Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs

Eli Pozniansky & Assaf Schuster

What is a Data Race?

- Two concurrent accesses to a shared location, at least one for writing.
  - Indicative of a bug

Thread 1	Thread 2
X++	T=Y
Z=2	T=X

How to prevent Data Races?

- Explicit synchronization
  - Locks
  - Critical Sections
  - Barriers
  - Mutexes
  - Semaphores
  - Monitors
  - Events
  - Etc.

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

Is This Enough?

- Yes!
- No!
  - Programmer dependent
    - For correctness – programmer will overdo.
      - Need tools to detect data races
  - Expensive
    - For efficiency – programmer will spend lifetime in removing redundant synch's.
      - Need tools to remove excessive synch's

Detecting Data Races?

- NP-hard **[Netzer&Miller 1990]**
  - Input size = # instructions performed
  - Even for 3 threads only
  - Even with no loops/recursion
- Execution orders/scheduling $(\#threads)^{thread\_length}$
- # inputs
- Detection-code's side-effects
- Weak memory, instruction reorder, atomicity

Where is Waldo?

Similar problem found in Java.util.vector

```
#define N 100
Type g_stack = new Type[N];
int g_counter = 0;
Lock g_lock;

void push( Type& obj ){lock(g_lock);...unlock(g_lock);}
void pop( Type& obj ) {lock(g_lock);...unlock(g_lock);}
void popAll( ) {
   lock(g_lock);
   delete[] g_stack;
   g_stack = new Type[N];
   g_counter = 0;
   unlock(g_lock);
}
int find( Type& obj, int number ) {
   lock(g_lock);
   for (int i = 0; i < number; i++)
       if (obj == g_stack[i]) break; // Found!!!
   if (i == number) i = -1;  // Not found… Return -1 to caller
   unlock(g_lock);
   return i;
}
int find( Type& obj ) {
   return find( obj, g_counter );
}
```

Apparent Data Races

- Based only the behavior of the explicit synch
  - not on program semantics
- Easier to locate
- Less accurate

Initially: grades = oldDatabase; updated = **false**;

Thread *T.A.*

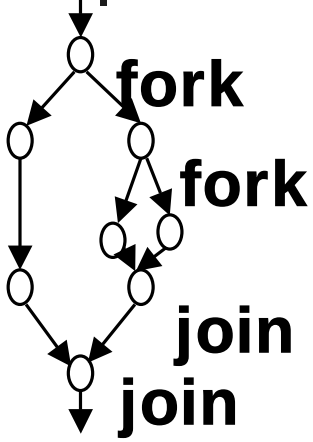
```
grades = newDatabase;
updated = true;
```

Thread *Lecturer*

```
while (updated == false);
X:=grades.gradeOf(lecturersSon);
```

- Exist iff "real" data race exist ☺
- Detection is still NP-hard ☹

Detection Approaches



- Restricted pgming model
  - Usually fork-join
- Static
  - Emrath, Padua 88
  - Balasundaram, Kenedy 89
  - Mellor-Crummy 93
  - Flanagan, Freund 01
- Postmortem
  - Netzer, Miller 90, 91
  - Adve, Hill 91
- On-the-fly
  - Nudler, Rudolph 88
  - Dinning, Schonberg 90, 91
  - Savage et.al. 97
  - Itzkovits et.al. 99
  - Perkovic, Keleher 00

Issues:

- pgming model
- synch' method
- memory model
- accuracy
- overhead
- granularity
- coverage

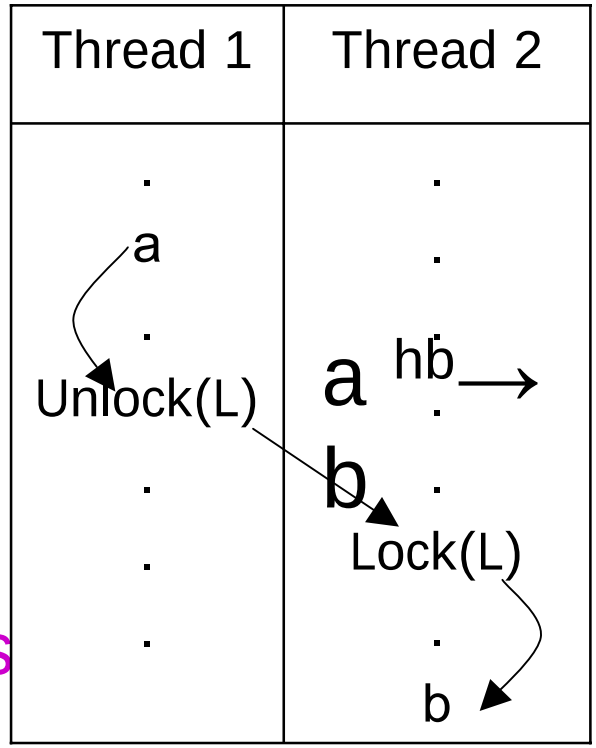
MultiRace Approach

- On-the-fly detection of “apparent data races”
- Two detection algorithms
  - Lockset **[Savage, Burrows, Nelson, Sobalvarro, Anderson 97]**
  - Djit^{+} **[Itzkovitz, Schuster, Zeev-ben-Mordechai 99]**
    - Correct even for weak memory systems ☺
- Detection granularity
  - Variables and Objects
  - Especially suited for OO programming languages
- Source-code (C++) instrumentation + Memory mappings
  - Transparent ☺
  - Low overhead ☺

Djit$^+$
Apparent Data Races

Lamport's *happens-before* partial order

- a, b concurrent if neither $a \xrightarrow{hb} b$ nor $b \xrightarrow{hb} a$
  - ➔ Apparent data race
  - Otherwise, they are "synchronized"
- Djit$^+$ basic idea: check each access performed against all "previously performed" accesses



Djit$^+$
Local Time Frames (LTF)

- The execution of each thread is split into a sequence of *time frames*.
- A new time frame starts on each unlock.

Thread	LTF
x = 1	1
lock(m1)	
z = 2	1
lock(m2)	
y = 3	1
unlock(m2)	
z = 4	2
unlock(m1)	
x = 5	3

Djit+
Local Time Frames (LTF)

- The execution of each thread is split into a sequence of *time frames*.
- A new time frame starts on each unlock.
- For every access there is a timestamp = a vector of LTFs known to the thread at the moment the event takes place

Thread	LTF
x = 1	1
lock(m1)	
z = 2	1
lock(m2)	
y = 3	1
unlock(m2)	
z = 4	2
unlock(m1)	
x = 5	3

Djit$^+$
Vector Time Frames (VTF)

- A vector $st_t[.]$ for each thread t
- $st_t[t]$ is the LTF of thread t
- $st_t[u]$ stores the latest LTF of u known to t
- If u is an acquirer of t's unlock

  for k=0 to maxthreads-1

  $st_u[k] = \max(st_u[k], st_t[k])$

Djit+
Vector Time Frames Example

Thread 1		Thread 2		Thread 3	
	(1 1 1)		(1 1 1)		(1 1 1)
write x					
unlock(m1)					
read z	(2 1 1)				
		unlock(m1)			
		read y	(2 1 1)		
		unlock(m2)			
		write x	(2 2 1)		
				unlock(m2)	
				write x	(2 2 1)

Djit^{+}
Checking Concurrency

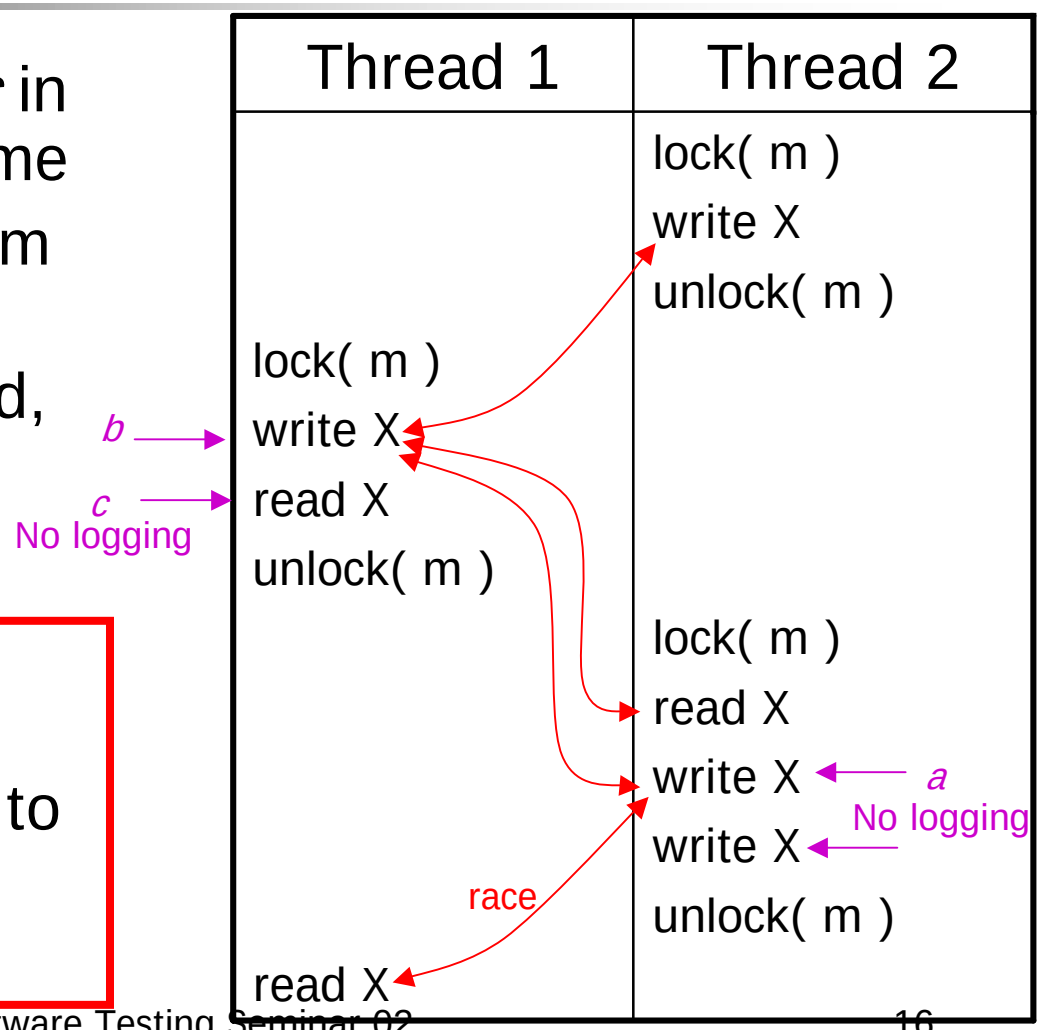
$$P(a,b) \square (a.type = write \square b.type = write) \square$$
$$\square (a.time\_frame \geq st_{b.thread\_id}[a.thread\_id])$$

