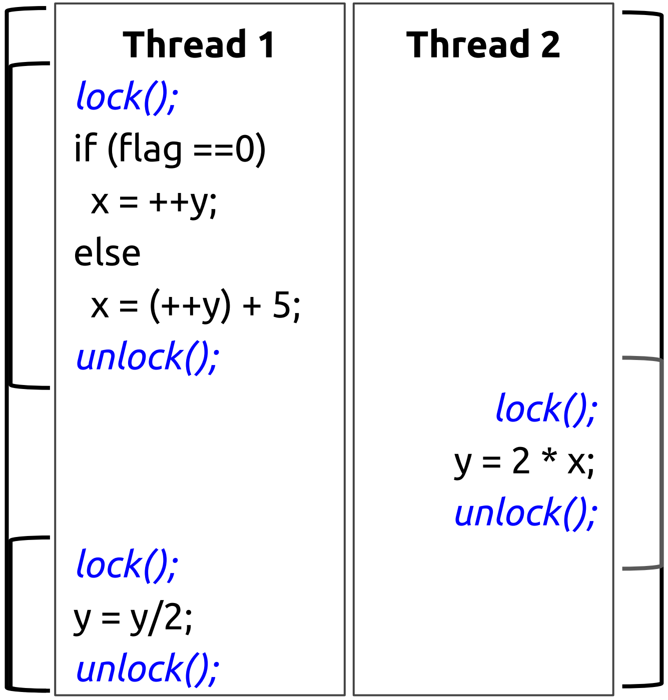
A simple example

Shared variables: x and y

```
Thread 1                    Thread 2
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();
                                  lock();
                               y = 2 * x;
                                unlock();
lock();
y = y/2;
unlock();
```

Step #1: Sub-computations

Shared variables: x and y



```
Thread 1                    Thread 2
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();
                                 lock();
                             y = 2 * x;
                              unlock();
lock();
y = y/2;
unlock();
```

Approaches:

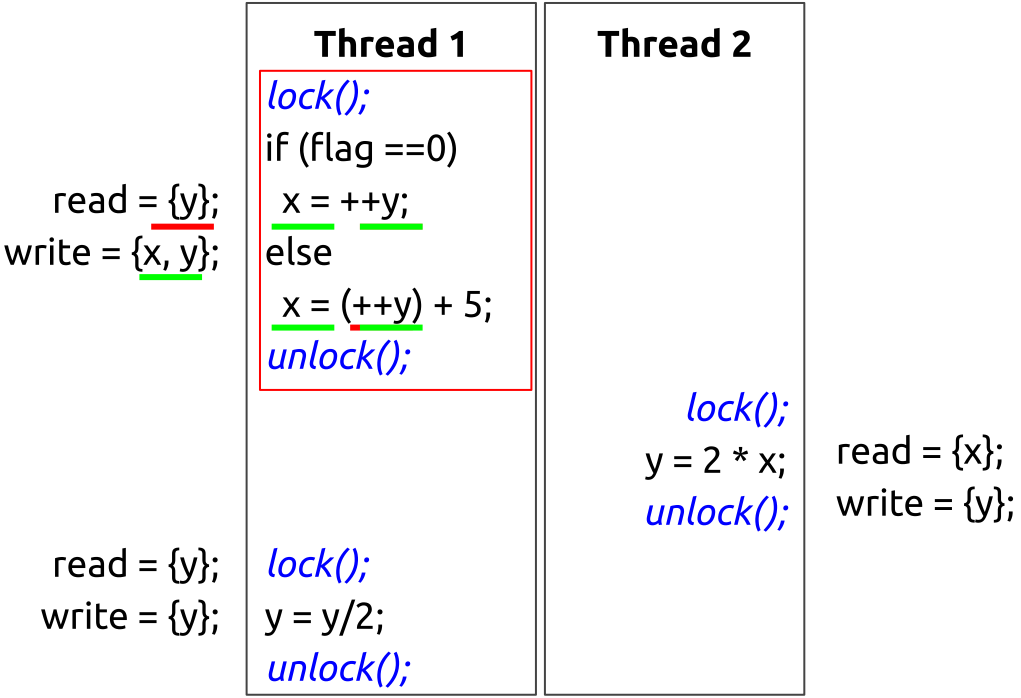
Coarsed Grained
- ○ Whole Thread
- → imprecise

