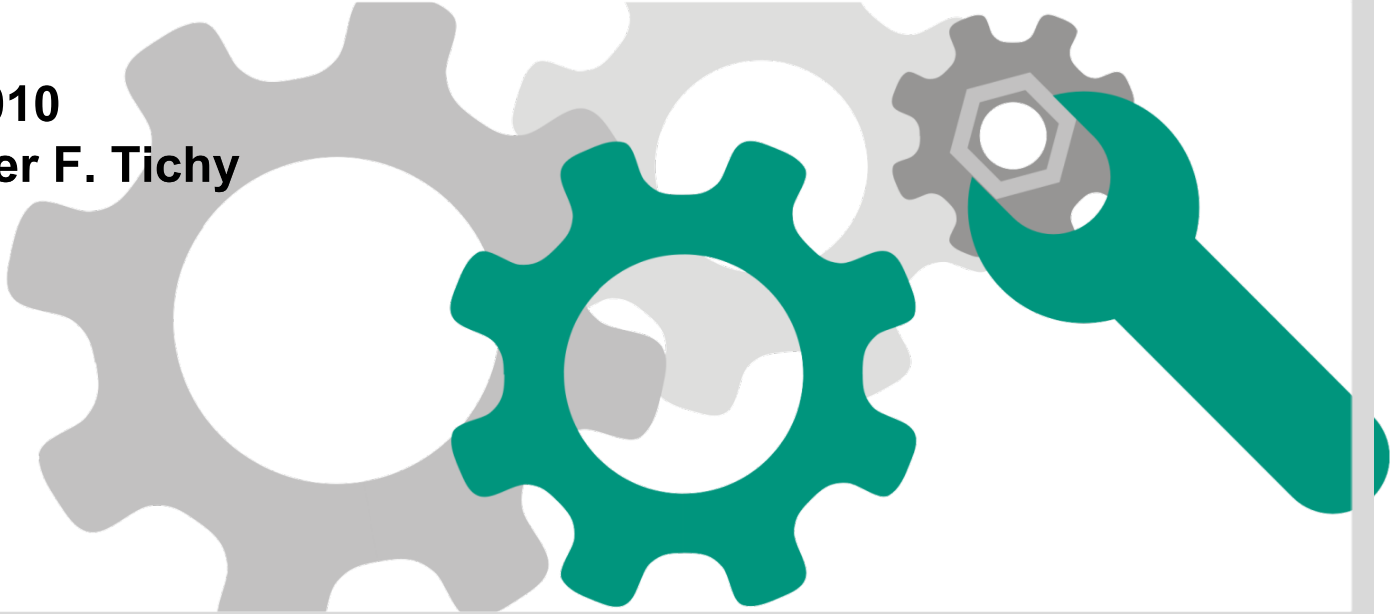
KIT
Karlsruhe Institute of Technology

Identifying Ad-hoc Synchronization for Enhanced Race Detection

IPD Tichy – Lehrstuhl für Programmiersysteme



IPDPS – 20 April, 2010
Ali Jannesari / Walter F. Tichy

KIT – die Kooperation von Forschungszentrum Karlsruhe GmbH und Universität Karlsruhe (TH)

Forschungszentrum Karlsruhe
in der Helmholtz-Gemeinschaft

Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

Motivation

- Data races (unsynchronized accesses to share variables) are a common defect in parallel programs.
- They are difficult to find.
- Race detectors are impractical
    - They produce thousands to millions of false warnings.
    - Programmers are overwhelmed by false positives.
- Why false positives?
    - Ad-hoc, programmer-defined synchronizations
    - Unknown synchronization libraries
    - Detectors cannot reason about these, causing many false positives
- Contribution: how to handle user-defined synchronization and unknown synchronization libraries, reducing false positives.

What is a Data Race?

- Two or more concurrent accesses to a shared location, at least one of them a write.

Thread 1	Thread 2
X = 0	
X++	T = X

Example – Data Race

- First Interleaving:

	Thread 1	Thread 2
1.	X=0	
2.		T=X
3.	X++	

- Second Interleaving:

	Thread 1	Thread 2
1.	X=0	
2.	X++	
3.		T=X

- T==0 or T==1?

How Can Data Races be Prevented?

- Explicit synchronization between threads:
  - Locks
  - Critical Sections
  - Barriers
  - Mutexes
  - Semaphores
  - Monitors
  - Events (signal/wait)
  - Etc.