- *a* was logged and tested earlier.
- *b* is currently performed.
- P returns TRUE iff *a* and *b* form a data race.

Djit+
Which Accesses to Check?

- *a* in thread t_1, and *b* and *c* in thread t_2 in same time frame
- *b* precedes *c* in the program order.
- If *a* and *b* are synchronized, then *a* and *c* are synchronized as well.

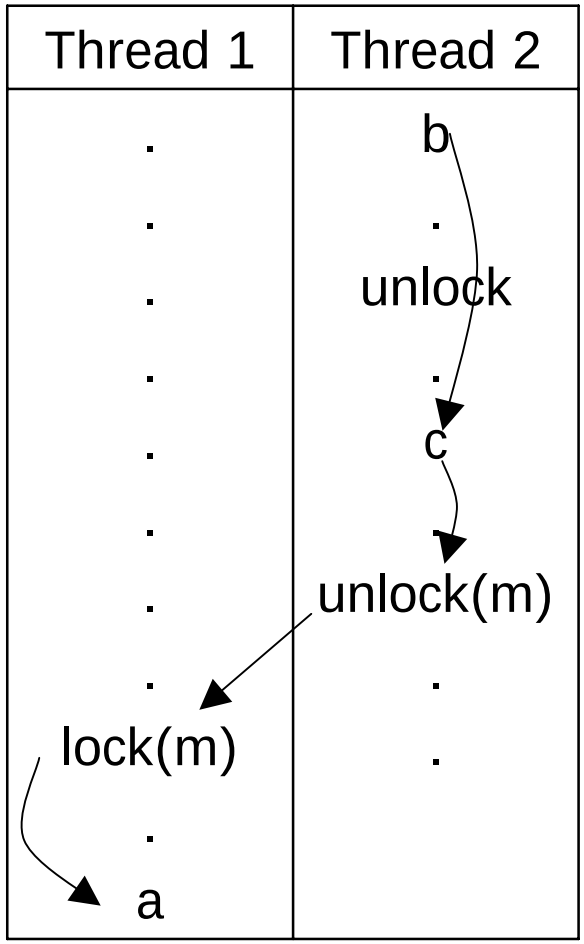
➔ It is sufficient to record only the first read access and the first write access to a variable in each time frame.



Djit+
Which Time Frames to Check?

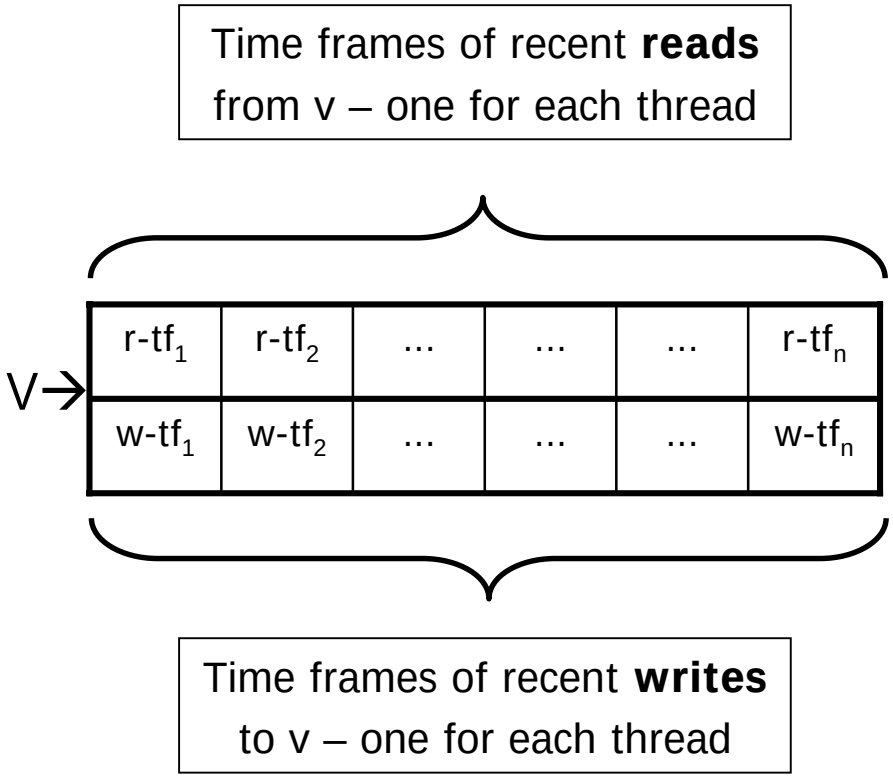
- *a* currently occurs in t_1
- *b* and *c* previously in t_2
- If *a* is synchronized with *c* then it is certainly synchronized with *b*.

➔ It is sufficient to check the current access with the "most recent" accesses in each of the other threads.



Djit+ Access History

- On each first read and first write to *v* in a time frame every thread updates the access history of *v*.
- If the access to *v* is a read, the thread checks all recent writes by other threads to *v*.
- If the access is a write, the thread checks all recent reads as well as all recent writes by other threads to *v*.



Djit$^+$
Pros and Cons

☺ No false alarms

☺ No missed races (in a given scheduling)

☹ Very sensitive to differences in scheduling

☹ Requires enormous number of runs. Yet:
cannot prove tested program is race free.

Lockset
The Basic Algorithm

- *C(v)* set of locks that protected all accesses to *v* so far
- *locks_held(t)* set of currently acquired locks

- Algorithm:
  - For each *v*, init *C(v)* to the set of all possible locks
  - On each access to *v* by thread *t*:
    - $lh_v \leftarrow locks\_held(t)$
    - if it is a read, then $lh_v \leftarrow lh_v \cup \{readers\_lock\}$
    - $C(v) \leftarrow C(v) \cap lh_v$
    - if $C(v) = \emptyset$, issue a warning

Lockset Example

r_l = *readers_lock* prevents from multiple reads to generate false alarms

Thread 1	Thread 2	lh_v	$C(v)$
		{ }	{m1, m2,r_l}
lock(m1)			
read v		{m1,r_l}	{m1,r_l}
unlock(m1)			
		{ }	
	lock(m2)		
	write v	{m2}	{ }
	unlock(m2)		
		{ }	

Warning: locking discipline for *v* is Violated !!!

Lockset

- Locking discipline: every shared location is consistently protected by a lock.
- Lockset detects violations of this locking discipline.

Thread 1	Thread 2	lh_v	$C(v)$
		{ }	{m1, m2}
lock(m1)			
read v		{m1}	{m1}
unlock(m1)			
		{ }	
	lock(m2)		
	write v	{m2}	{ }
	unlock(m2)		
		{ }	

Warning: locking Discipline for *v* is Violated

Lockset vs. Djit^{+}

Thread 1	Thread 2
y++[1]	
lock(m)	
v++	
unlock(m)	
	lock(m)
	v++
	unlock(m)
	y++[2]

[1] $\xrightarrow{hb}$ [2], yet there is a data race on y under a different scheduling, since the locking discipline is violated

Lockset
Which Accesses to Check?