Fine Grained
- ○ Every Instruction
- → expensive

Middle Ground
- ○ Sub-Computations

Step #2: Read-write sets

Shared variables: x and y



Thread 1

```
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();
```

read = {y};
write = {x, y};

```
lock();
y = y/2;
unlock();
```

read = {y};
write = {y};

Thread 2

```
lock();
y = 2 * x;
unlock();
```

read = {x};
write = {y};

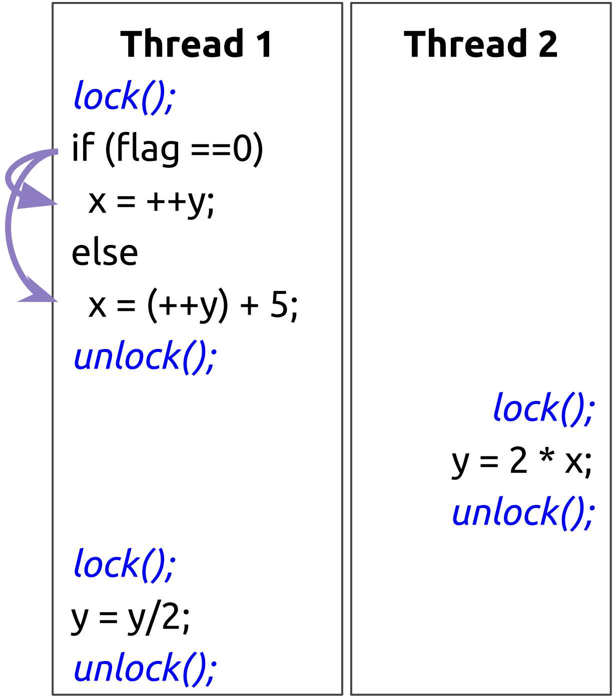
Step #3: Provenance graph

We record three dependencies:

A. Control
B. Schedule
C. Data

A: Control dependencies

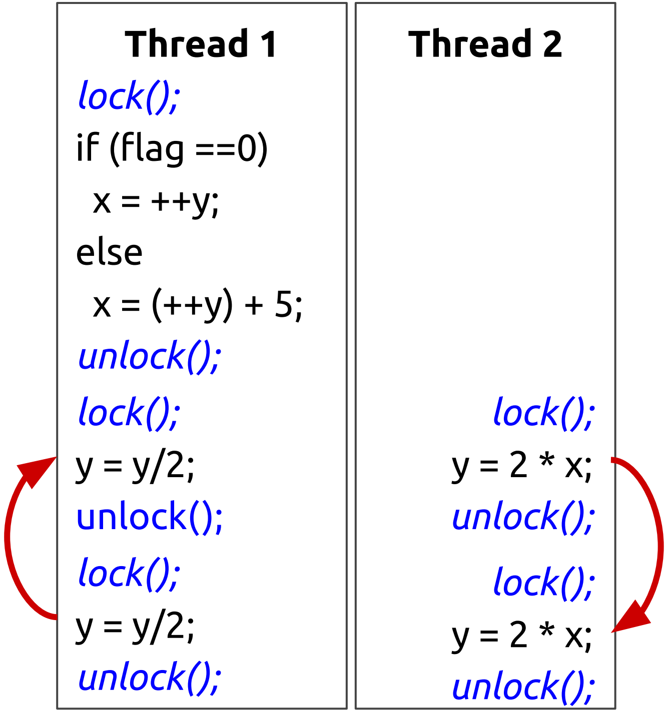
Shared variables: x and y



Thread 1	**Thread 2**
lock();	
if (flag ==0)	
x = ++y;	
else	
x = (++y) + 5;	
unlock();	
	lock();
	y = 2 * x;
	unlock();
lock();	
y = y/2;	
unlock();	