Thread 1	Thread 2
Lock(m)	
X=0	
X++	
Unlock(m)	**Lock(m)**
	T=X
	Unlock(m)

Detection Approaches

- **Static**: perform a compile-time analysis of the code, reporting potential races.
- **Dynamic**: use tracing mechanism to detect whether a particular execution of a program actually exhibits data-races
  - The program must be instrumented with additional instructions to monitor shared variables and synchronization operations.
  - Every shared variable has a shadow cell in which the race detector stores additional information.

Dynamic Data Race Detection

- Dynamic Data Race Detection
  - Lockset analysis
  - Happens-before analysis
  - Hybrids (combining Lockset and Happens-before)

Lockset Analysis

- Observe all instances where a shared variable is accessed by a thread.
- Check whether the shared variable is always protected by the same lock.
- If variable isn’t protected, issue a warning.
- The lockset for a variable is initially set to all locks occurring in program.
- Whenever a variable is accessed, remove all locks from the variable’s lockset that are not currently protecting the variable.
- When the lockset is empty, issue a warning.

Lockset Analysis

Thread 1	Thread 2	Lockset_v
		{m1, m2, …}
Lock(m1); v = v + 1; Unlock(m1);		{m1}
	Lock(m1); v = v + 1; Unlock(m1);	{m1}
v = v + 1;		{ }

Lockset - False Positives

- The lockset algorithm will produce a false alarm in the following simple case:
  - Not able to detect signal-wait operation

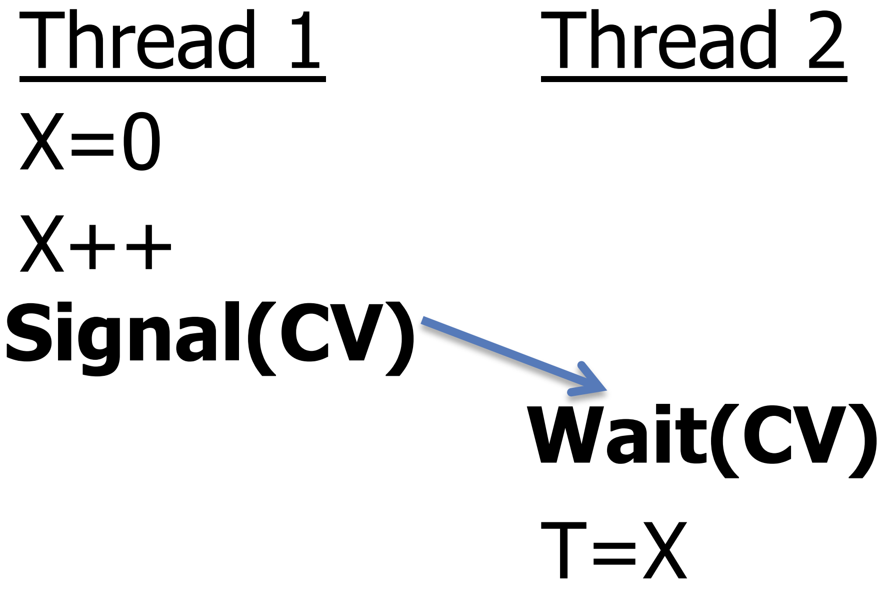
Thread 1	Thread 2
X=0	
X++	
Signal(CV)	
	Wait(CV)
	T=X

Happens-Before Relation

- Based on Lamport's Clock
- Let event **a** be in thread A and event **b** be in thread B.
    - If event **a** and event **b** are paired synchronization operations, construct a happens-before edge between them:
        - E.g. If a = unlock(mu) and b = lock(mu) then
          $a \xrightarrow{hb} b$ (a *happens-before* b)
- Shared accesses i and j are concurrent
    - if neither $i \xrightarrow{hb} j$ nor $j \xrightarrow{hb} i$ holds.
- Data races between threads are possible if accesses to shared variables are not ordered by happens-before.

