Grand Challenge: Real-time Soccer Analytics
Leveraging Low-Latency Complex Event Processing

The 7th ACM International Conference on Distributed Event-Based Systems (DEBS)
June 29 - July 3, 2013, Arlington, Texas, USA

Martin Jergler

Joint work with: Christoph Doblander, Mohammedreza Najafi, Hans-Arno Jacobsen

Technical University of Munich
Department of Computer Science
Chair for Application and Middleware Systems (I13)
Middleware Systems Research Group (MSRG)
Outline

• Approach
  – Design & Architecture
  – Java & Low Latency
  – Example - Query 1
  – Aggregation

• Results

• Outlook & Conclusion
Our Approach

• Java-based & custom made
  – Performance?
  – Latency?

• Workflow-like modeling of queries
  – Producer/consumer principle
  – Query is split into tasks
  – Tasks can be executed in parallel
  – Tasks are connected by buffers
High-level System Architecture

Modular architecture

• Event Replay
  – Timed by thread yielding

• Parallel Query Processing
  – Query workflows
  – Efficient message passing

• Monitoring
  – Gathers performance statistics

• Client
  – Visualizes results
Java & Low Latency?

Problem
• Message passing between threads

More concretely
• Java garbage collection
• Locking
• Context switches
• Cache invalidations
Java & Low Latency?

Problem
- Message passing between threads

More concretely
- Java garbage collection
- Locking
- Context switches
- Cache invalidations

Solutions
- Avoid allocations during runtime
  - Re-use objects
- Avoid locking
  - Clever multithreading
  - Single thread (core) access to mutable memory
  - “Mechanical Sympathy”
Java & Low Latency?

Problem
- Message passing between threads

More concretely
- Java garbage collection
- Locking
- Context switches
- Cache invalidations

Solutions
- Avoid allocations during runtime
  - Re-use objects
- Avoid locking
  - Clever multithreading
  - Single thread (core) access to mutable memory
  - “Mechanical Sympathy”

Disruptor Pattern & Ringbuffer
Disruptor Library

• Concurrency module for Java
  – High-performance, low-latency message exchange between threads
  – **Ringbuffer** (wraps around, overwrite, pointers)
  – Event handlers (tasks)
  – Multi-producer/multi-consumer

• Tasks
  – Single threaded -> sequential

• Open-sourced by LMAX Trading (Apache 2.0 license)
Query Processing Architecture

- **Distributor** – distributes sensor data to queries
- **Merger** – collects queries results
Query Processing Architecture

- **Distributor** – distributes sensor data to queries
- **Merger** – collects queries results
Query 1 – Running Statistics (1)

• Query 1 is comprised of three different tasks (21 threads)
  1. Distribution of sensor events for individual players  [1 thread]
  2. Calculation of statistics per player (distance, intensity) [16 threads]
  3. Aggregation over different time windows  [4 threads]
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

```java
Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …
```

Statistics calculation (per player)

```java
List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

```java
Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());

oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();
```

// Aggregation for 5 minute window ...
// Aggregation for 20 minute window ...
// Aggregation for whole match ...

Statistics calculation (per player)

```java
List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingBuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, ...);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, ...));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingBuffer.add(Pair.of(player, ringBuffer));
}
```

// Input Disruptor (i.e., Packetizer) ...
Query 1 – Running Statistics (2)

Aggregation over time windows (here: 1 minute)

```java
Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());

oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();
```

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …

Statistics calculation (per player)

```java
List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}
```

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …

Statistics calculation (per player)

List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …

Statistics calculation (per player)

List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …

Statistics calculation (per player)

List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

```java
Disruptor<PlayerStatistics> oneMinuteAggregate =
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();
```

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …

Statistics calculation (per player)

```java
List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}
```

// Input Disruptor (i.e., Packetizer) …
Query 1 – Running Statistics (2)

Aggregation over timewindows (here: 1 minute)

```java
Disruptor<PlayerStatistics> oneMinuteAggregate = 
    new Disruptor<>(PlayerStatistics, ProducerType.MULTI, new BlockWaitingStrategy());
oneMinuteAggregate.handleEventsWith(new PlayerAggregation(1));
RingBuffer<PlayerStatistics> oneMinuteRB = oneMinuteAggregate.start();

// Aggregation for 5 minute window …
// Aggregation for 20 minute window …
// Aggregation for whole match …
```

Statistics calculation (per player)