B: Synchronization dependencies

Shared variables: x and y

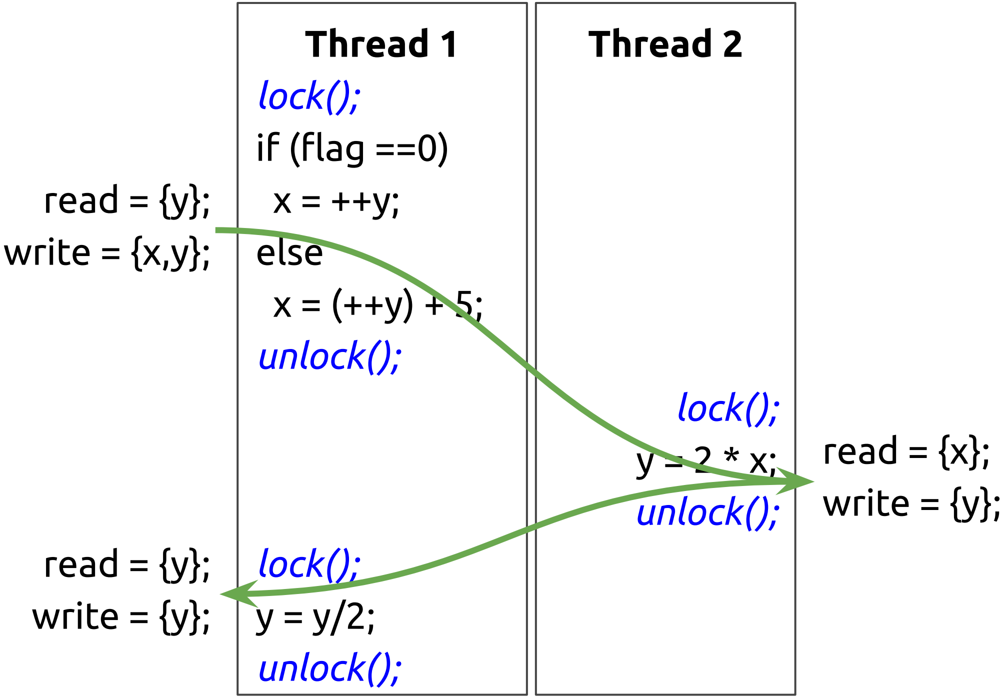


```
Thread 1                     Thread 2
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();
lock();                      lock();
y = y/2;                     y = 2 * x;
unlock();                    unlock();
lock();                      lock();
y = y/2;                     y = 2 * x;
unlock();                    unlock();
```

Change of Schedule

C: Data dependencies

Shared variables: x and y



Thread 1

Thread 2

read = {y};
write = {x,y};

```
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();
```

