Commutativity Race Detection

Abstract

This paper introduces the concept of a commutativity race. A commutativity race occurs in a given execution when two library method invocations can happen concurrently yet they do not commute. Commutativity races are an elegant concept enabling reasoning about concurrent interaction at the library interface. We present a dynamic commutativity race detector. Our technique is based on a novel combination of vector clocks and a structural representation automatically obtained from a commutativity specification. Conceptually, our work can be seen as generalizing classical read-write race detection.

We also present a new logical fragment for specifying commutativity conditions. This fragment is expressive, yet guarantees a constant number of comparisons per method invocation rather than linear with unrestricted specifications.

We implemented our analyzer and evaluated it on real-world applications. Experimental results indicate that our analysis is practical: it discovered harmful commutativity races with overhead comparable to state-of-the-art, low-level race detectors.

1. Introduction

In response to the growing complexity of software, common patterns are increasingly being encapsulated into thread-safe libraries (e.g., Java collections). At the same time, virtually all modern applications use some form of concurrency and heavily rely on various libraries to accomplish their objectives. This raises a key challenge: even though a library is implemented correctly, concurrent threads can invoke its operations in a way which causes undesirable interference at the library interface level, leading to incorrect program behaviors. We observe that such errors are fundamentally caused when concurrent threads perform non-commutative library operations. We refer to this phenomenon as a commutativity race. To illustrate the concept, consider the following simple program:

\[
\begin{align*}
T_1 & = 1. fork; \\
T_2 & = 2. m.put(5, 7); \\
T_3 & = 3. int \text{ v.m.get}(5); \\
T_4 & = 4. m.put(5, 7); \\
\end{align*}
\]

Initially, the concurrent hashmap has all keys initialized to the value 1. Here, we have a commutativity race between \( m\).put(5, 7) and \( m\).get(5): the two operations do not commute and they can happen in any order. As a result of this commutativity race, the values returned by \( m\).get(5) differ in each of the two possible executions, returning \( 7 \) if \( m\).put(5, 7) occurred before \( m\).get(5) in the execution or returning \( 5 \) otherwise. Such non-determinism can sometimes lead to undesirable behaviors further in the execution. In fact, in Java, if \( v \) was initially empty, and \( m\).execute before \( m\).put(5, 7), the program will throw a NullPointerException when unboxing the result from \( m\).get(5). Of course, not all commutativity races are harmful, however, the presence of a commutativity race may indicate undesirable interference.

Commutativity race detection. In this paper, we introduce the concept of a commutativity race and present a dynamic commutativity race detector with the appropriate formal guarantees. Our approach is parametric on a declarative commutativity specification and can be used to analyze any library equipped with such a specification. The analysis is based on a novel combination of commutativity information with classic vector clocks [12, 14]. Conceptually, our work can be seen as a generalization of traditional data race detection (e.g., [31]) to deal with much richer and abstract notions of conflict, beyond the level of basic reads and writes.

We introduce an expressive logical fragment (ECL) for capturing commutativity conditions as well as an automatic translation procedure from ECL to a structural representation used by the commutativity race detector. ECL admits practical specifications (e.g., maps, sets) yet is amenable to efficient analysis: any commutativity condition expressed in ECL can be checked by the analysis via a constant number of comparisons per method invocation as opposed to linear with unrestricted specifications. We believe the results in this paper are of interest beyond race detection: the structural representation, ECL, and the translation procedure between them can serve as a basis for other concurrency analyzers as well as for optimistic concurrency schemes (e.g., transactional memory).

Contributions. The main contributions of this paper are:

- A dynamic commutativity race detector based on a novel combination of classic vector clocks for tracking the happens-before relation with a structural representation of a commutativity specification (Section 5). Our approach generalizes traditional race detection to richer, more abstract notions of conflict.
- A logical fragment, called ECL, which captures useful commutativity conditions (e.g., maps), yet allows the analysis to perform a constant number of operations per method invocation, as opposed to linear with arbitrary specifications (Section 6.1).
- A translation procedure from ECL to the structural representation used by the analysis (Section 6.2).
- An implementation and an evaluation of our approach on two industrial Java applications (Section 7). Our results indicate that the performance overhead is acceptable and further, our tool discovered harmful commutativity races in these applications when using ConcurrentHashMap.
My Killer App

Intel Core 2 Extreme quad-core

Intel Core 2 Extreme quad-core
My Killer App

- Agent 1
- Agent 2
- Agent 3
- Agent 4

Concurrent HashMap
Concurrent Queue
Concurrent List
What happens here?