- *a* and *b* in same thread, same time frame, *a* precedes *b*
- Then: $Locks_a(v) \square Locks_b(v)$
  - $Locks_u(v)$ set of real locks held during access *u* to *v*.
- It follows that:
  - $[C(v) \cap Locks_a(v)] \square [C(v) \cap Locks_b(v)]$
  - If $C(v) \cap Locks_a(v) \neq \square$ then $C(v) \cap Locks_b(v) \neq \square$

➔ Only first accesses need be checked in every time frame

Thread	$Locks_u(v)$
unlock	
…	
lock(m1)	
write x	{m1}
write x	{m1}= {m1}
lock(m2)	
write x	{m1,m2}□
unlock(m2)	{m1}
unlock(m1)	

Lockset
Pros and Cons

☺ Less sensitive to scheduling

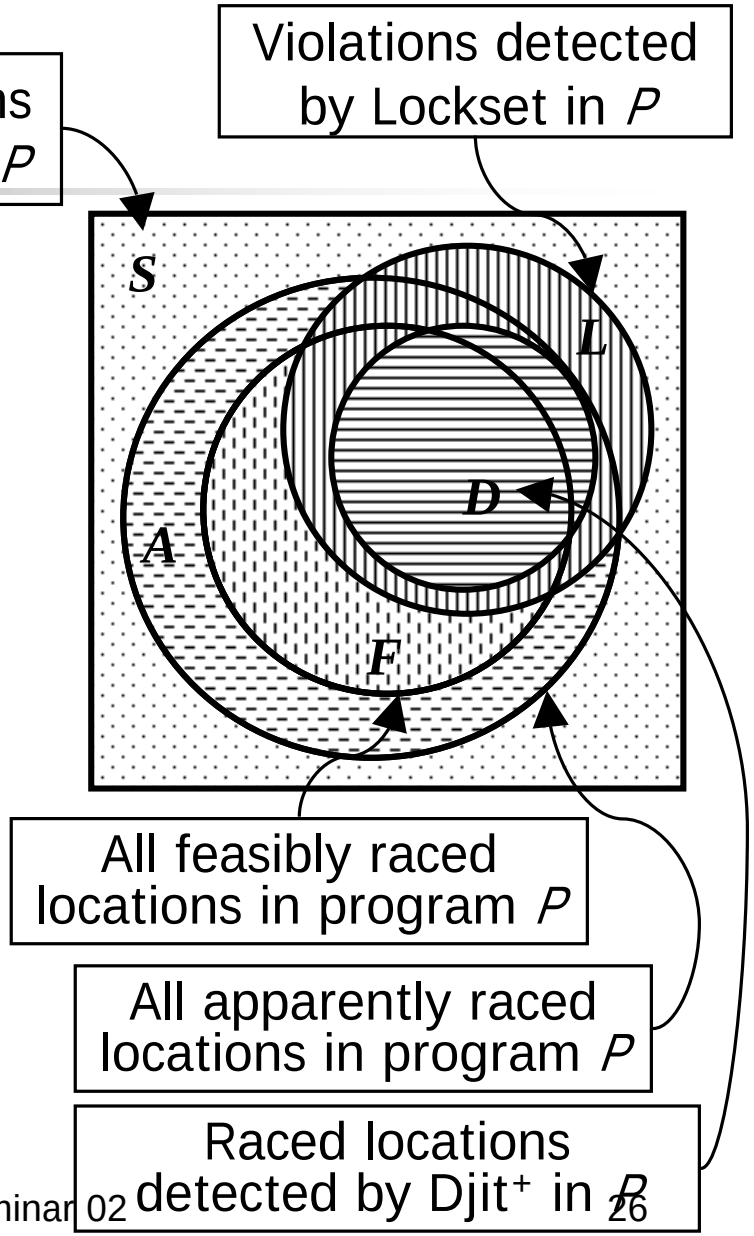
☹ Lots of false alarms

☹ Still dependent on scheduling:

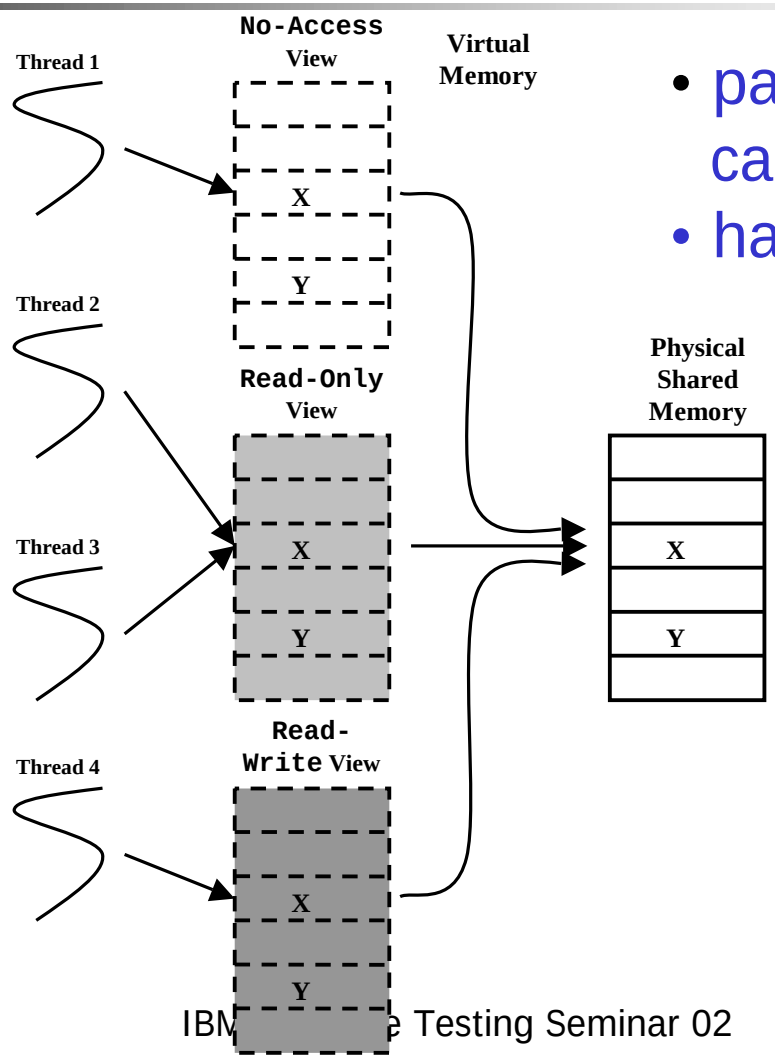
cannot prove tested program is race free

Combining Djit$^+$ and Lockset

- Lockset can detect suspected races in more execution orders.
- Djit$^+$ can filter out the spurious races reported by Lockset.
- The number of checks performed by Djit$^+$ can be reduced using the additional information obtained from Lockset.
- The implementation overhead comes mainly from access logging – can be shared by both algorithms.



Implementing Access Logging: Recording First Accesses



- page faults are caught by MultiRace
- handler is a detector

Swizzling Between Views

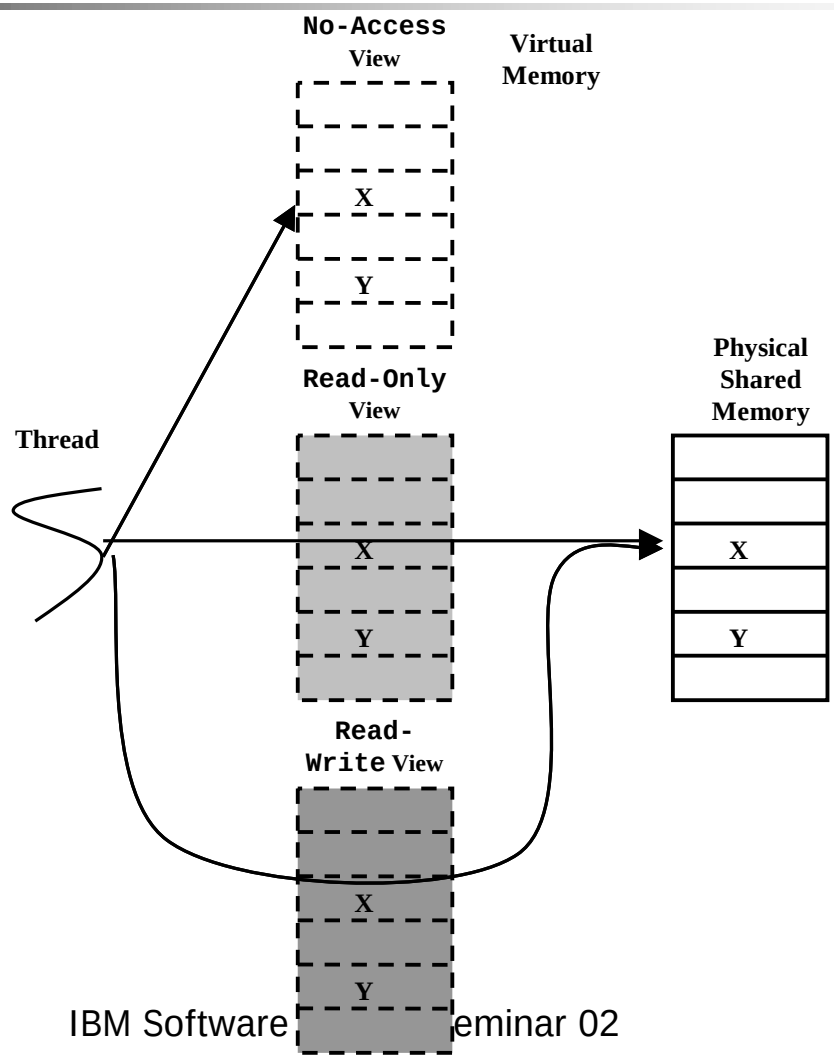
unlock(m)

