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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X++</td>
<td>T=Y</td>
</tr>
<tr>
<td>Z=2</td>
<td>T=X</td>
</tr>
</tbody>
</table>
How to prevent Data Races?

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

Thread 1
- \textbf{Lock}(m)
- \textbf{Unlock}(m)

Thread 2
- \( T=X \)
- \textbf{Lock}(m)
- \textbf{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 \((#\text{threads})^{\text{thread\_length}}\)
- # inputs
- Detection-code’s side-effects
- Weak memory, instruction reorder, atomicity
Where is Waldo?

```c
#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 );
}
```

Similar problem found in Java.util.vector
Apparent Data Races

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

Initially: grades = oldDatabase; updated = \texttt{false};

Thread \textit{T.A.}

\begin{align*}
\text{grades} &= \text{newDatabase}; \\
\text{updated} &= \texttt{true};
\end{align*}

Thread \textit{Lecturer}

\begin{align*}
\textbf{while} \ (\text{updated} == \texttt{false}); \\
X &= \text{grades.gradeOf(lecturersSon)};
\end{align*}

- Exist iff “real” data race exist 😊
- Detection is still NP-hard 😞
Detection Approaches

- **Restricted programming 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:**
- programming 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 $a \xrightarrow{\text{hb}} b$ nor $b \xrightarrow{\text{hb}} a$
  - $\Rightarrow$ 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.

<table>
<thead>
<tr>
<th>Thread</th>
<th>LTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>1</td>
</tr>
<tr>
<td>lock( m1 )</td>
<td>1</td>
</tr>
<tr>
<td>z = 2</td>
<td>1</td>
</tr>
<tr>
<td>lock( m2 )</td>
<td>1</td>
</tr>
<tr>
<td>y = 3</td>
<td>1</td>
</tr>
<tr>
<td>unlock( m2 )</td>
<td>2</td>
</tr>
<tr>
<td>z = 4</td>
<td>2</td>
</tr>
<tr>
<td>unlock( m1 )</td>
<td>3</td>
</tr>
<tr>
<td>x = 5</td>
<td>3</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Thread</th>
<th>LTF</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>1</td>
</tr>
<tr>
<td>lock( m1 )</td>
<td></td>
</tr>
<tr>
<td>z = 2</td>
<td>1</td>
</tr>
<tr>
<td>lock( m2 )</td>
<td></td>
</tr>
<tr>
<td>y = 3</td>
<td>1</td>
</tr>
<tr>
<td>unlock( m2 )</td>
<td></td>
</tr>
<tr>
<td>z = 4</td>
<td>2</td>
</tr>
<tr>
<td>unlock( m1 )</td>
<td></td>
</tr>
<tr>
<td>x = 5</td>
<td>3</td>
</tr>
</tbody>
</table>
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**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>write $x$</td>
<td>(1 1 1)</td>
<td>(1 1 1)</td>
</tr>
<tr>
<td>unlock( $m_1$ )</td>
<td>(2 1 1)</td>
<td>unlock( $m_1$ )</td>
</tr>
<tr>
<td>read $z$</td>
<td>(2 1 1)</td>
<td>read $y$</td>
</tr>
</tbody>
</table>
Djit$^+$

Checking Concurrency

$P(a,b) \sqcap (a\text{.type} = \text{write} \sqcap b\text{.type} = \text{write}) \sqcap$

$\sqcap (a\text{.time}\_\text{frame} \geq st_{b\text{.thread}\_id[a\text{.thread}\_id]}$)

- $a$ 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.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock( m )</td>
<td>lock( m )</td>
</tr>
<tr>
<td>write X</td>
<td>write X</td>
</tr>
<tr>
<td>read X</td>
<td>read X</td>
</tr>
<tr>
<td>unlock( m )</td>
<td>unlock( m )</td>
</tr>
</tbody>
</table>

No logging
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.
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
- $\textit{locks\_held}(t)$ 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 $t$:
  - $lh_v \leftarrow \textit{locks\_held}(t)$
  - if it is a read, then $lh_v \leftarrow lh_v \cap \{\textit{readers\_lock}\}$
  - $C(v) \leftarrow C(v) \cap lh_v$
  - if $C(v) = \emptyset$, issue a warning
**Lockset Example**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>$lh_v$</th>
<th>$C(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock (m1)</td>
<td></td>
<td>{}</td>
<td>{m1, m2, r_l}</td>
</tr>
<tr>
<td>read v</td>
<td></td>
<td>{m1, r_l}</td>
<td>{m1, r_l}</td>
</tr>
<tr>
<td>unlock (m1)</td>
<td></td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock (m2)</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write v</td>
<td>{m2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock (m2)</td>
<td>{}</td>
<td></td>
</tr>
</tbody>
</table>

$r_{l} = readers_{lock}$ prevents from multiple reads to generate false alarms.