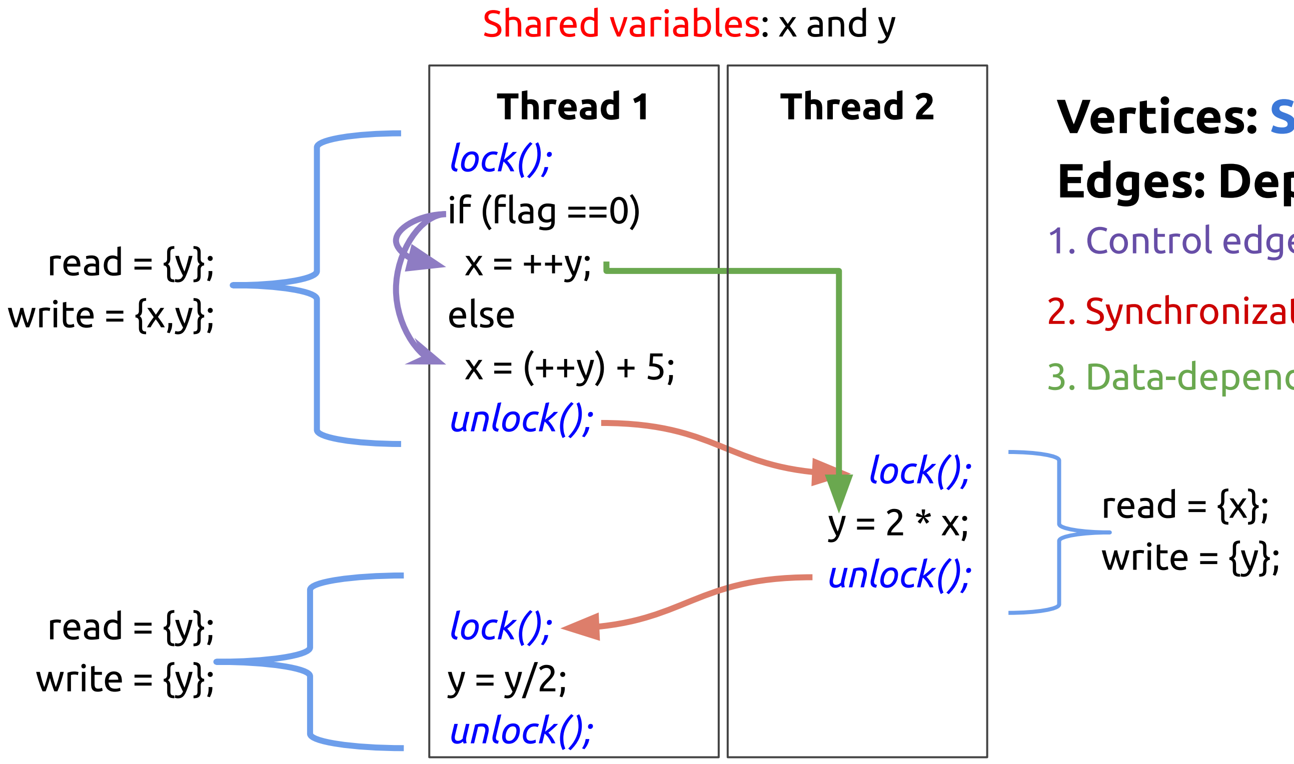
```
lock();
y = 2 * x;
unlock();
```

read = {x};
write = {y};

read = {y};
write = {y};

```
lock();
y = y/2;
unlock();
```

Concurrent Provenance Graph (CPG)



Shared variables: x and y

Thread 1

```
lock();
if (flag ==0)
  x = ++y;
else
  x = (++y) + 5;
unlock();



lock();
y = y/2;
unlock();
```

read = {y};
write = {x,y};

read = {y};
write = {y};

Thread 2

```
lock();
y = 2 * x;
unlock();
```

read = {x};
write = {y};

Vertices: Sub-Computations

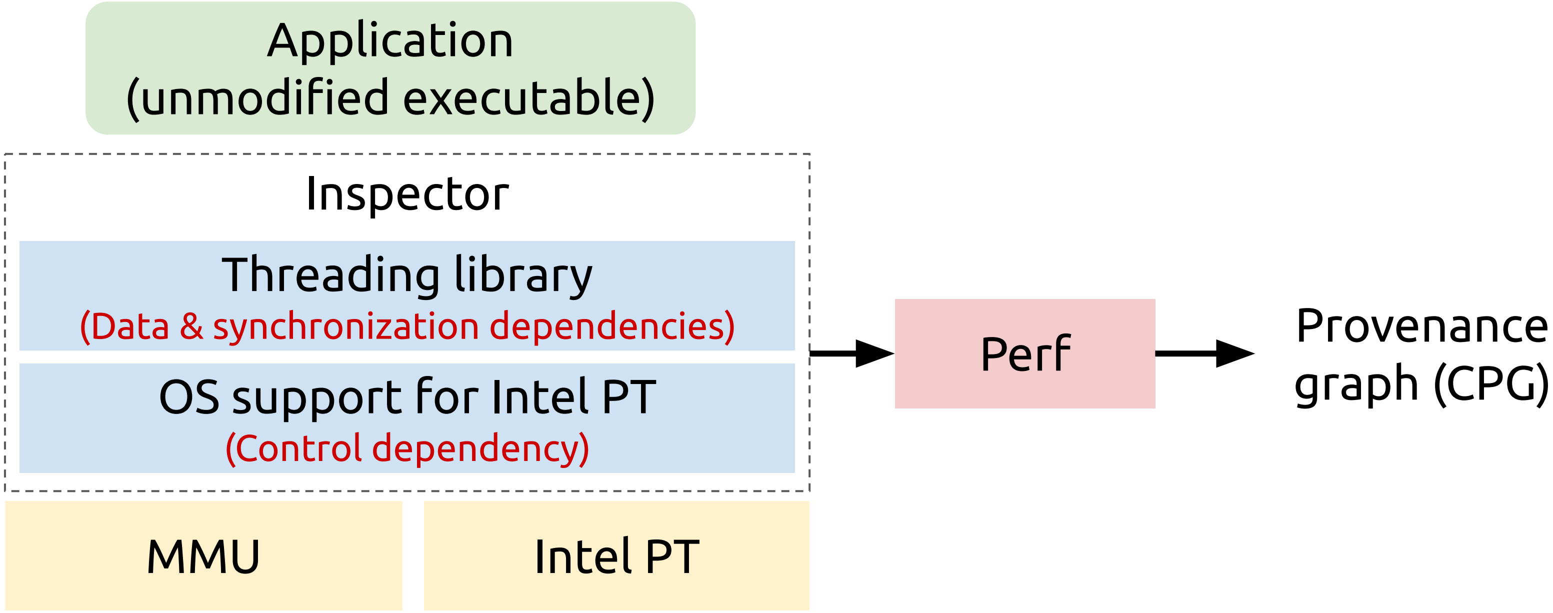
Edges: Dependencies

1. Control edges
2. Synchronization edges
3. Data-dependence edges

Agenda

- ✓ Motivation
- ✓ Design
- ❏ Implementation
- ❏ Evaluation

Inspector architecture



Application
(unmodified executable)