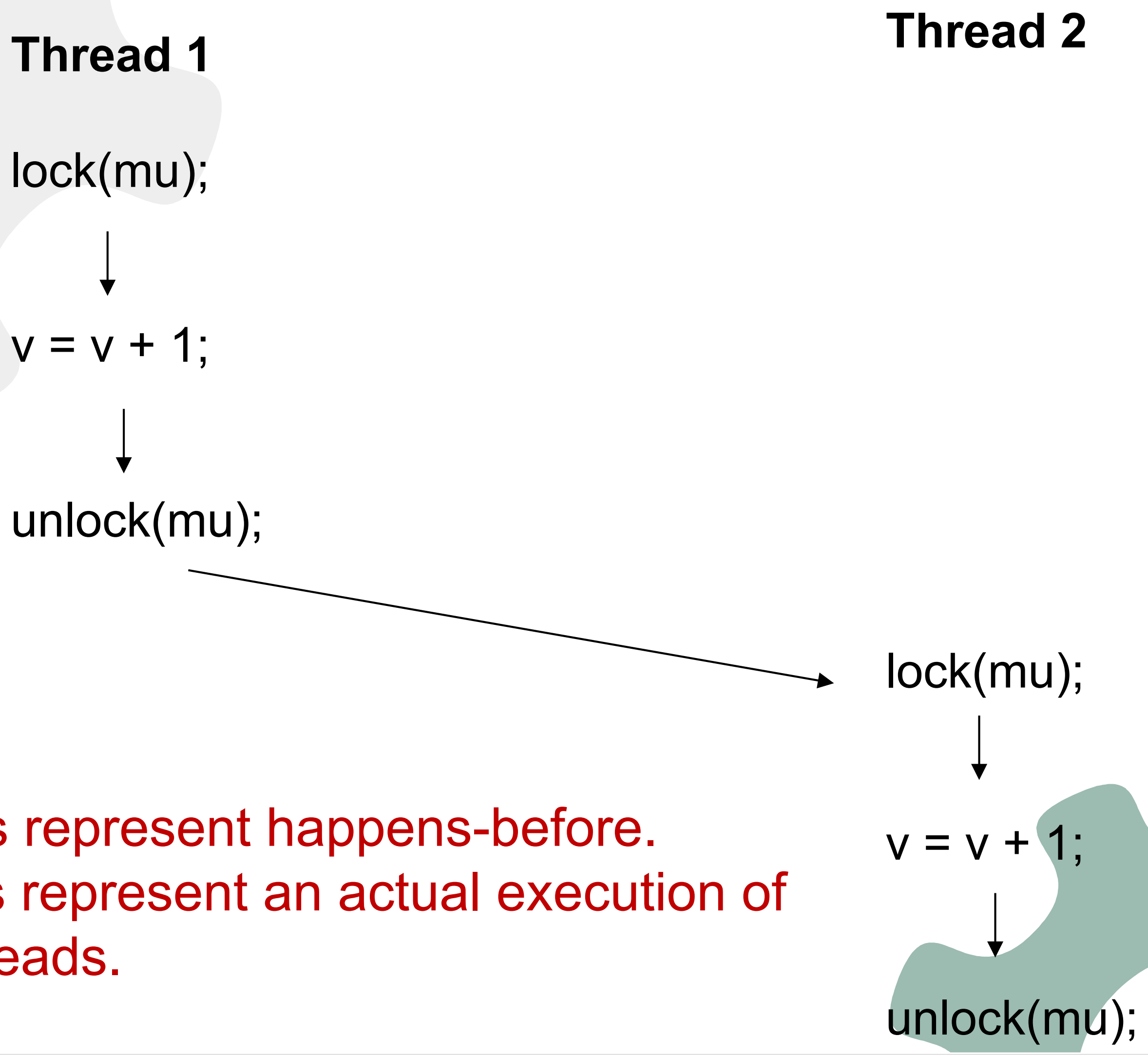
Happens-Before - Example 1

- Happens-before analysis will **eliminate** the false alarm in the following simple case:



Thread 1	Thread 2
X=0	
X++	
Signal(CV)	
	Wait(CV)
	T=X

Happens-Before - Example 2



Thread 1

lock(mu);

v = v + 1;

unlock(mu);

Thread 2

lock(mu);

v = v + 1;

unlock(mu);

The arrows represent happens-before.
The events represent an actual execution of the two threads.