read x

write x

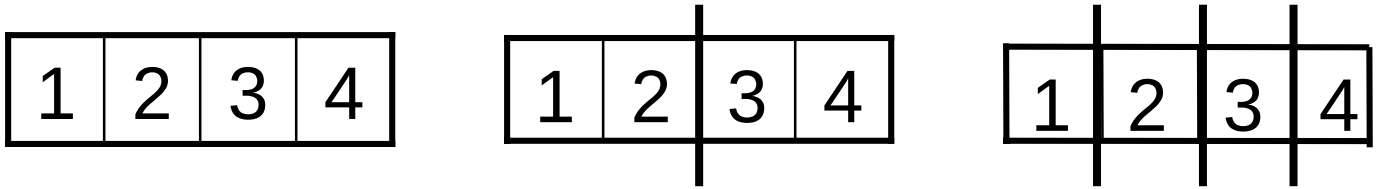
unlock(m)

write x



Detection Granularity

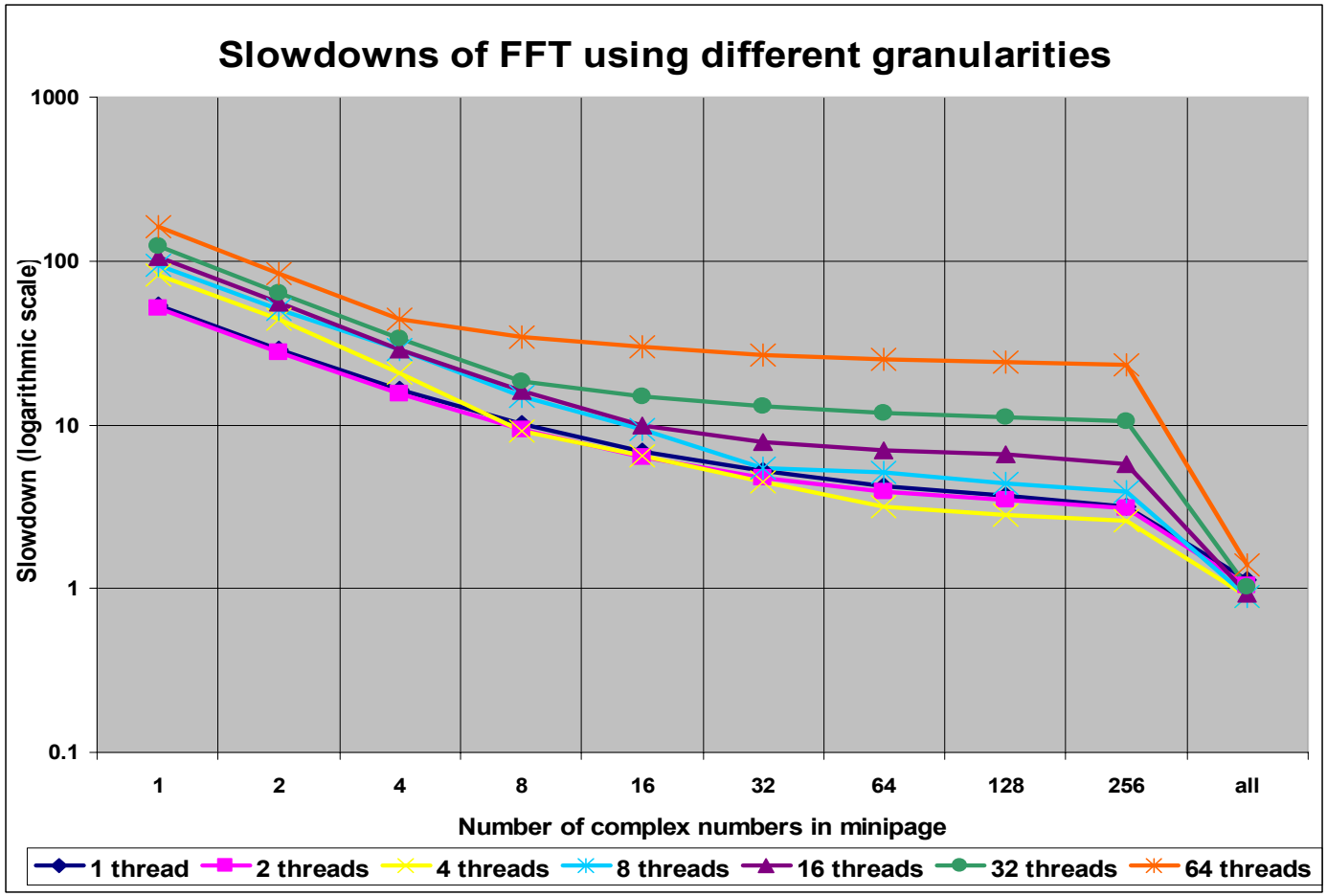
- A minipage (= detection unit) can contain:
  - Objects of primitive types – char, int, double, etc.
  - Objects of complex types – classes and structures
  - Entire arrays of complex or primitive types
- An array can be placed on a single minipage or split across several minipages.
  - Array still occupies contiguous addresses.



Playing with Detection Granularity to Reduce Overhead

- Large minipages ➔ reduced overhead
  - Less faults
- A minipage should be refined into smaller minipages when suspicious alarms occur
  - Replay technology can help (if available)
- When suspicion resolved – regroup
  - May disable detection on the accesses involved

Detection Granularity



Instrumentation (1)

- new and malloc overloaded, get two additional args
  - # requested elements
  - # elements per minipage

```
Type* ptr = new(50, 1) Type[50];
```

- Every class Type inherits from SmartProxy<Type> template class.

```
class Type : public SmartProxy<Type> {
  ... }
```

- The functions of SmartProxy<Type> class return a pointer or a reference to the instance through its correct view.

Instrumentation (2)

- All occurrences of potentially shared primitive types are wrapped into fully functional classes:

```
int → class _int_ : public SmartProxy<int> {
            int val;...}
```

- Global and static objects are copied to our shared space during initialization.
- No source code – the objects are 'touched':

```
memcpy( dest->write(n),src->read(n),
                                n*sizeof(Type) )
```

- All accesses to class data members are instrumented in the member functions:

```
data_mmbr=0; → smartPointer()->data_mmbr=0;
```

Example of Instrumentation

```
void func( Type* ptr, Type& ref, int num ) {
   for ( int i = 0; i < num; i++ ) {
     ptr->smartPointer()->data +=
              ref.smartReference().data;
     ptr++;
   }
   Type* ptr2 = new(20, 2) Type[20];
   memset( ptr2->write(20), 0, 20*sizeof(Type) );
   ptr = &ref;
   ptr2[0].smartReference() = *ptr->smartPointer();
   ptr->member_func( );
}
```

The desired value is specified by user through source code annotation

No Change!!!

Loop Optimizations

- **Original code:**

```
for ( i = 0; i < num; i++)
    arr[i].data = i;
```

- **Instrumented code:**

```
for ( i = 0; i < num; i++)
    arr[i].smartReference().data = i;  ← Very expensive code
```

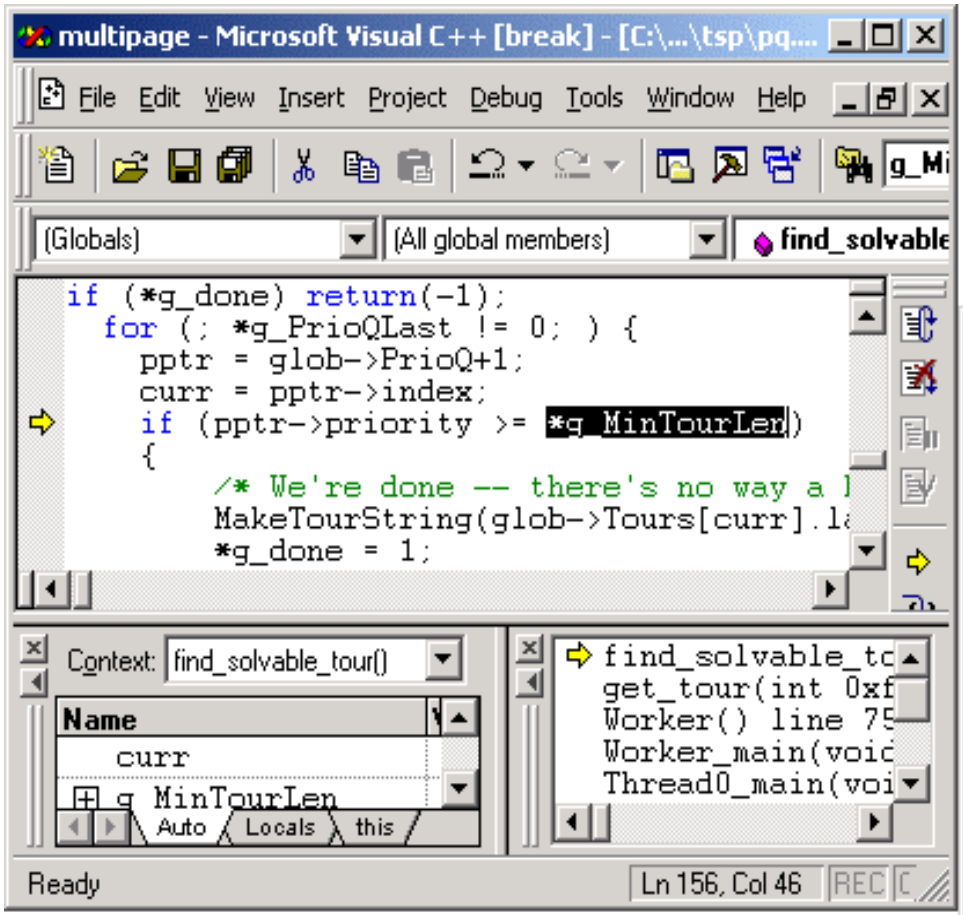
- **Optimized code (entire array on single minipage):**