$T_1$:  
1: fork $T_2$;  
2: m.$\text{put}(5, 7)$

$T_2$:  
1: m.$\text{get}(5)$

Why is this a conflict?
What happens here?

T₁:
1: fork T₂;
2: m.put(5, 7);

T₂:

m.get(5);

1: fork T₂;
2: m.put(5, 7);
3: m.get(5);

Why is this a conflict?

returns 7

{5 ↦ 1} m.put(5, 7);
{5 ↦ 1} m.get(5);
{5 ↦ 7}

returns 1

{5 ↦ 1} m.put(5, 7);
{5 ↦ 7}
What happens here?

$T_1$: 
1: fork $T_2$;  
2: $m$.$\text{put}(5,7)$

$T_2$: 
1: 
2: $m$.$\text{put}(5,2)$

Why is this a conflict?

\[
\begin{array}{c}
\{5 \mapsto 1\} m.$\text{put}(5,7); m.$\text{put}(5,2) \\
\{5 \mapsto 1\} m.$\text{put}(5,2); m.$\text{put}(5,7) \\
\end{array}
\]

\[
\begin{array}{c}
\{5 \mapsto 2\} \\
\{5 \mapsto 7\} \\
\end{array}
\]

different
What happens here?

T₁:
1: fork T₂;
2: m.put(5, 7)

T₂:
3: m.get(5);

Two operations **commute** iff, when applied in either order, (i) return the same values and (ii) lead to the same final state.

A **commutativity race** is when there are two concurrent operations and:
1. They do not commute
2. They may happen in any order

Lead to nondeterministic behavior
Detect commutativity races automatically.
Another Example

1: var o = dictionary();  // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
4:     fork {
5:         o.put(host, createConnection(host));
6:     }
7: }
8:
9: print o.size() + " connections established";
Another Example

Duplicate hosts?

Multiple connections to same host

```javascript
1: var o = dictionary(); // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
  4:   fork {
  5:     o.put(host, createConnection(host));
  6:   }
  7: }
8: joinall;
9: print(o.size() + " connections established");

τ₃: o.put('a.com', c₁)/nil;
τ₂: o.put('a.com', c₂)/c₁;
```

ok
fail
Our Approach

Step 1: Obtain a commutativity specification
Step 2: Obtain an access point specification
Step 3: Commutativity race detection

Diagram:
- Step 1: commutativity specification
- Step 2: logic-to-access points translator
- Step 3: program trace
- Vector clocks
- Step 3: commutativity race detector
- Report races
- No races found
Our Approach

Step 1: Obtain a commutativity specification
Step 2: Obtain an access point specification
Step 3: Commutativity race detection

For every pair of methods, provide a formula.

\[
\text{o.put}(k_1, v_1)/p_1 \text{ commutes with } \text{o.put}(k_2, v_2)/p_2 \\
\iff \\
k_1 \neq k_2 \lor (v_1 = p_1 \land v_2 = p_2)
\]
Our Approach

Step 1: Obtain a commutativity specification

Step 2: Obtain an access point specification

Step 3: Commutativity race detection

Map actions (concrete instances) to access points

- $o$.put$(k, v) / nil$
- $o$.put$(k, v_2) / p$

- $\langle \chi_o, \eta_o, C_o \rangle$
- $\eta_o$
- $\chi_o$
- $C_o$

value of $k$ has changed
Our Approach

Step 1: Obtain a commutativity specification
Step 2: Obtain an access point specification
Step 3: Commutativity race detection

\[ \langle \chi_o, \eta_o, C_o \rangle \]

\[ \varphi(m,n) \]
Our Approach

Step 1: Obtain a commutativity specification
Step 2: Obtain an access point specification
Step 3: Commutativity race detection

\[ \langle \chi, \eta, C \rangle \]
Review of vector clocks …
Rules:

1. Vector initialized to 0 at each process
   \[ V_i[j] = 0 \text{ for } i, j = 1, \ldots, N \]

2. Process increments its element of the vector in local vector before timestamping event:
   \[ V_i[i] = V_i[i] + 1 \]

3. Message is sent from process \( P_i \) with \( V_i \) attached to it

4. When \( P_j \) receives message, compares vectors element by element and sets local vector to higher of two values
   \[ V_j[i] = \max(V_i[i], V_j[i]) \text{ for } i = 1, \ldots, N \]

For example,
   received: \([0, 5, 12, 1]\), have: \([2, 8, 10, 1]\)
   new timestamp: \([2, 8, 12, 1]\)
Define

\[ V = V' \text{ iff } V[i] = V'[i] \text{ for } i = 1 \ldots N \]
\[ V \leq V' \text{ iff } V[i] \leq V'[i] \text{ for } i = 1 \ldots N \]

For any two events e, e'

if \( e \rightarrow e' \) then \( V(e) < V(e') \)

... just like Lamport’s algorithm

if \( V(e) < V(e') \) then \( e \rightarrow e' \)

Two events are **concurrent** if neither

\[ V(e) \leq V(e') \text{ nor } V(e') \leq V(e) \]
Event		timestamp
a		(1,0,0)
b		(2,0,0)
Event
timestamp
---
a
b
1,0,0
(2,0,0)
c
(2,1,0)
d
(2,2,0)
e
f

Diagram:

- P1: a → b
- P2: c → d
- P3: e → f

Event timestamps:
- a: (1,0,0)
- b: (2,0,0)
- c: (2,1,0)
- d: (2,2,0)
Event       timestamp
----------   ------------
a            (1,0,0)
b            (2,0,0)
c            (2,1,0)
d            (2,2,0)
e            (0,0,1)
Event | timestamp
----- | ---------
a      | (1,0,0)  
b      | (2,0,0)  
c      | (2,1,0)  
d      | (2,2,0)  
e      | (0,0,1)  
f      | (2,2,2)  

concurrent events
• A “vector” can be an list of tuples:
  – For processes $P_1, P_2, P_3, \ldots$
  – Each process has a globally unique Process ID
    • e.g., MAC address: PID
  – Each process maintains its own timestamp: $T_{P_1}, T_{P_2}, \ldots$
  – Vector: $\{ <P_1, T_{P_1}>, <P_2, T_{P_2}>, <P_3, T_{P_3}>, \ldots \}$

• Any one process may have only partial knowledge of others
  – New timestamp for a received message:
    • Compare all matching sets of process IDs: set to highest of values
    • Any non-matched $<P, T>$ sets get added to the timestamp
  – For a happened-before relation:
    • At least one set of process IDs must be common to both timestamps
    • Match all corresponding $<P, T>$ sets: $A: <P_i, T_a>$, $B: <P_i, T_b>$
    • If $T_a \leq T_b$ for all common processes $P$, then $A \rightarrow B$
\[ \langle \chi_0, \eta_0, C_0 \rangle \]

Concurrent HashMap
Concurrent Queue
Concurrent Agent
My Killer App

Observe traces

<table>
<thead>
<tr>
<th>Event</th>
<th>Modifications of auxiliary state</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e = \tau : fork(u) )</td>
<td>( T(u) \leftarrow \text{inc}_u(T(\tau)) )</td>
</tr>
<tr>
<td></td>
<td>( T(\tau) \leftarrow \text{inc}_\tau(T(\tau)) )</td>
</tr>
<tr>
<td>( e = \tau : join(u) )</td>
<td>( T(\tau) \leftarrow T(\tau) \cup T(u) )</td>
</tr>
<tr>
<td>( e = \tau : acq(l) )</td>
<td>( T(\tau) \leftarrow T(\tau) \cup L(l) )</td>
</tr>
<tr>
<td>( e = \tau : rel(l) )</td>
<td>( L(l) \leftarrow T(\tau) )</td>
</tr>
<tr>
<td></td>
<td>( T(\tau) \leftarrow \text{inc}_\tau(T(\tau)) )</td>
</tr>
</tbody>
</table>