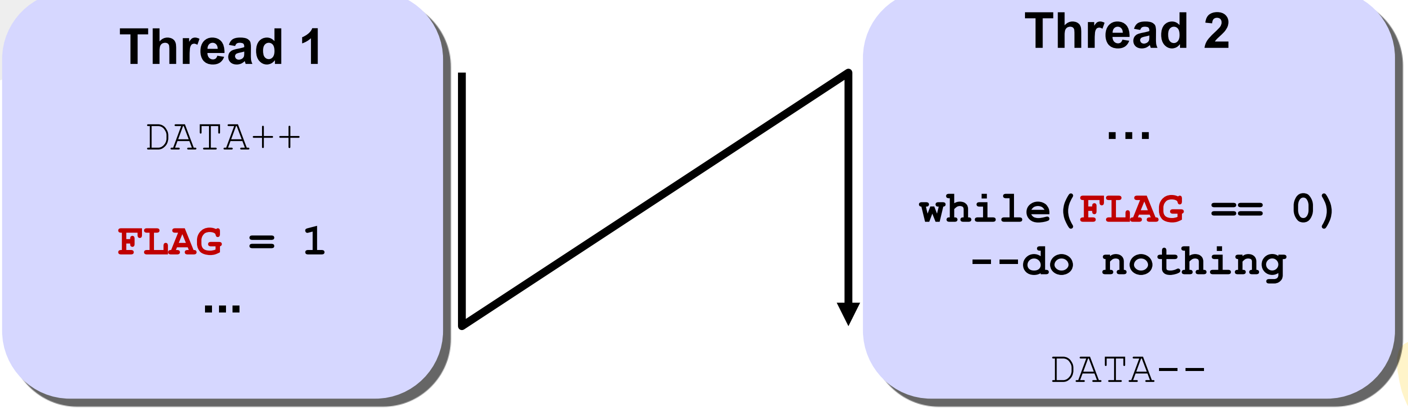
Helgrind+

- Efficient hybrid dynamic race detector
  - Introduces a new hybrid algorithm based on lockset algorithm and happens-before analysis
  - Does runtime analysis and uses code and semantic information
- Different memory state machines for
  - short-running applications (during development - unit test)
    - More sensitive, but produces more false positives
  - long-running applications (integration testing)
    - Less sensitive, might miss a race on first iteration, but not on second
- Automatically handling of synchronization bug patterns related to condition variables without any source code annotation
  - Lost signal detector
  - Spurious wake-up detection

Ad-hoc (User-defined) Synchronization

- Synchronization constructs implemented by user for performance reasons
  - High level synchronizations (e.g. task queues)
  - Spinning read loop instead of a library wait operation

/*Initially **FLAG** is zero */



Thread 1

```
DATA++

FLAG = 1
...
```

Thread 2

```
...

while(FLAG == 0)
  --do nothing

DATA--
```

- Ad-hoc synchronizations are widely used
  - 12 - 31 in SPLASH-2 and 32 - 329 in PARSEC 2.0

Ad-hoc Synchronization

- Source of false positives
  - **Apparent races** (e.g. `DATA`)
  - **Synchronization races** (e.g. **`FLAG`**)
  - Detectors should identify and suppress them
- We developed a dynamic method to detect ad-hoc synchronization
  - Automatically without any user action
  - Capable of identifying synchronization primitives of unknown libraries
    - Eliminates false races (apparent and synchronization races) caused by unknown synchronization primitives of a library
    - No need to upgrade the detector for a new library

Common Pattern

- Spinning read loop (spin-lock) is a common pattern for ad-hoc synchronizations
  - Happens-before relation induced by spin-lock synchronization

Thread 1

```
do_before(X)

Set CONDITION to TRUE
...
...
```

Counterpart write

Thread 2

```
...

while(!CONDITION){
 /* do_nothing() */
}

do_after(X)
```

Spinning read loop

Common Pattern

- Implementation of different synchronization primitives in libraries follows the same pattern as in spinning read loop
    - e.g. implementation of **`Barrier():`**