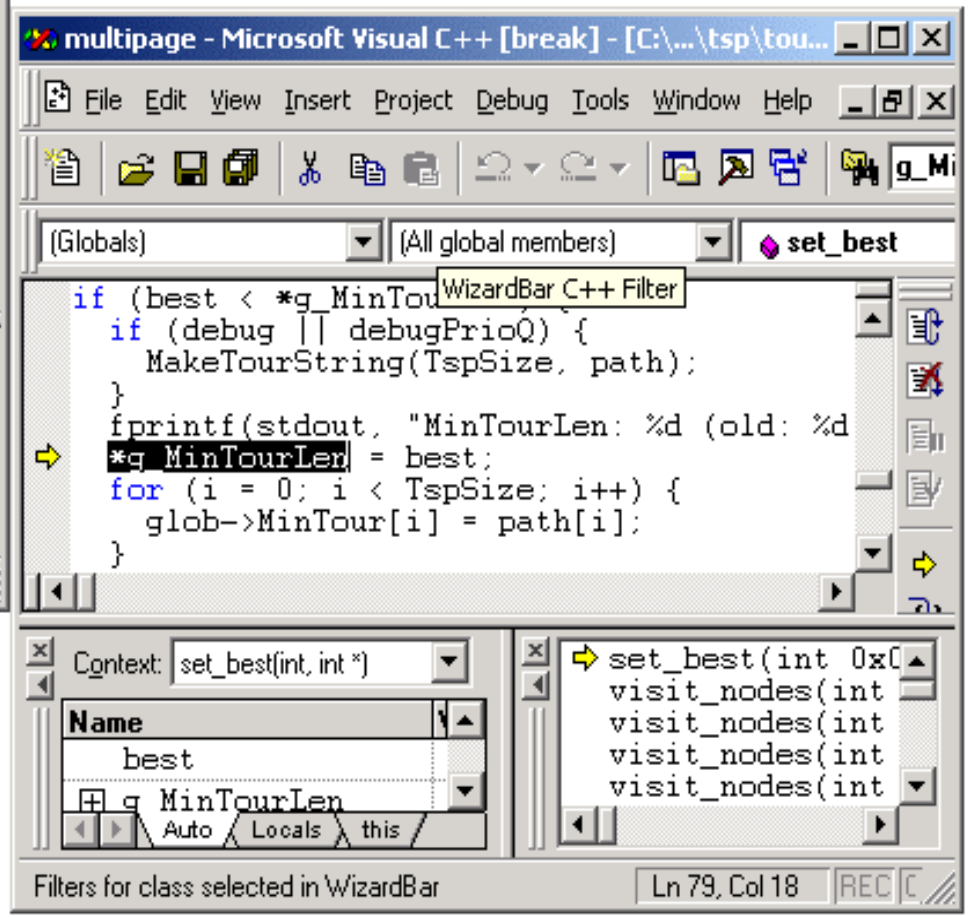
```
if ( num > 0 ) arr[0].smartReference().data = 0;  ← Touch first element
for ( i = 1; i < num; i++)        ← i runs from 1 and not from 0
    arr[i].data = i;              ← Access the rest of array without faults
```

- **Additional optimization (no synchronization in loop):**

```
arr.write( num );                 ← Touch for writing all minipages of array
for ( i = 0; i < num; i++)        ← Efficient if number of elements in array
    arr[i].data = i;                is high and number of minipages is low
```

Reporting Races in MultiRace



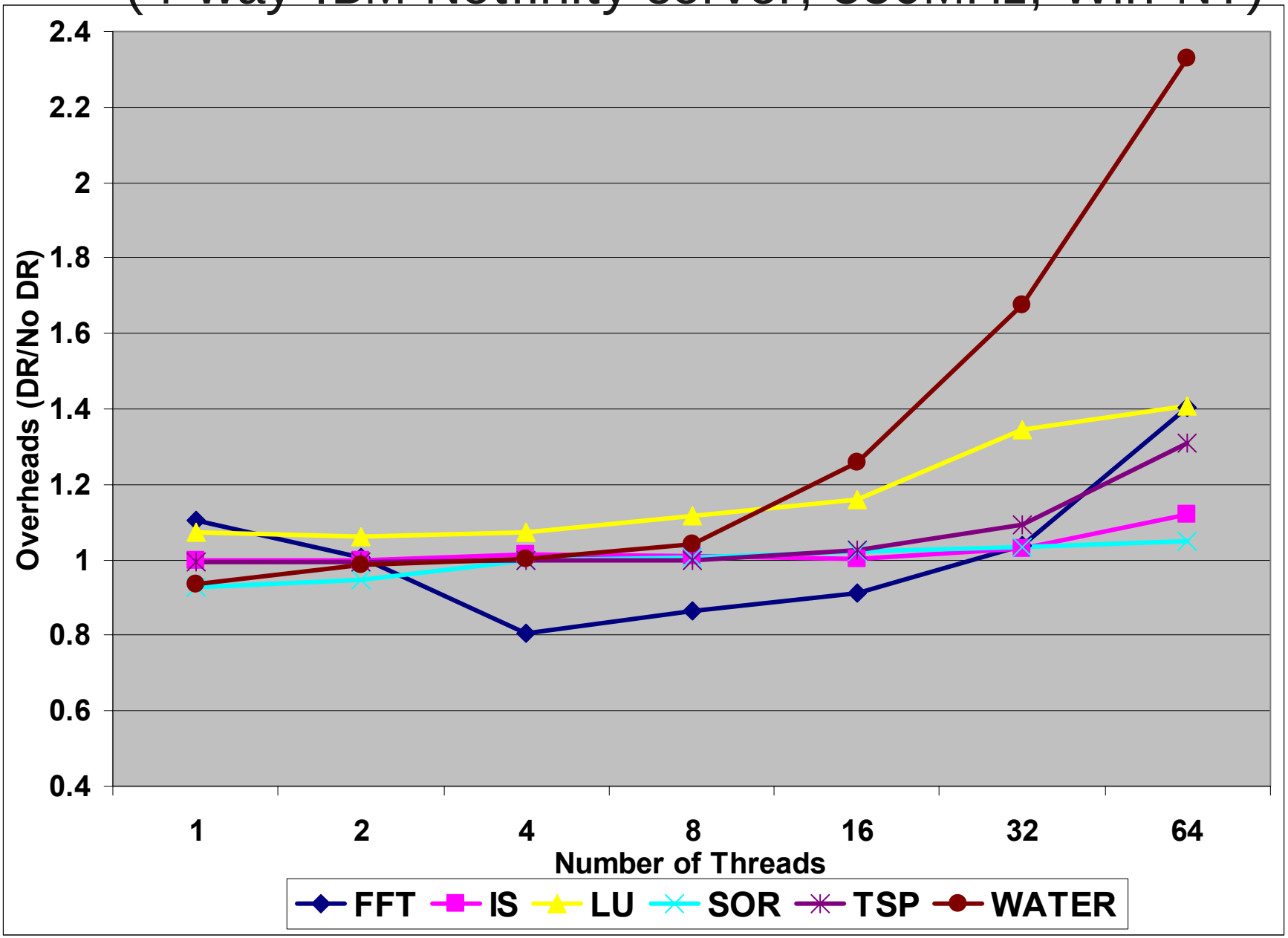


Benchmark Specifications (2 threads)