\( e = \tau : o.m(\bar{x})/\bar{y} \) \quad \text{vc}(e) \leftarrow T(\tau) \\
and check for races
Thread $\tau_m$:

\[\langle 1, 0, 0 \rangle\]

\[
\ldots
\]

\[\langle 2, 0, 0 \rangle\]

$\tau_2 : \text{fork ()}$

\[\langle 3, 0, 0 \rangle\]

$\tau_3 : \text{fork ()}$

```javascript
1: var o = dictionary();  // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
4:   fork {
5:     o.put(host, createConnection(host));
6:   }
7: }
8: joinall;
9: print(o.size() + " connections established");
```
My Killer App

Concurrent HashMap
Concurrent Queue
Concurrent Queue

Observe traces

Thread $\tau_m$:

$\langle 1, 0, 0 \rangle$

$\langle 2, 0, 0 \rangle$

$\tau_2 : \text{fork}()$

$\langle 3, 0, 0 \rangle$

$\tau_3 : \text{fork}()$

Thread $\tau_3$:

$\langle 3, 0, 1 \rangle$

$a_1$

$o.\text{put}('a.com', c_1)/\text{nil}$

$o:w:'a.com'$

1: var o = dictionary(); // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
4:   fork {
5:     o.\text{put}(host, createConnection(host));
6:   }
7: }
8: joinall;
9: print(o.size() + " connections established");
Thread $\tau_3$: 

\[ \langle 3, 0, 1 \rangle \]

\[ o.\text{put}('a.com', c_1)/\text{nil} \]

\[ o:w:'a.com' \]
Concurrent HashMap
Concurrent Queue
Concurrent Agent
Agent 1
Agent 2
Agent 3
Agent 4
My Killer App

\[
\langle \chi_0, \eta_0, C_0 \rangle
\]

Temporal relationship

Observe traces

Commutativity relationship

Thread \( \tau_2 \):

\[
o \cdot \text{put ('a.com', } c_2) / c_1
\]

Thread \( \tau_3 \):

\[
o \cdot \text{put ('a.com', } c_1) / \text{nil}
\]

Incomparable vector clocks and Touch same access point

Race
Concurrent HashMap
Concurrent Queue
Concurrent Agent

My Killer App

1: var o = dictionary(); // Empty dictionary.
2: var hosts = ReadListOfHosts();
3: for host in hosts {
4:     fork {
5:         o.put(host, createConnection(host));
6:     }
7: }
8: joinall;
9: print(o.size() + " connections established");

Thread $\tau_2$:
\[ \langle 2, 1, 0 \rangle \]
\[ o.put('a.com', c_2)/c_1 \]
\[ o.w:'a.com' \]
\[ a_2 \]

Thread $\tau_3$:
\[ \langle 3, 0, 1 \rangle \]
\[ o.put('a.com', c_1)/\text{nil} \]
\[ o.w:'a.com' \]
\[ a_1 \]

Thread $\tau_0$:
\[ \langle 4, 1, 1 \rangle \]
\[ o.size() / 1 \]
\[ a_3 \]

Note: If $\tau_m$ omits joinall, then no conflict on $o.size$
Race detector orthogonal to Access Point representation
Asymptotic time complexity depends on Access Point representation
Next

1. Logical commutativity specifications
2. Access Point specifications
3. Translation from Logical Spec. to Access Points
4. Evaluation
Predicate $\varphi$ over pairs of actions $a, b \in Act$ such that $\varphi(a, b)$ implies that $a$ and $b$ commute.