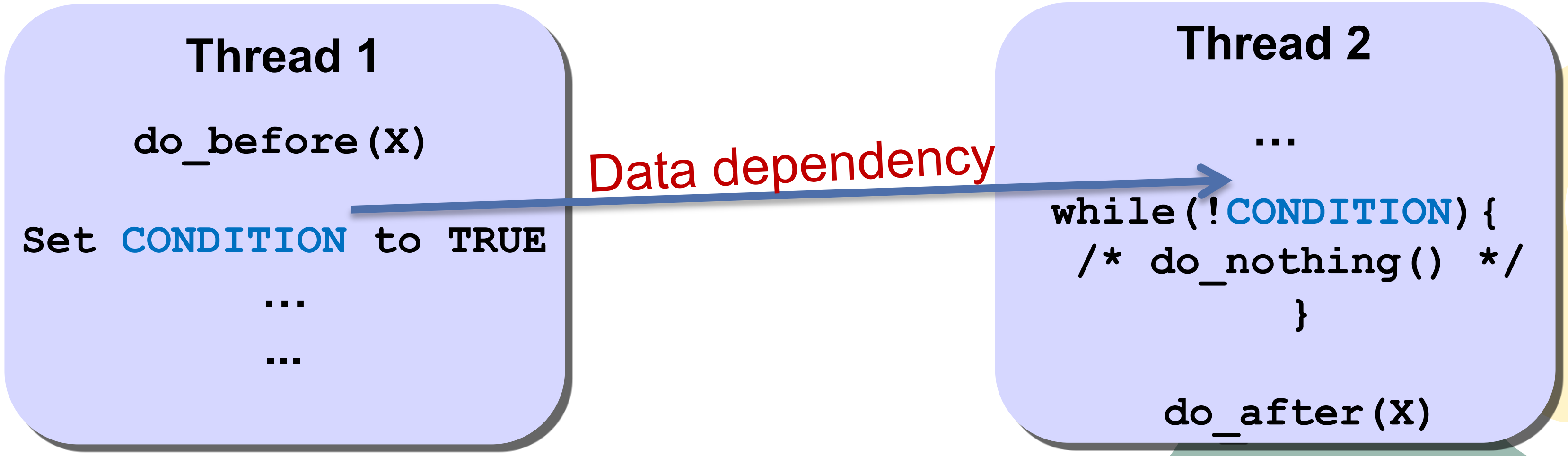
```
                 ...

             Lock(L)
                 counter++
            Unlock(L)

while(!counter!=NUMBER_THREADS){
        /* do_nothing() */
                  }
                 ...
```

Detecting Ad-hoc Synchronizations

- General dynamic approach
  - **Instrumentation** phase and
  - **Runtime** phase
- Instrumentation phase (code/semantic analysis)
  - Search the binary code to find all loops
    - Control flow analysis on the fly
    - Consider small loops (3 to 7 basic blocks)
  - Detect the spinning read loop based on the following criteria:
    - The loop condition involves at least on load instruction from memory
    - The value of loop condition is **not changed** inside the loop
  - Instrument the loop and mark the variables that affect the value of the loop condition to be treated specially.

Detecting Ad-hoc Synchronizations

- Runtime phase
  - Data dependency analysis
    - Monitor all write/read accesses
    - Identify the write/read dependency
      - Between the variables of instrumented spinning loop condition and those in counterpart write
    - Establish a happens-before relation between corresponding parts



Thread 1

```
do_before(X)

Set CONDITION to TRUE
...
...
```

Counterpart write

Data dependency

Thread 2

```
...
while(!CONDITION){
 /* do_nothing() */
}

do_after(X)
```

Spinning read loop

Detecting Unknown Synchronization Primitives

- Synchronization operations are ultimately implemented by spinning read loops
- Identify unknown synchronization operations if based on spinning read loops.
- **If this works, then we actually get a universal race detector**
  - Not limited to synchronization primitives of a particular library
  - General approach to identify synchronization operations
    - Information about libraries can be removed entirely from the detector

Implementation

- We implement the presented approach into our race detector **Helgrind$^+$**
- Helgrind$^+$
  - A hybrid dynamic race detector
    - Combines lockset algorithm and happens-before analysis
  - It is open source and built on top of Valgrind (a binary instrumentation tool)

Experiments & Evaluation

- The approach is evaluated on different benchmarks
  - data-race-test – a test suite framework for race detectors
  - PARSEC 2.0 Benchmarks
- All experiments were conducted on:
  - 2 * 1,86 GHz Xeon E5320 Quadcores, 8 GB RAM
  - OS: Linux (Ubuntu 8.10.2)
- New features in Helgrind$^+$
  - Reduces the number of false positives due to ad-hoc synchronizations and unknown libraries dramatically

Test Suite – data-race-test

- 120 different test cases (2-16 Threads)
  - Test cases are racy or race-free programs (using Pthread)
    - Includes difficult cases
  - Spinning read loop detection of up to 7 basic blocks
    - 24 false positives and one false negative are removed
  - Removing information about Pthread library (unknown library)
    - Only one false positive more