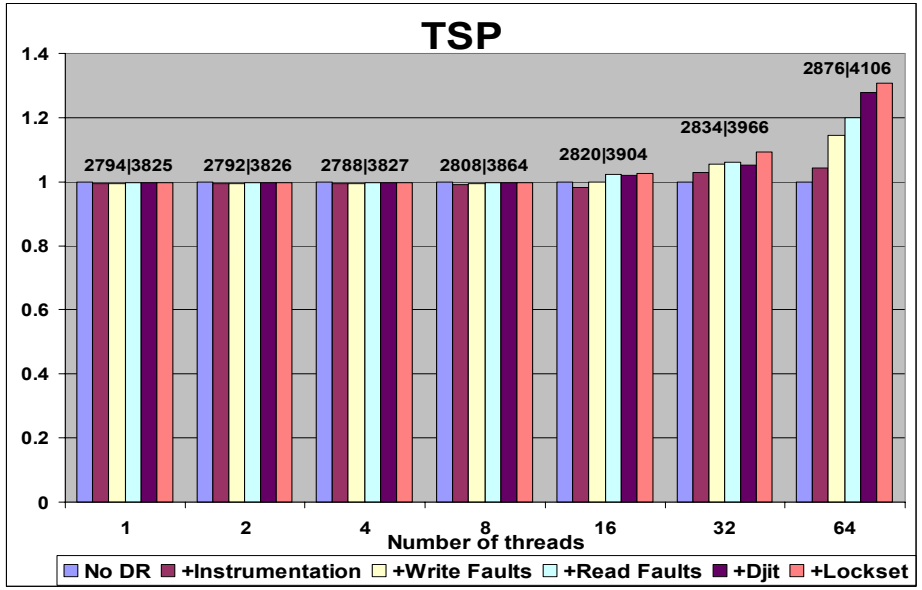
	Input Set	Shared Memory	# Mini-pages	# Write/ Read Faults	# Time-frames	Time in sec (NO DR)
FFT	2^8*2^8	3MB	4	9/10	20	0.054
IS	2^{23} numbers 2^{15} values	128KB	3	60/90	98	10.68
LU	1024*1024 matrix, block size 32*32	8MB	5	127/186	138	2.72
SOR	1024*2048 matrices, 50 iterations	8MB	2	202/200	206	3.24
TSP	19 cities, recursion level 12	1MB	9	2792/ 3826	678	13.28
WATER	512 molecules, 15 steps	500KB	3	15438/ 15720	15636	9.55

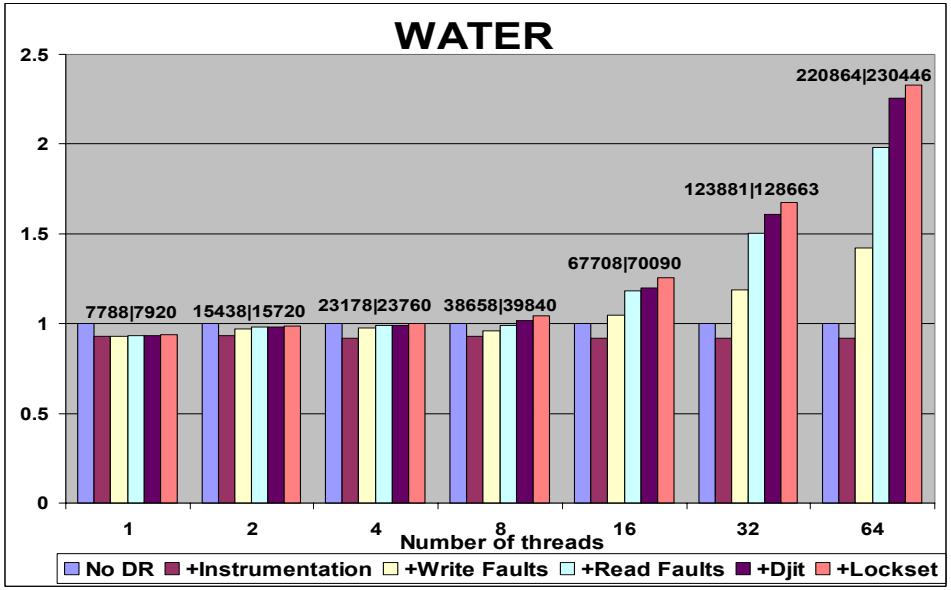
Benchmark Overheads

(4-way IBM Netfinity server, 550MHz, Win-NT)



Overhead Breakdown





- Numbers above bars are # write/read faults.
- Most of the overhead come from page faults.
- Overhead due to detection algorithms is small.

Summary
MultiRace is:

- Transparent
- Supports two-way and global synchronization primitives: locks and barriers
- Detects races that actually occurred (Djit$^+$)
- Usually does not miss races that could occur with a different scheduling (Lockset)
- Correct for weak memory models
- Imposes much lower overhead than existing tools
- Exhibits variable detection granularity

Conclusions

- MultiRace makes it easier for programmer to trust his programs
  - No need to add synchronization “just in case”
- In case of doubt - MultiRace should be activated each time the program executes

Future work

- Implement instrumenting pre-compiler
- Higher transparency
- Higher scalability
- Automatic dynamic granularity
- Integrate with scheduling-generator
- Integrate with record/replay
- Integrate with the compiler/debugger
- May get rid of faults and views
- Optimizations through static analysis
- Etc.

The End

Minipages and Dynamic Granularity of Detection

- *Minipage* is a shared location that can be accessed using the approach of views.
- We detect races on minipages and not on fixed number of bytes.
- Each minipage is associated with the access history of Djit$^+$ and Lockset state.
- The size of a minipage can vary.

Implementing Access Logging

- In order to record only the first accesses (reads and writes) to shared locations in each of the time frames, we use the concept of *views*.
- A view is a region in virtual memory.
- Each view has its own protection – NoAccess / ReadOnly / ReadWrite.
- Each shared object in physical memory can be accessed through each of the three views.
- Helps to distinguish between reads and writes.
- Enables the realization of the dynamic detection unit and avoids false sharing problem.

Disabling Detection

- Obviously, Lockset can report false alarms.
- Also Djit^{+} detects apparent races that are not necessarily feasible races:
  - Intentional races
  - Unrefined granularity
  - Private synchronization
- Detection can be disabled through the use of source code annotations.

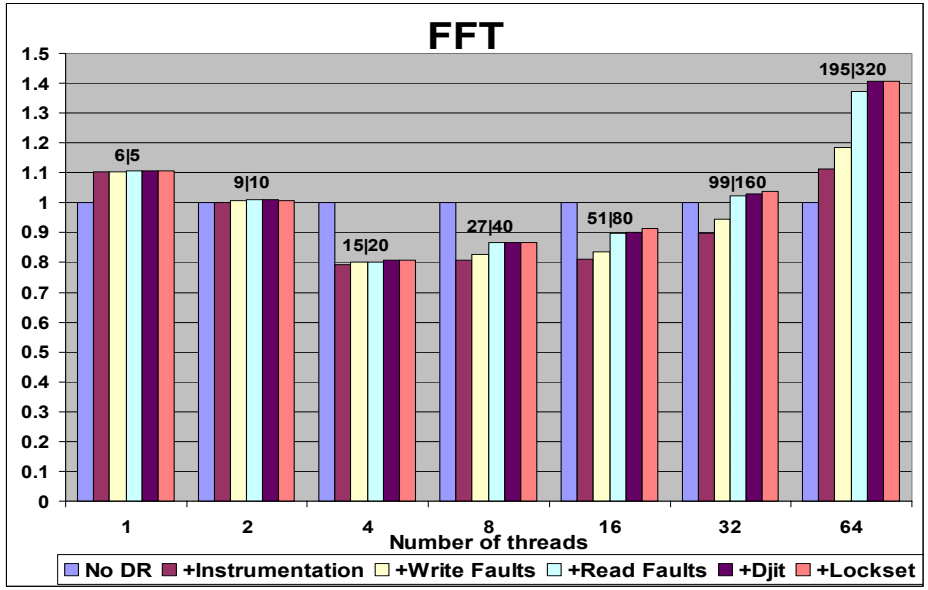
Overheads

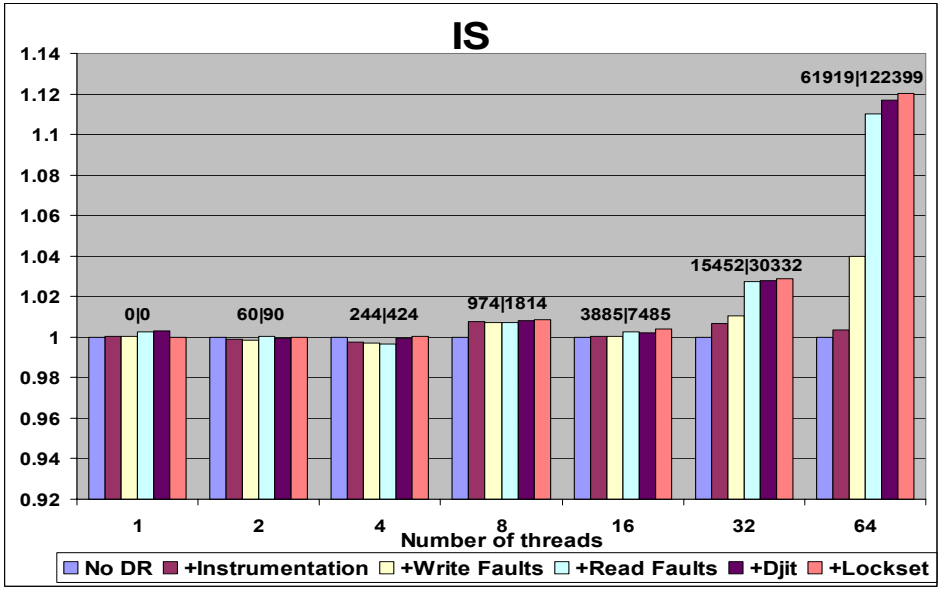
- The overheads are steady for 1-4 threads – we are scalable in number of CPUs.
- The overheads increase for high number of threads.
- Number of page faults (both read and write) increases linearly with number of threads.
- In fact, any on-the-fly tool for data race detection will be unscalable in number of threads when number of CPUs is fixed.