```java
List<Pair<HasIdForSubscription, RingBuffer<SensorEvent>>> playerPerRingbuffer = new ArrayList<>();
for(Player player : players) {
    Disruptor<SensorEvent> runningAnalysisPerPlayer = new Disruptor<>(SensorEvent, ProducerType.SINGLE, …);
    runningAnalysisPerPlayer.handleEventsWith(new RunningAnalysisPerPlayer(player, oneMinuteRB, fiveMinuteRB, …));
    RingBuffer<SensorEvent> ringBuffer = runningAnalysisPerPlayer.start();
    playerPerRingbuffer.add(Pair.of(player, ringBuffer));
}

// Input Disruptor (i.e., Packetizer) …
```
Aggregation Strategy

- **Fixed-size queue**
  - Stores pre-aggregations
  - Wraps around
  - Pointers to end of interval
  - Operations: `push()/poll()` in $O(1)$

- **Aggregator**
  - For each time window
  - For each grid configuration
    1. Add new pre-aggregation (push)
    2. Subtract old pre-aggregation (poll)

![Diagram of Aggregation Strategy](image-url)

- **1 minute pre-aggregate**
  (occ. of player in grid)

- **Ringbuffer**

- **Aggregations of Heatmap statistics for the different time windows**
  - 1 minute pre-aggregate completed
  - Whole game pre-aggregate completed
Experimental Evaluation – Setup

• 4 machines
  – 1 x Intel i7-3520M Dual Core (@ 2.90 GHz) (1x2)
  – 1 x Intel i7-3720QM Quad Core (@ 2.60 GHz) (1x4)
  – 2 x Intel Xeon E5-2665 CPU Octo Core (@ 2.40 GHz) (2x8)
  – 4 x Intel Xeon E7-4807 CPU Six Core (@ 1.86 GHz) (4x6)

• Latency measurements at percentile
  – For Query 1 & Query 3
  – Histogram component
  – 50.000 – 99.805

• Throughput with speedup of event stream
  – For Query 1
  – 1 – 48
Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 - XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

**Query 1**

- **96.875 %**: 1x2 faster than 1x4
  - Better cache affinity
  - Higher clock rate
- **99.805 %**: 1x4 still predictable
  - Machine was never overloaded
- **2x8**: Always predictable
  - Thread affinity increases performance
- **4x6**: Performs best
  - More cores than threads
  - However, not always predictable

---

Query 1 - Latency at percentile in μs.
Thread **affinity** set to second CPU; **Spin** waiting strategy
## Results – Latency In Microseconds

### Query 1

- **96.875%**: 1x2 faster than 1x4
  - Better cache affinity
  - Higher clock rate

- **99.805%**: 1x4 still predictable
  - Machine was never overloaded

- **2x8**: Always predictable
  - Thread affinity increases performance

- **4x6**: Performs best
  - More cores than threads
  - However, not always predictable

---

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

**Query 1 - Latency at percentile in μs.**

Thread **affinity** set to second CPU; **Spin** waiting strategy
Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

Query 1 - Latency at percentile in µs.
Thread affinity set to second CPU; Spin waiting strategy

- **96.875 %: 1x2 faster than 1x4**
  - Better cache affinity
  - Higher clock rate

- **99.805 %: 1x4 still predictable**
  - Machine was never overloaded

- **2x8: Always predictable**
  - Thread affinity increases performance

- **4x6: Performs best**
  - More cores than threads
  - However, not always predictable
Results – Latency In Microseconds

Query 1

- 96.875 %: 1x2 faster than 1x4
  - Better cache affinity
  - Higher clock rate
- 99.805 %: 1x4 still predictable
  - Machine was never overloaded
- 2x8: Always predictable
  - Thread affinity increases performance
- 4x6: Performs best
  - More cores than threads
  - However, not always predictable

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

Query 1 - Latency at percentile in μs.
Thread affinity set to second CPU; Spin waiting strategy
Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