**Dictionary Example**

\[
\begin{align*}
\varphi_{\text{put}(k_1,v_1)/p_1} &:= k_1 \neq k_2 \lor (v_1 = p_1 \land v_2 = p_2) \\
\varphi_{\text{put}(k_2,v_2)/p_2} &:= k_1 \neq k_2 \lor (v_1 = p_1) \\
\varphi_{\text{put}(k_1,v_1)/p_1} &:= (v_1 = \text{nil} \land p_1 = \text{nil}) \lor (v_1 \neq \text{nil} \land p_1 \neq \text{nil}) \\
\varphi_{\text{size}()}/r &:= \text{true}
\end{align*}
\]
Access Point Representation

\[ \langle \chi_o, \eta_o, C_o \rangle \]

1. \( \chi_o \) is a set of access points.
2. \( \eta_o \in \text{Act}_o \rightarrow \mathcal{P}(\chi_o) \) indicates the finite set of access points touched by each action of the object (\( \text{Act}_o \) stands for the set of all actions of object \( o \)).
3. \( C_o \subseteq \chi_o \times \chi_o \) is a symmetric binary relation describing which access points conflict.
Access Point Representation

Dictionary Example

\[ \mathcal{X}_o = \{o:r:k\}_{k \in K} \cup \{o:w:k\}_{k \in K} \cup \{o:\text{size}, o:\text{resize}\} \]

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Access Points ((\eta_o))</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(o.\text{put}(k, v)/p)</td>
<td>(o:w:k, o:\text{resize})</td>
<td>(v \neq p) and size changed</td>
</tr>
<tr>
<td></td>
<td>(o:w:k)</td>
<td>(v \neq p) and size unchanged</td>
</tr>
<tr>
<td></td>
<td>(o:r:k)</td>
<td>(v = p)</td>
</tr>
<tr>
<td>(o.\text{get}(k)/v)</td>
<td>(o:r:k)</td>
<td></td>
</tr>
<tr>
<td>(o.\text{size}())</td>
<td>(o:\text{size})</td>
<td></td>
</tr>
</tbody>
</table>
Access Point Representation

Accurate representation of a logical specification

For every action $a$ of $m_1$ and $b$ of $m_1$,

$$\left( \eta_o(a) \times \eta_o(b) \right) \cap C_o = \emptyset \iff \varphi_{m_2}^m(a, b)$$
# Race Detection

<table>
<thead>
<tr>
<th>Event</th>
<th>Modifications of auxiliary state</th>
</tr>
</thead>
</table>
| $e = \tau : fork(u)$ | $T(u) \leftarrow inc_u(T(\tau))$  
|             | $T(\tau) \leftarrow inc_\tau(T(\tau))$ |
| $e = \tau : join(u)$ | $T(\tau) \leftarrow T(\tau) \sqcup T(u)$ |
| $e = \tau : acq(l)$ | $T(\tau) \leftarrow T(\tau) \sqcup L(l)$ |
| $e = \tau : rel(l)$ | $L(l) \leftarrow T(\tau)$  
|             | $T(\tau) \leftarrow inc_\tau(T(\tau))$ |

$e = \tau : o.m(\vec{x})/\vec{y}$  
$vc(e) \leftarrow T(\tau)$  
Execute Algorithm 1

Novel
Race Detection

Input: event $e = \tau: o.m(\bar{x})/\bar{y}$, vector clock $vc(e)$, access point representation $\langle \chi_o, \eta_o, C_o \rangle$

Output: report a commutativity race or update auxiliary state

begin

// phase 1: check for commutativity races
for $pt \in \eta_o(o.m(\bar{x})/\bar{y})$ do
  for $pt' \in \text{active}(o) \cap C_o(pt)$ do
    if $pt'.vc \not\subseteq vc(e)$ then
      report "commutativity race";

// phase 2: update auxiliary state
for $pt \in \eta_o(o.m(\bar{x})/\bar{y})$ do
  if $pt \in \text{active}(o)$ then
    $pt.vc \leftarrow pt.vc \cup vc(e)$
  else
    // initialize
    $pt.vc \leftarrow vc(e)$;
    $\text{active}(o) \leftarrow \text{active}(o) \cup \{pt\}$;