Inspector

Threading library
(Data & synchronization dependencies)

OS support for Intel PT
(Control dependency)

MMU

Intel PT

Perf

Provenance graph (CPG)

Agenda

- ✓ Motivation
- ✓ Design
- ✓ Implementation
- ❑ Evaluation

Evaluation

Questions:

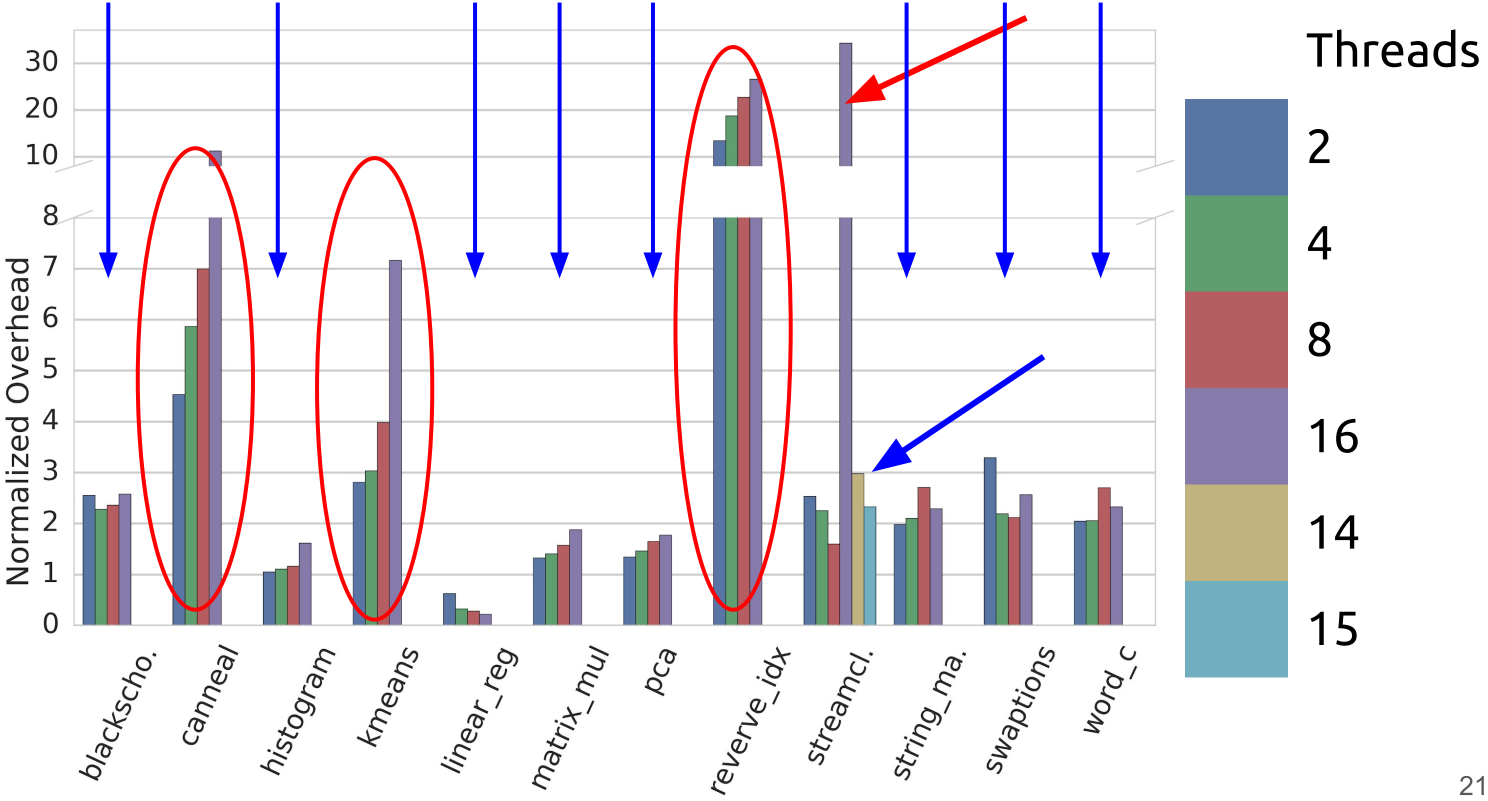
1. Performance overheads
2. Sources for these overheads