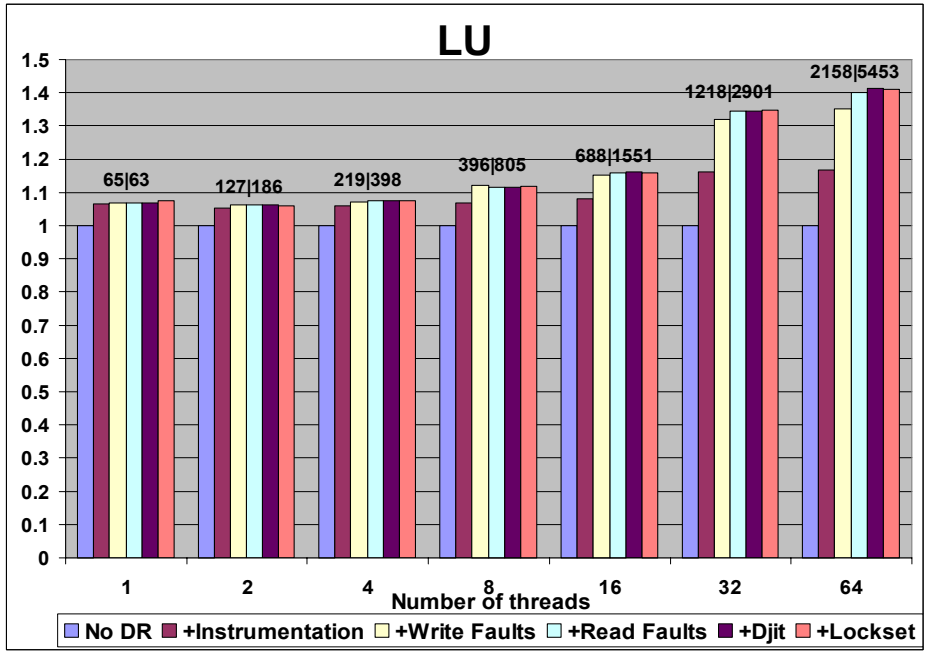
Instrumentation Limitations

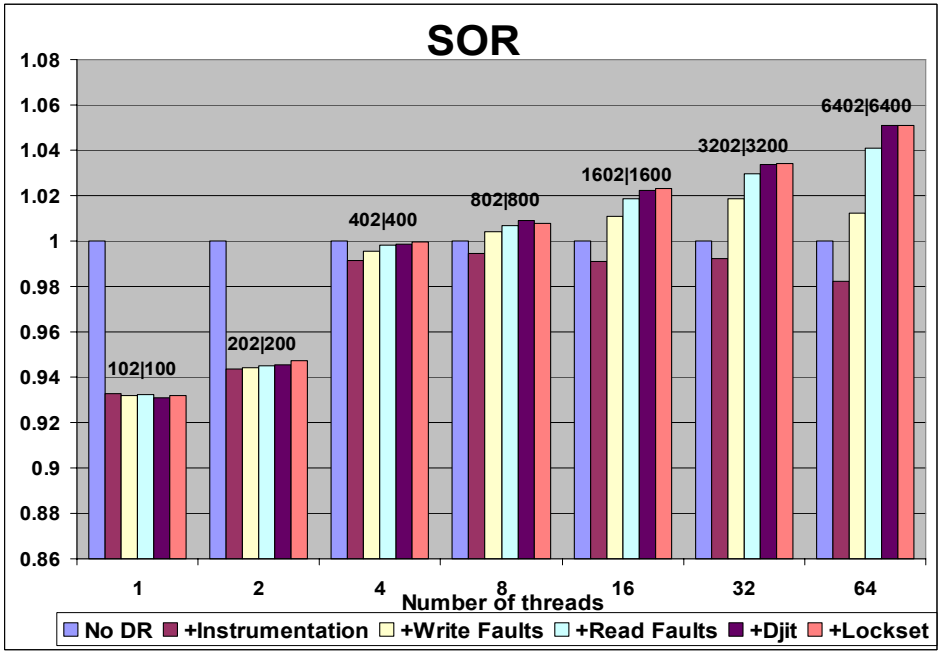
- Currently, no transparent solution for instrumenting global and static pointers.
  - In order to monitor all accesses to these pointers they should be wrapped into classes → compiler's automatic pointer conversions are lost.
  - Will not be a problem in Java.
- All data members of the same instance of class always reside on the same minipage.
  - In the future – will split classes dynamically.

Breakdowns of Overheads









References

- T. Brecht and H. Sandhu. The Region Trap Library: Handling traps on application-defined regions of memory. In *USENIX Annual Technical Conference, Monterey, CA*, June 1999.
- A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordechai. Towards Integration of Data Race Detection in DSM System. In *The Journal of Parallel and Distributed Computing (JPDC)*, 59(2): pp. 180-203, Nov. 1999
- L. Lamport. Time, Clock, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, 21(7): pp. 558-565, Jul. 1978
- F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel & Distributed Algorithms*, pp. 215-226, 1989.

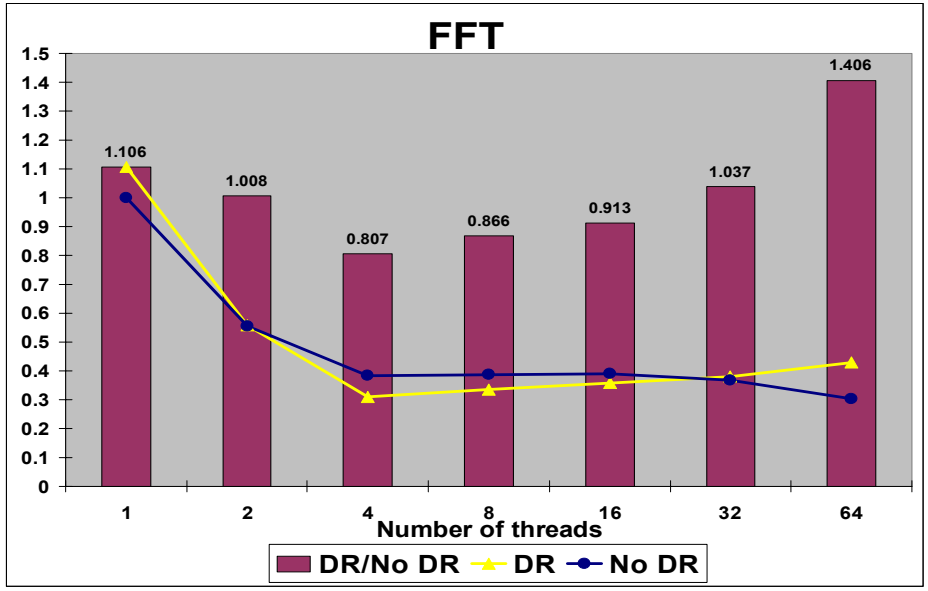
References Cont.

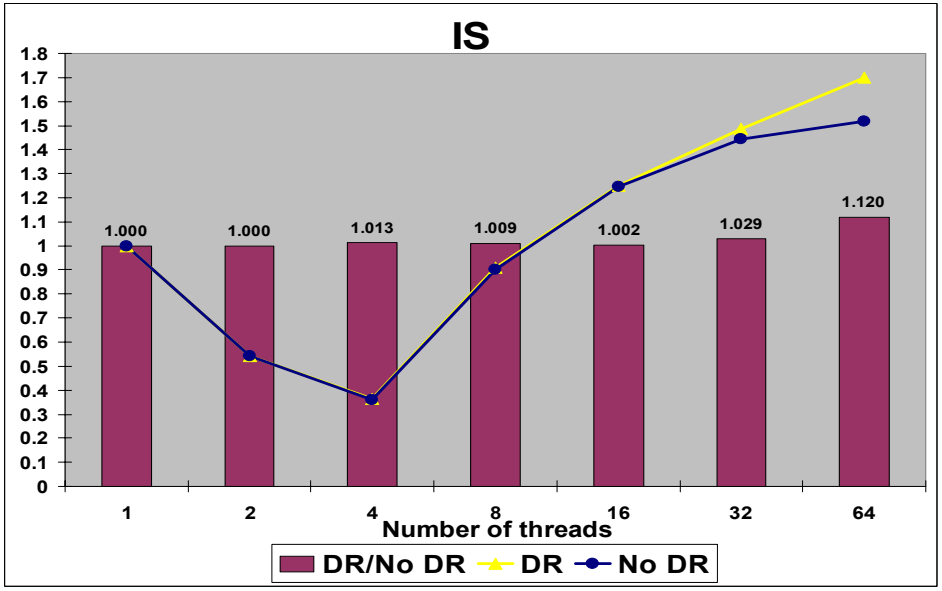
- R. H. B. Netzer and B. P. Miller. What Are Race Conditions? Some Issues and Formalizations. In *ACM Letters on Programming Languages and Systems*, 1(1): pp. 74-88, Mar. 1992.
- R. H. B. Netzer and B. P. Miller. On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. In *1990 International Conference on Parallel Processing*, 2: pp. 93-97, Aug. 1990
- R. H. B. Netzer and B. P. Miller. Detecting Data Races in Parallel Program Executions. In *Advances in Languages and Compilers for Parallel Processing,* MIT Press 1991, pp. 109-129.