\( ptvc: \chi \rightarrow VC \)
\( \text{active}: Obj \rightarrow \mathcal{P}(\chi) \)
Race Detection

Input: event $e = \tau : o.m(\bar{x})/\bar{y}$, vector clock $vc(e)$, access point representation $\langle \chi_o, \eta_o, \mathcal{C}_o \rangle$
Output: report a commutativity race or update auxiliary state

begin

// phase 1: check for commutativity races
for $pt \in \eta_o(o.m(\bar{x})/\bar{y})$ do
  for $pt' \in active(o) \cap \mathcal{C}_o(pt)$ do
    if $pt'.vc \nsubseteq vc(e)$ then
      report “commutativity race”;;

// phase 2: update auxiliary state
for $pt \in \eta_o(o.m(\bar{x})/\bar{y})$ do
  if $pt \in active(o)$ then
    $pt.vc \leftarrow pt.vc \cup vc(e)$
  else
    

<table>
<thead>
<tr>
<th>event action</th>
<th>identifier</th>
<th>vector clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_3 : o.put('a.com', c_1)/\text{nil}$</td>
<td>$a_1$</td>
<td>$\langle 3, 0, 1 \rangle$</td>
</tr>
<tr>
<td>$\tau_2 : o.put('a.com', c_2)/c_1$</td>
<td>$a_2$</td>
<td>$\langle 2, 1, 0 \rangle$</td>
</tr>
<tr>
<td>$\tau_m : \text{join } {\tau_2, \tau_3}$</td>
<td>$a_3$</td>
<td>$\langle 4, 1, 1 \rangle$</td>
</tr>
</tbody>
</table>

$ptvc : \chi \rightarrow VC$
$active : Obj \rightarrow \mathcal{P}(\chi)$

Look for active access point $pt'$ that conflicts with $pt$ and has incomparable vector clock

The vector clock for every access point $pt$ accumulates the vector clocks of all actions that have touched $pt$ so far
Race Detection

- Complexity dominated by Line 4. Effectively, $C_o(pt)$
- Soundness: reports a race iff the trace contains a race
- If a trace is free of races, then each trace which admits the same happens-before relation as the observed trace is free of traces and computes the same end result.
<table>
<thead>
<tr>
<th>ECL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$ ::= $V_1 \neq V_2$</td>
</tr>
<tr>
<td>$B$ ::= $P_{V_1}$</td>
</tr>
<tr>
<td>$X$ ::= $S$</td>
</tr>
</tbody>
</table>
**ECL**

**Hash Table**

**Commutativity Specification**

\[
\begin{align*}
\varphi_{\text{put}(k_1, v_1)/p_1} &= k_1 \neq k_2 \lor (v_1 = p_1 \land v_2 = p_2) \\
\varphi_{\text{get}(k_2)/v_2} &= k_1 \neq k_2 \lor (v_1 = p_1) \\
\varphi_{\text{put}(k_1, v_1)/p_1} &= (v_1 = \text{nil} \iff p_1 = \text{nil})
\end{align*}
\]

**Access Point Specification**

- \(X_o = \{o:w:k \mid k \in K\} \cup \{o:x:k \mid k \in K\} \cup \{o:\text{size}, o:\text{resize}\}\)
- \(\eta_o(a) = \begin{cases} 
\{o:\text{size}\} & \text{if } a = \text{size}(k) \\
\{o:x:k\} & \text{if } a = \text{get}(k)/x \\
\text{changeap}(a) \cup \text{resizeap}(a) & \text{if } a = \text{put}(k, v)/p
\end{cases}\)

where \(\text{changeap}(\text{put}(k, v)/p) = \text{if } v \neq p \text{ then } \{o:w:k\} \text{ else } \{o:x:k\}\)

and \(\text{resizeap}(\text{put}(k, v)/p) = \text{if } (v = \text{nil} \land p = \text{nil}) \lor (v \neq \text{nil} \land p \neq \text{nil}) \text{ then } \emptyset \text{ else } \{o:resize\}\)