More results in the paper

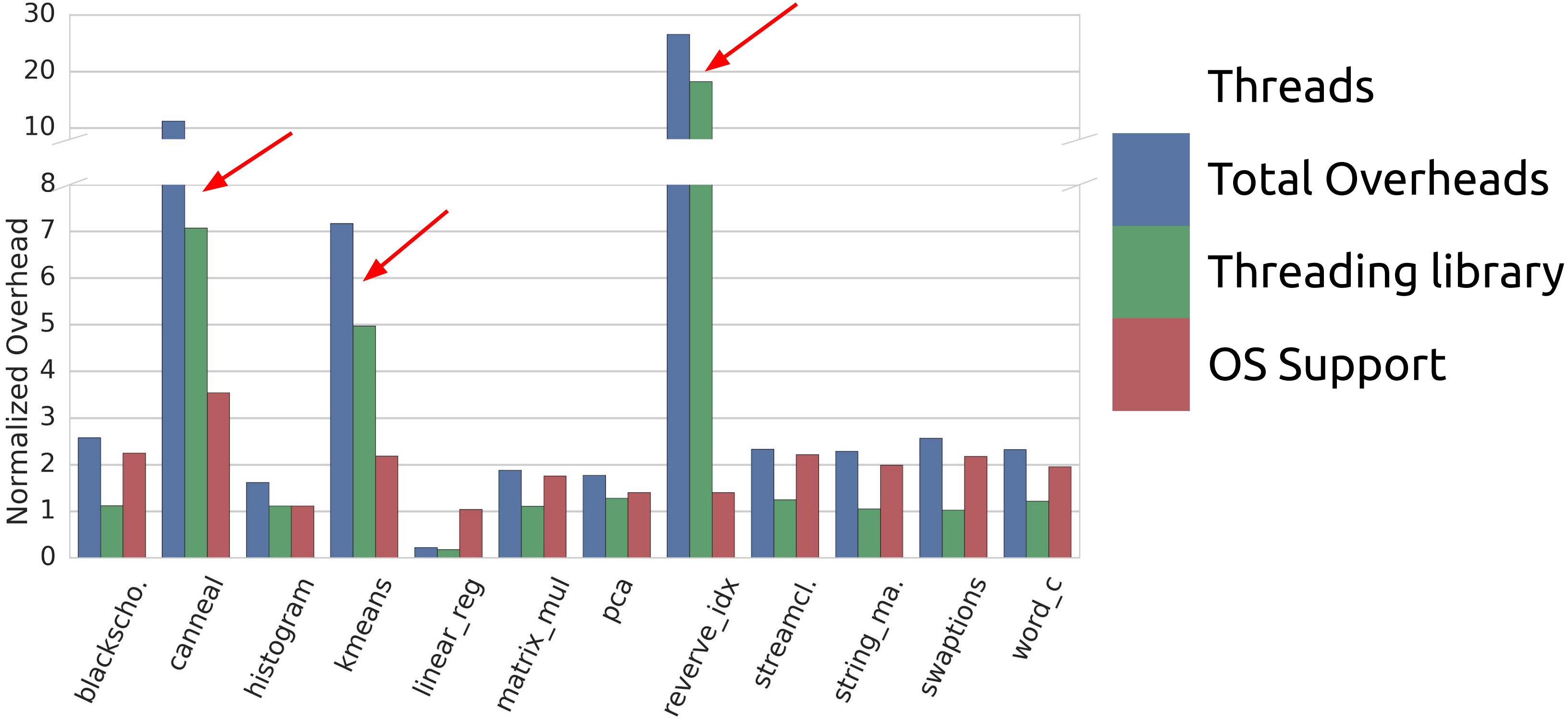
Experimental setup:

- Benchmarks: Phoenix 2.0 and PARSEC 3.0
- Platform: Intel Broadwell CPU with 8 cores (16 hyper-threads)

Q1: Performance overheads



Q2: Source of the overheads



Summary

Inspector: Data provenance using Intel Processor Trace (PT)

- Transparent: Targets unmodified multithreaded programs
- General: Supports the shared-memory model w/ POSIX sync primitives
- Efficient: Employs a parallel provenance algorithm

Usage: A dynamically linkable shared library

- Source Code: https://github.com/Mic92/inspector