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.

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>$lh_v$</th>
<th>$C(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(m1)</td>
<td></td>
<td>{ }</td>
<td>{m1, m2}</td>
</tr>
<tr>
<td>read v</td>
<td></td>
<td>{m1}</td>
<td>{m1}</td>
</tr>
<tr>
<td>unlock(m1)</td>
<td></td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock(m2)</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write v</td>
<td>{m2}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(m2)</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

Warning: locking Discipline for $v$ is Violated
**Lockset vs. Djit⁺**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y++[^1]</code></td>
<td><code>lock( m )</code></td>
</tr>
<tr>
<td><code>v++</code></td>
<td><code>v++</code></td>
</tr>
<tr>
<td><code>unlock( m )</code></td>
<td><code>unlock( m )</code></td>
</tr>
<tr>
<td></td>
<td><code>y++[^2]</code></td>
</tr>
</tbody>
</table>

[^1] 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: $\text{Lock}\_a\_a(v) \subseteq \text{Lock}\_b\_b(v)$
  - $\text{Lock}\_u\_u(v)$ set of real locks held during access $u$ to $v$
- It follows that:
  - $[C(v) \cap \text{Lock}\_a\_a(v)] \subseteq [C(v) \cap \text{Lock}\_b\_b(v)]$
  - If $C(v) \cap \text{Lock}\_a\_a(v) \neq \emptyset$ then
    - $C(v) \cap \text{Lock}\_b\_b(v) \neq \emptyset$

$\Rightarrow$ Only first accesses need be checked in every time frame
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\textsuperscript{+} and Lockset

- Lockset can detect suspected races in more execution orders.
- Djit\textsuperscript{+} can filter out the spurious races reported by Lockset.
- The number of checks performed by Djit\textsuperscript{+} 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 $\rightarrow$ 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

Slowdowns of FFT using different granularities

- Number of complex numbers in minipage
- Slowdown (logarithmic scale)
- Thread count: 1, 2, 4, 8, 16, 32, 64, all

IBM Software Testing Seminar 02
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:
  ```cpp
  int \rightarrow \text{class } _\text{int}_ : \text{public } \text{SmartProxy<int> } \{ \\
  \text{int } \text{val}; \ldots \}
  ```

- Global and static objects are copied to our shared space during initialization.

- No source code – the objects are ‘touched’:
  ```cpp
  \text{memcpy}( \text{dest->write(n)}, \text{src->read(n)}, \text{n*sizeof(Type)} )
  ```

- All accesses to class data members are instrumented in the member functions:
  ```cpp
  \text{data_mmbr}=0; \rightarrow \text{smartPointer()\rightarrow data_mmbr}=0;
  ```
Example of Instrumentation

```c++
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.
Loop Optimizations

- **Original code:**
  ```java
  for ( i = 0; i < num; i++)
    arr[i].data = i;
  ```

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

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

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

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

FFT

IS

LU

SOR

Number of threads

No DR +Instrumentation +Write Faults +Read Faults +Djit +Lockset

Number of threads

No DR +Instrumentation +Write Faults +Read Faults +Djit +Lockset

Number of threads

No DR +Instrumentation +Write Faults +Read Faults +Djit +Lockset

Number of threads

No DR +Instrumentation +Write Faults +Read Faults +Djit +Lockset
References


References Cont.


References Cont.


Overheads

**FFT**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>DR/No DR</th>
<th>DR</th>
<th>No DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.807</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.866</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.913</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.037</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.406</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**IS**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>DR/No DR</th>
<th>DR</th>
<th>No DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.013</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.029</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.120</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LU**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>DR/No DR</th>
<th>DR</th>
<th>No DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.074</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.075</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.346</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.409</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SOR**

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>DR/No DR</th>
<th>DR</th>
<th>No DR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.932</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.947</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1.008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.023</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1.034</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.051</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The testing platform:

- 4-way IBM Netfinity, 550 MHz
- 2GB RAM
- Microsoft Windows NT