**Query 1**

- **96.875 %: 1x2 faster than 1x4**
  - Better cache affinity
  - Higher clock rate
- **99.805 %: 1x4 still predictable**
  - Machine was never overloaded
- **2x8: Always predictable**
  - Thread affinity increases performance
- **4x6: Performs best**
  - More cores than threads
  - However, not always predictable

Query 1 - Latency at percentile in μs.
Thread affinity set to second CPU; Spin waiting strategy
## Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

**Query 1**

- **96.875 %: 1x2 faster than 1x4**
  - Better cache affinity
  - Higher clock rate
- **99.805 %: 1x4 still predictable**
  - Machine was never overloaded
- **2x8: Always predictable**
  - Thread affinity increases performance
- **4x6: Performs best**
  - More cores than threads
  - However, not always predictable

---

**Query 1 - Latency at percentile in μs.**

Thread **affinity** set to second CPU; **Spin** waiting strategy
Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

**Query 1**

- **96.875 %: 1x2 faster than 1x4**
  - Better cache affinity
  - Higher clock rate
- **99.805 %: 1x4 still predictable**
  - Machine was never overloaded
- **2x8: Always predictable**
  - Thread affinity increases performance
- **4x6: Performs best**
  - More cores than threads
  - However, not always predictable

**Query 3**

- **Behavior similar to Q1**
- **Exception 4x6: Performs worst**
  - More threads than cores
  - Lower clock rate
  - Many context switches
Results – Latency In Microseconds

<table>
<thead>
<tr>
<th>Result Percentile</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x2 - i7</td>
<td>9.2</td>
<td>18.4</td>
<td>38.9</td>
<td>1,471.4</td>
<td>9,206.7</td>
</tr>
<tr>
<td>1x4 - i7</td>
<td>49.7</td>
<td>59.2</td>
<td>61.2</td>
<td>69.5</td>
<td>76.9</td>
</tr>
<tr>
<td>2x8 - XEON</td>
<td>30.1</td>
<td>52.2</td>
<td>80.7</td>
<td>108.9</td>
<td>145.1</td>
</tr>
<tr>
<td>2x8 - XEON - aff</td>
<td>12.7</td>
<td>39.9</td>
<td>48.9</td>
<td>71.7</td>
<td>83.1</td>
</tr>
<tr>
<td>4x6 – XEON - spin</td>
<td>3.9</td>
<td>4.5</td>
<td>5.3</td>
<td>12.0</td>
<td>1,883.0</td>
</tr>
</tbody>
</table>

Query 1 - Latency at percentile in μs.
Thread affinity set to second CPU; Spin waiting strategy

- 96.875%: 1x2 faster than 1x4
  - Better cache affinity
  - Higher clock rate
- 99.805%: 1x4 still predictable
  - Machine was never overloaded
- 2x8: Always predictable
  - Thread affinity increases performance
- 4x6: Performs best
  - More cores than threads
  - However, not always predictable

Query 3

- Behavior similar to Q1
  - Exception 4x6: Performs worst
    - More threads than cores
    - Lower clock rate
    - Many context switches
Results – Latency In Microseconds

Query 1
- 96.875 %: 1x2 faster than 1x4
  - Better cache affinity
  - Higher clock rate
- 99.805 %: 1x4 still predictable
  - Machine was never overloaded
- 2x8: Always predictable
  - Thread affinity increases performance
- 4x6: Performs best
  - More cores than threads
  - However, not always predictable

Query 3
- Behavior similar to Q1
- Exception 4x6: Performs worst
  - More threads than cores
  - Lower clock rate
  - Many context switches
Results – Speedup (Throughput)

<table>
<thead>
<tr>
<th>X</th>
<th>Rate</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13,315</td>
<td>27.3</td>
<td>60.2</td>
<td>411.2</td>
<td>3,598.4</td>
<td>9,150.7</td>
</tr>
<tr>
<td>2</td>
<td>25,399</td>
<td>29.0</td>
<td>67.6</td>
<td>647.9</td>
<td>8,321.2</td>
<td>16,471.8</td>
</tr>
<tr>
<td>4</td>
<td>52,130</td>
<td>26.9</td>
<td>58.0</td>
<td>907.4</td>
<td>8,023.6</td>
<td>10,284.9</td>
</tr>
<tr>
<td>8</td>
<td>105,271</td>