Tools	**False alarms**	**Missed races**	**Failed cases**	**Correctly analyzed cases**
Helgrind$^+$ lib	32	8	40	80
Helgrind$^+$ lib+spin(7)	**8**	**7**	**15**	**105**
Helgrind$^+$ nolib+spin(7)	9	7	16	104
DRD	13	20	33	87

Test Suite – data-race-test

- Best result achieved with seven basic blocks using spinning read loop detection as a complementary method
- In most cases spinning read loops contain more than 3 basic blocks
  - loop conditions use templates and complex function calls

Tools	**False alarms**	**Missed races**	**Failed cases**	**Correctly analysed cases**
Helgrind$^+$ lib+spin(3)	24	7	31	89
Helgrind$^+$ lib+spin(6)	23	7	30	90
Helgrind$^+$ lib+spin(7)	**8**	**7**	**15**	**105**
Helgrind$^+$ lib+spin(8)	8	7	15	105

PARSEC 2.0

Program	Parallelization model	LOC	Synchronisation primitives			Ad-hoc
			CVs	Locks	Barriers	
blackscholes	POSIX	812	-	-	✓	-
swaptions	POSIX	4,029	-	-	-	-
fluidanimate	POSIX	3,689	-	✓	-	-
canneal	POSIX	29,31	-	✓	-	-
freqmine	OpenMP	10,279	-	-	-	-
vips	GLIB	1,255	✓	✓	-	✓
bodytrack	POSIX	9,735	✓	✓	✓	✓
facesim	POSIX	1,391	✓	✓	-	✓
ferret	POSIX	2,706	✓	✓	-	✓
x264	POSIX	1,494	✓	✓	-	✓
dedup	POSIX	3,228	✓	✓	-	✓
streamcluster	POSIX	40,393	✓	✓	✓	✓
raytrace	POSIX	13,302	✓	✓	-	✓

Programs without Ad-hoc Synchronizations

- No false positives for first 4 programs
- In case of using the unknown library OpenMP only 2 false positives remain

Program	Para. model	LOC	Racy Contexts			
			Helgrind$^+$ lib	Helgrind$^+$ lib+spin	Helgrind$^+$ nolib+spin	DRD
blackscholes	POSIX	812	0	**0**	0	0
swaptions	POSIX	4,029	0	**0**	0	0
fluidanimate	POSIX	3,689	0	**0**	0	0
canneal	POSIX	29,31	0	**0**	0	0
freqmine	OpenMP	10,279	153.4	**2**	2	1000

Programs with Ad-hoc Synchronizations

- In 5 out of 8 programs false positives are completely eliminated

Program	Para. model	LOC	Racy Contexts			
			Helgrind$^+$ lib	Helgrind$^+$ lib+spin	Helgrind$^+$ nolib+spin	DRD
vips	GLIB	1,255	50.8	**0**	0	858.6
bodytrack	POSIX	9,735	36.8	**3.6**	32.4	34.6
facesim	POSIX	1,391	113.8	**0**	0	1000
ferret	POSIX	2,706	111	**2**	47	214.6
x264	POSIX	1,494	1000	**19**	28	1000
dedup	POSIX	3,228	1000	**0**	2	0
streamcluster	POSIX	40,393	4	**0**	1	1000
raytrace	POSIX	13,302	106,4	**0**	0	1000

Programs with Ad-hoc Synchronizations

- 3 programs produce false positives (2 to 19 warnings)
  - Function pointers for condition evaluation and obscure implementation of task queue (do not match the spin patterns)

Program	**Para. model**	**LOC**	**Racy Contexts**			
			Helgrind$^+$ lib	**Helgrind$^+$ lib+spin**	**Helgrind$^+$ nolib+spin**	**DRD**
vips	GLIB	1,255	50.8	**0**	0	858.6
bodytrack	POSIX	9,735	36.8	**3.6**	32.4	34.6
facesim	POSIX	1,391	113.8	**0**	0	1000
ferret	POSIX	2,706	111	**2**	47	214.6
x264	POSIX	1,494	1000	**19**	28	1000
dedup	POSIX	3,228	1000	**0**	2	0
streamcluster	POSIX	40,393	4	**0**	1	1000
raytrace	POSIX	13,302	106,4	**0**	0	1000