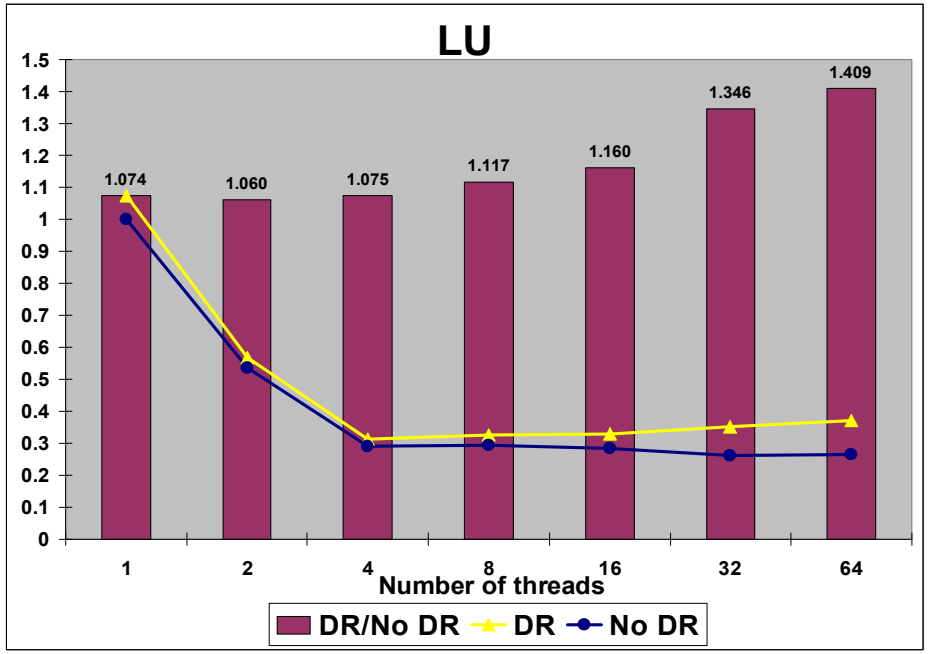
References Cont.

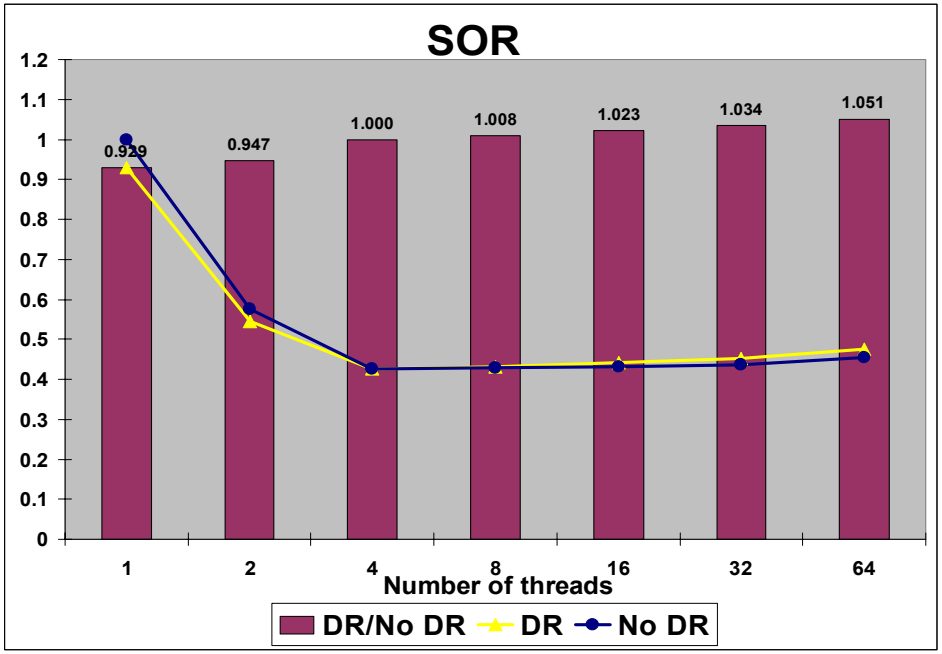
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In *ACM Transactions on Computer Systems*, 15(4): pp. 391-411, 1997
- E. Pozniansky. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. *Research Thesis*.
- O. Zeev-Ben-Mordehai. Efficient Integration of On-The-Fly Data Race Detection in Distributed Shared Memory and Symmetric Multiprocessor Environments. *Research Thesis*, May 2001.

Overheads

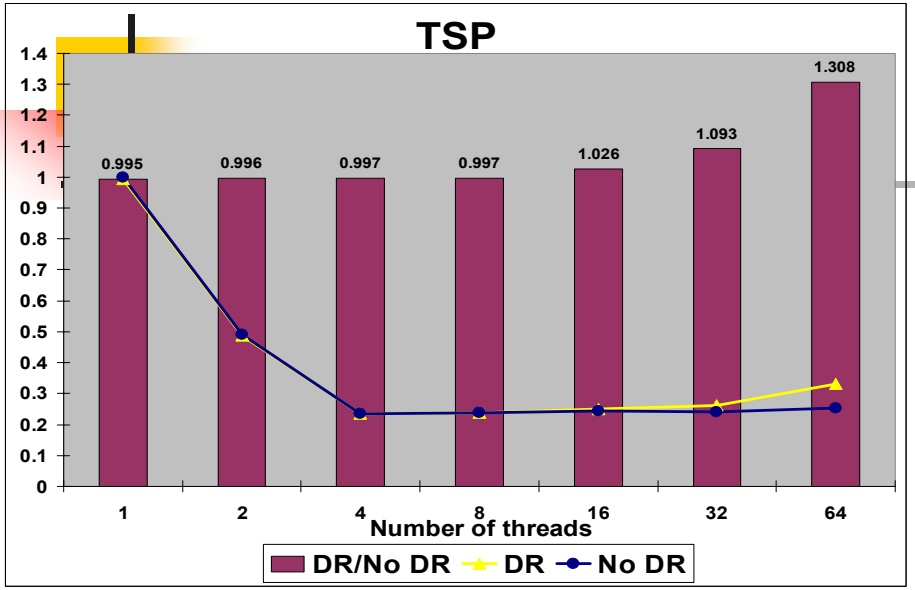

FFT
1.106
1.008
0.807
0.866
0.913
1.037
1.406
1 2 4 8 16 32 64
Number of threads
DR/No DR
DR
No DR

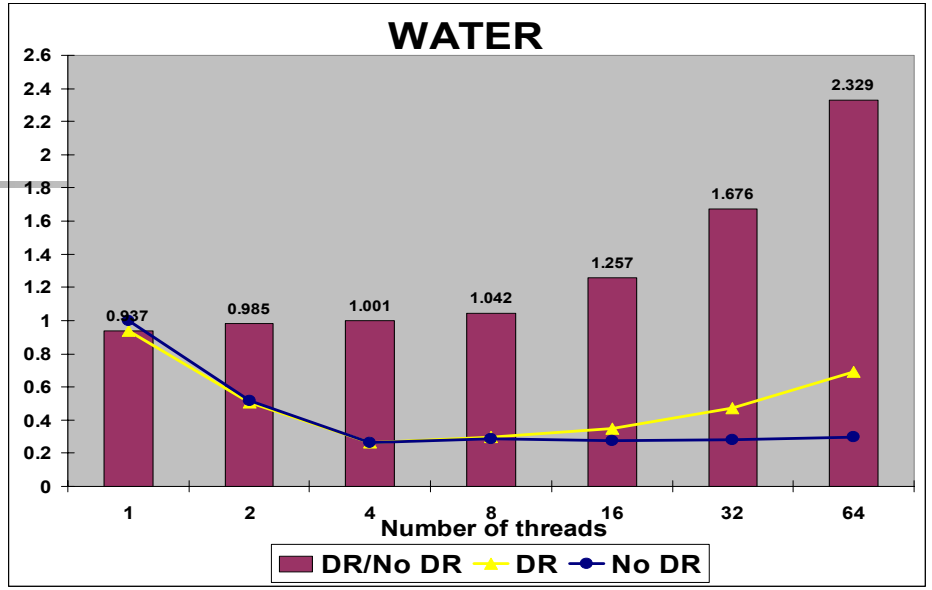

IS
1.000
1.000
1.013
1.009
1.002
1.029
1.120
1 2 4 8 16 32 64
Number of threads
DR/No DR
DR
No DR


LU
1.074
1.060
1.075
1.117
1.160
1.346
1.409
1 2 4 8 16 32 64
Number of threads
DR/No DR
DR
No DR


SOR
0.929
0.947
1.000
1.008
1.023
1.034
1.051
1 2 4 8 16 32 64
Number of threads
DR/No DR
DR
No DR

Overheads





- The testing platform:
  - 4-way IBM Netfinity, 550 MHz
  - 2GB RAM
  - Microsoft Windows NT