- \(c_o = (\{(o:w:k, o:w:k) \mid k \in K\} \cup \{(o:w:k, o:x:k) \mid k \in K\} \cup \{(o:x:k, o:w:k) \mid k \in K\} \cup \{(o:resize, o:size)\} \cup \{(o:size, o:resize)\})\)
ECL

Hash Table

Commutativity Specification

\[
\begin{align*}
\phi_{\text{put}(k_1, v_1)/p_1} &:= k_1 \neq k_2 \lor (v_1 = p_1 \land v_2 = p_2) \\
\phi_{\text{put}(k_2, v_2)/p_2} &:= k_1 \neq k_2 \\
\phi_{\text{get}(k_1, v_1)/p_1} &:= k_1 \neq k_2 \\
\phi_{\text{get}(k_2)/v_2} &:= k_1 \neq k_2 \\
\phi_{\text{get}(k_1)} &:= \text{true} \\
\phi_{\text{get}(k_2)} &:= \text{true} \\
\phi_{\text{size}()} &:= \text{true} \\
\phi_{\text{size}()} &:= \text{true}
\end{align*}
\]

Translation

\[
\begin{align*}
o \cdot m: \beta : ds \\
o \cdot m: \beta : i : w_i
\end{align*}
\]

atomic prop. to truth value

\[
\langle x_o, \eta_o, c_o \rangle
\]
Results
Results

java.util.ConcurrentHashMap
Results

java.util.ConcurrentHashMap

open source JDBC SQL database server

Apache, open source, distributed database

PolePosition benchmarks
(www.polepos.org)
Results

java.util.ConcurrentHashMap

Multi-version store: (snapshot isolation)

Performance Ranks of database nodes:
DynamicEndpointSnitch

cassandra

H2
## Results

**Overhead**

```java
java.util.ConcurrentHashMap
```

<table>
<thead>
<tr>
<th>Application</th>
<th>Benchmark</th>
<th>Performance: qps or seconds</th>
<th></th>
<th>Races: total (distinct)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uninstrumented</td>
<td>FASTTRACK</td>
<td>RD2</td>
</tr>
<tr>
<td>H2 database</td>
<td>ComplexConcurrency</td>
<td>2011 qps</td>
<td>685 qps</td>
<td>425 qps</td>
</tr>
<tr>
<td></td>
<td>ComplexConcurrency (alternate query distrib.)</td>
<td>1610 qps</td>
<td>601 qps</td>
<td>457 qps</td>
</tr>
<tr>
<td></td>
<td>QueryCentricConcurrency</td>
<td>1666 qps</td>
<td>599 qps</td>
<td>605 qps</td>
</tr>
<tr>
<td></td>
<td>InsertCentricConcurrency</td>
<td>1912 qps</td>
<td>622 qps</td>
<td>622 qps</td>
</tr>
<tr>
<td></td>
<td>Complex</td>
<td>1874 qps</td>
<td>1143 qps</td>
<td>989 qps</td>
</tr>
<tr>
<td></td>
<td>NestedLists</td>
<td>1893 qps</td>
<td>1086 qps</td>
<td>807 qps</td>
</tr>
<tr>
<td>Cassandra</td>
<td>DynamicEndpointSnitch test</td>
<td>2.907 s</td>
<td>12.226 s</td>
<td>13.527 s</td>
</tr>
</tbody>
</table>
Results

Bugs

java.util.ConcurrentHashMap

1. Concurrent access to the chunks map in H2’s MVStore could lead to recomputing things.
2. Concurrent access to freedPageSpace.

1. New entries to the samples map concurrent while size being used as performance hint.
Contributions

• Dynamic commutativity race detector
• Logical fragment called ECL
• Translation procedure
• Implementation / Evaluation on ConcurrentHashMap
Related Work

Race Detection

- Flanagan and Freund. FastTrack. PLDI’09.
- Flanagan et al. Velodrome. PLDI’08.

Commutativity

- Kulkarni et al. Exploiting the commutativity lattice. PLDI’11.
- Kim and Renard. Verification of semantic commutativity conditions and inverse operations on linked data structures. PLDI’11.
Thank you!