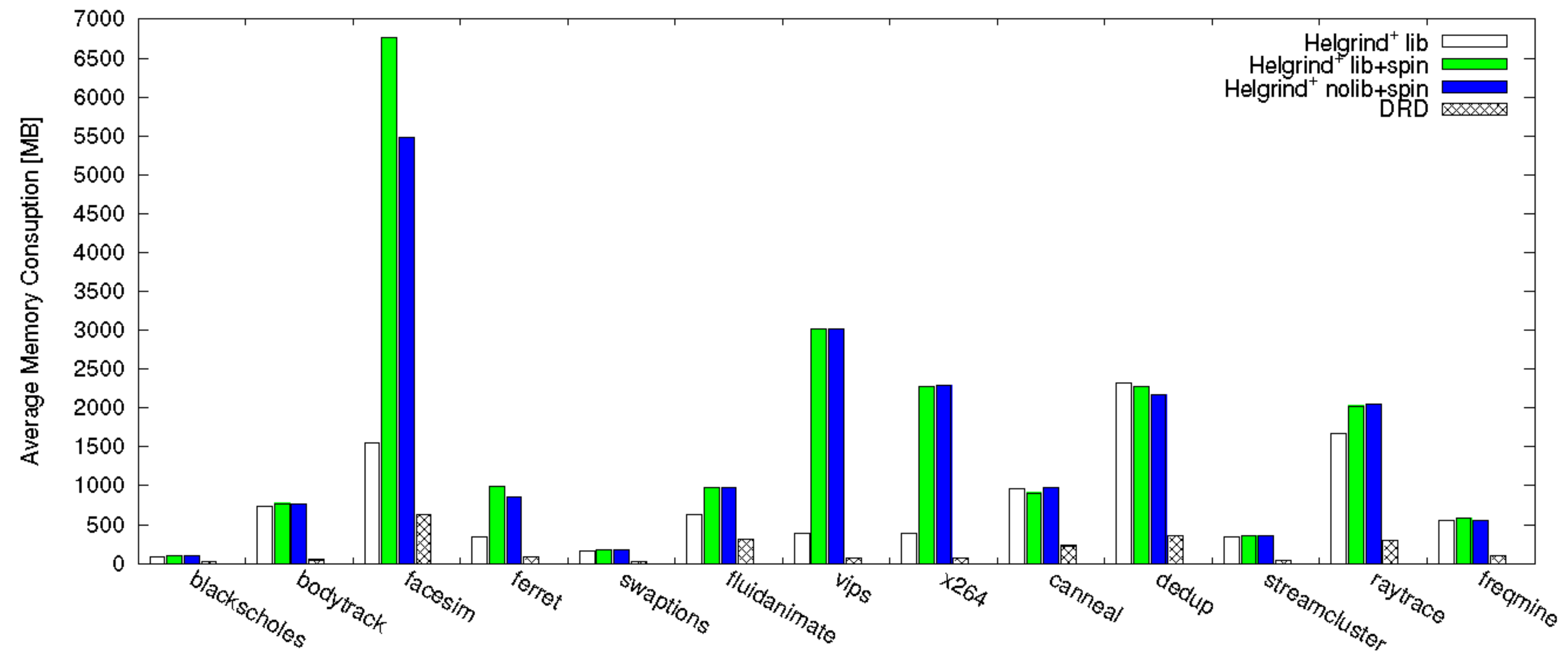
Universal Race Detector

Program	Para. model	LOC	Racy Contexts			
			Helgrind$^+$ lib	Helgrind$^+$ lib+spin	Helgrind$^+$ nolib+spin	DRD
			0	**0**	0	0
			0	**0**	0	0
			0	**0**	0	0
canneal	POSIX	29,31	0	**0**	0	0
freqmine	OpenMP	10,279	153.4	**2**	2	1000
vips	GLIB	1,255	50.8	**0**	0	858.6
bodytrack	POSIX	9,735	36.8	**3.6**	32.4	34.6
facesim	POSIX	1,391	113.8	**0**	0	1000
ferret	POSIX	2,706	111	**2**	47	214.6
x264	POSIX	1,494	1000	**19**	28	1000
dedup	POSIX	3,228	1000	**0**	2	0
streamcluster	POSIX	40,393	4	**0**	1	1000
raytrace	POSIX	13,302	106,4	**0**	0	1000