<td>27.9</td>
<td>70.4</td>
<td>1,205.7</td>
<td>9,553.1</td>
<td>15,107.1</td>
</tr>
<tr>
<td>16</td>
<td>208,869</td>
<td>24.1</td>
<td>150.7</td>
<td>4,792.5</td>
<td>11,652.0</td>
<td>14,103.9</td>
</tr>
<tr>
<td>32</td>
<td>372,625</td>
<td>26.5</td>
<td>84.5</td>
<td>713.5</td>
<td>8,353.0</td>
<td>10,781.4</td>
</tr>
<tr>
<td>48</td>
<td>510,290</td>
<td>21.9</td>
<td>215.9</td>
<td>11,932.4</td>
<td>13,974.3</td>
<td>14,288.9</td>
</tr>
</tbody>
</table>

Query 1 - Speedup by factor X on dual core. Latencies at percentile in μs.

Setting

- Query 1
- 1x2 @ 2.90 GHz
- Solid-state-drive

Result

- Latency does not deteriorate
- Massive queuing would increase latency
- Limiting factor: Disk speed
Results – Speedup (Throughput)

<table>
<thead>
<tr>
<th>X</th>
<th>Rate</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13,315</td>
<td>27.3</td>
<td>60.2</td>
<td>411.2</td>
<td>3,598.4</td>
<td>9,150.7</td>
</tr>
<tr>
<td>2</td>
<td>25,399</td>
<td>29.0</td>
<td>67.6</td>
<td>647.9</td>
<td>8,321.2</td>
<td>16,471.8</td>
</tr>
<tr>
<td>4</td>
<td>52,130</td>
<td>26.9</td>
<td>58.0</td>
<td>907.4</td>
<td>8,023.6</td>
<td>10,284.9</td>
</tr>
<tr>
<td>8</td>
<td>105,271</td>
<td>27.9</td>
<td>70.4</td>
<td>1,205.7</td>
<td>9,553.1</td>
<td>15,107.1</td>
</tr>
<tr>
<td>16</td>
<td>208,869</td>
<td>24.1</td>
<td>150.7</td>
<td>4,792.5</td>
<td>11,652.0</td>
<td>14,103.9</td>
</tr>
<tr>
<td>32</td>
<td>372,625</td>
<td>26.5</td>
<td>84.5</td>
<td>713.5</td>
<td>8,353.0</td>
<td>10,781.4</td>
</tr>
<tr>
<td>48</td>
<td>510,290</td>
<td>21.9</td>
<td>215.9</td>
<td>11,932.4</td>
<td>13,974.3</td>
<td>14,288.9</td>
</tr>
</tbody>
</table>

Query 1 - Speedup by factor X on dual core. Latencies at percentile in μs.

Setting
- Query 1
- 1x2 @ 2.90 GHz
- Solid-state-drive

Result
- Latency does not deteriorate
- Massive queuing would increase latency
- Limiting factor: Disk speed
### Results – Speedup (Throughput)

<table>
<thead>
<tr>
<th>X</th>
<th>Rate</th>
<th>50%</th>
<th>87.5%</th>
<th>96.875%</th>
<th>99.21875%</th>
<th>99.8046875%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13,315</td>
<td>27.3</td>
<td>60.2</td>
<td>411.2</td>
<td>3,598.4</td>
<td>9,150.7</td>
</tr>
<tr>
<td>2</td>
<td>25,399</td>
<td>29.0</td>
<td>67.6</td>
<td>647.9</td>
<td>8,321.2</td>
<td>16,471.8</td>
</tr>
<tr>
<td>4</td>
<td>52,130</td>
<td>26.9</td>
<td>58.0</td>
<td>907.4</td>
<td>8,023.6</td>
<td>10,284.9</td>
</tr>
<tr>
<td>8</td>
<td>105,271</td>
<td>27.9</td>
<td>70.4</td>
<td>1,205.7</td>
<td>9,553.1</td>
<td>15,107.1</td>
</tr>
<tr>
<td>16</td>
<td>208,869</td>
<td>24.1</td>
<td>150.7</td>
<td>4,792.5</td>
<td>11,652.0</td>
<td>14,103.9</td>
</tr>
<tr>
<td>32</td>
<td>372,625</td>
<td>26.5</td>
<td>84.5</td>
<td>713.5</td>
<td>8,353.0</td>
<td>10,781.4</td>
</tr>
<tr>
<td>48</td>
<td>510,290</td>
<td>21.9</td>
<td>215.9</td>
<td>11,932.4</td>
<td>13,974.3</td>
<td>14,288.9</td>
</tr>
</tbody>
</table>

**Query 1 - Speedup by factor X on dual core. Latencies at percentile in μs.**

**Setting**
- Query 1
- 1x2 @ 2.90 GHz
- Solid-state-drive

**Result**
- Latency does not deteriorate
- Massive queuing would increase latency
- Limiting factor: Disk speed

x 48 speedup for only 50% more latency
Conclusions & Outlook

• Our approach shows that
  – Low latency event processing in Java is possible
  – “Mechanical Sympathy” is realizable on the JVM and can significantly improve performance

• Possible improvements & further directions are
  – Reduce memory allocations
  – Reduce number of threads
  – Leverage more aggressive waiting strategies
  – Optimizer – determine optimal scheduling strategies
THANK YOU

Questions?