Happens-before detector
- **false positives are Slightly increased in 4 cases**

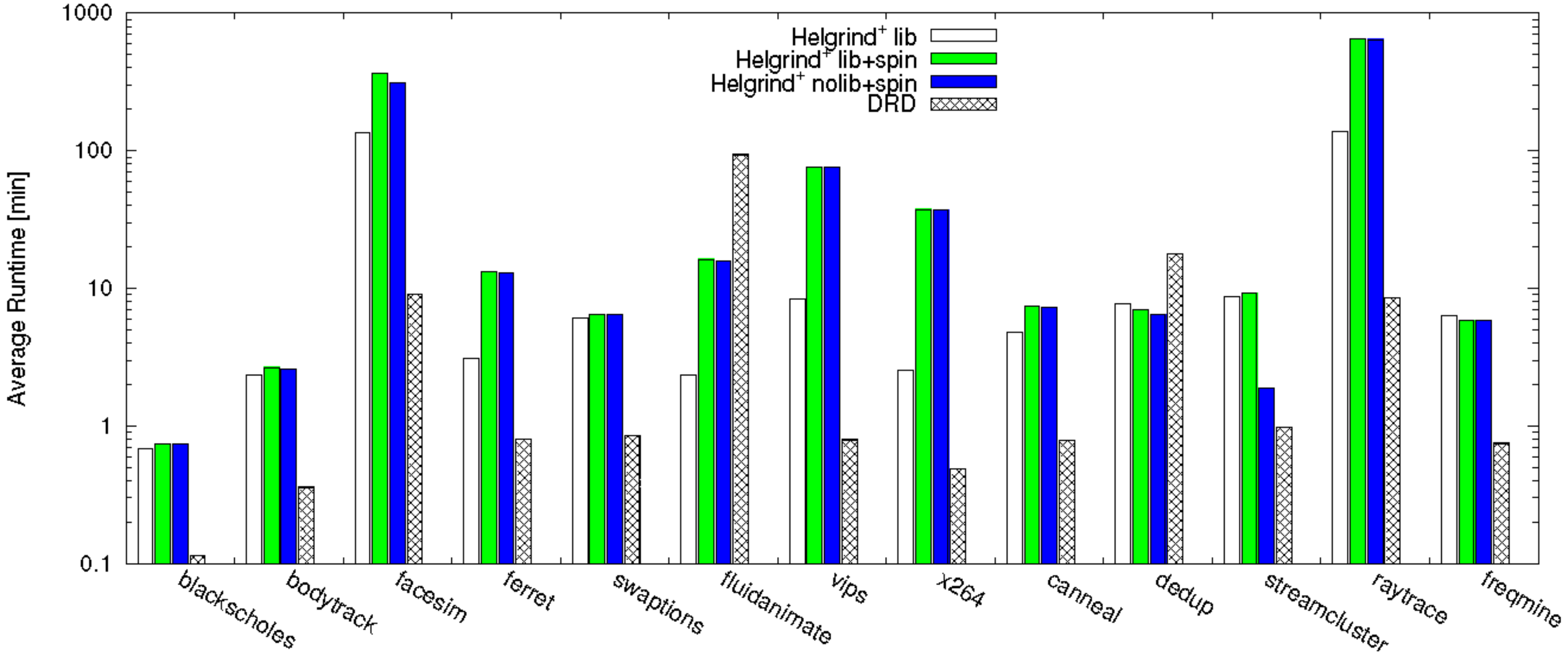
Performance

- Minor overhead due to the new feature for spinning read detection
- Memory consumption:



Performance

- Slight runtime overhead:



Summary

- Knowledge of all synchronization operations are crucial for accurate data race detection
  - Missing ad-hoc synchronizations causes a lot of false positives
- We present a dynamic method that is able to identify ad-hoc and unknown synchronizations in programs

- **Universal race Detector**
  - No need to upgrade the detector for unknown libraries
- Best results achieved when using it as complementary method (applicable for every race detector)
- Future work: Improving the accuracy of the universal race detector by identifying the lock operations (enabling lockset analysis).

Thank you

Questions?

This work: Ali Jannesari, Walter F. Tichy, *Identifying Ad-hoc Synchronization for Enhanced Race Detection,* to appear in *International Parallel & Distributed Processing Symposium **(IPDPS'10**), Apr 2010.*
Helgrind+: Ali Jannesari, Kaibin Bao, Victor Pankratius, Walter F. Tichy, *Helgrind+: An Efficient Dynamic Race Detector, Proceedings of the 23rd international Parallel & Distributed Processing Symposium (**IPDPS'09**), 2009.*
www.ipd.uka.de/Tichy/