Java on Scalable Memory Architectures

University of Crete, 25th of October 2016

Foivos S. Zakkak

Outline

Moore's Law, Power Wall & Coherence Protocols

- The size of a transistor shrinks to half every ~18 months [Krzanich2015]
- Due to the power wall frequency scaling stopped
- The extra transistors are used for extra cores
- Cache coherency not scaling in terms of performance and energy consumption [Kaxiras et al. 2010, Choi et al. 2011]

Emerging Trends

- Hundreds of cores per U (rack unit)
- Lack of global memory coherence
- Faster communication (TCP-IP vs RDMA)

Architectures

EuroServer [Durand et al. 2014]

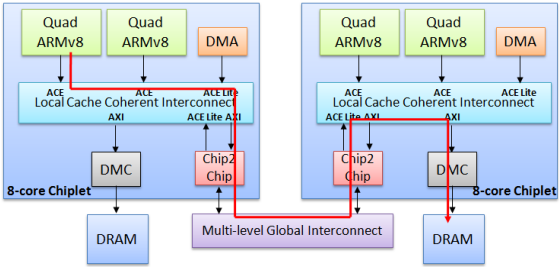


Figure Source: *Durand et al. 2014*

Runnemede [Carter et al. 2013]

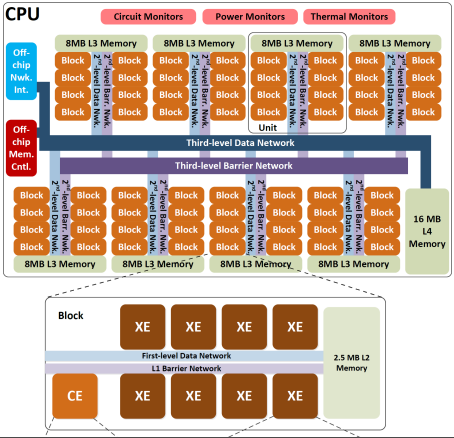


Figure Source: *Carter et al. 2013*

Managed Programming Languages

Highlights

- Consistent behavior across different platforms
- Abstract away hardware details
- Shorter time to market (TTM)
- Portability: *"write once run everywhere"*

Mechanisms

- Language Specification
- Memory Model
- Process Virtual Machine

Managed Programming Languages

Highlights

- Consistent behavior across different platforms
- Abstract away hardware details
- Shorter time to market (TTM)
- Portability: *"write once run everywhere"*

Mechanisms

- Language Specification
- Memory Model
- Process Virtual Machine

Contributions

- Java Distributed Memory Model (JDMM) [ISMM'14]
  - Extension of JMM that exposes memory management operations
  - Adheres to JMM
  - Allows same optimizations as JMM
- Process Virtual Machine Design and Implementation [JTRES'16]
  - Design of novel algorithms for software caching and synchronization
  - Implementation of a distributed JVM for non-cache-coherent architectures
  - Evaluation on Formic, a 512-core Emulator
- Distributed Java Calculus (DJC) [PPPJ'16]
  - Mathematical model of the implementation
  - Proof of adherence to JMM

Outline

What is a memory model?

- Formal specification that describes the behavior of a program
- Models all possible executions of a program (legal or not)
- Provides rules that determine which executions are legal

What is it good for?

- Contract between language or machine designers, implementers, and end-users
- Language requirements regarding interaction of threads through memory
- Helps language implementers build the runtime and/or the compiler
- Helps developers understand a program's behavior

The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

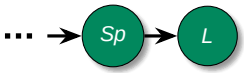
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

$\leq_{po}$: program order

The Java Memory Model

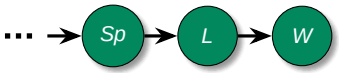
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



$\leq_{po}$: program order

The Java Memory Model

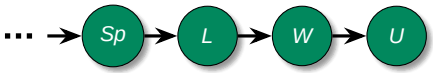
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



$\leq_{po}$: program order

The Java Memory Model

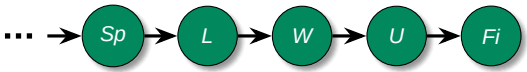
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  } ←
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



$\leq_{po}$: program order

The Java Memory Model

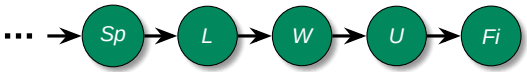
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



$\leq_{po}$: program order

The Java Memory Model

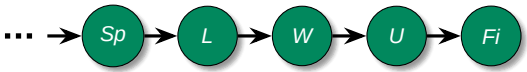
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

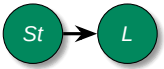


$\leq_{po}$: program order

The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

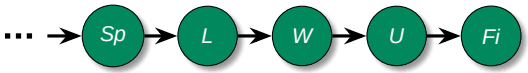


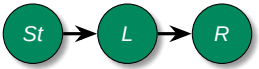


$\leq_{po}$: program order

The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

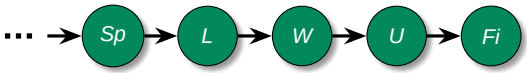


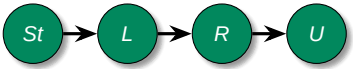


$\leq_{po}$: program order

The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

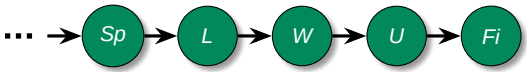




$\leq_{po}$: program order

The Java Memory Model

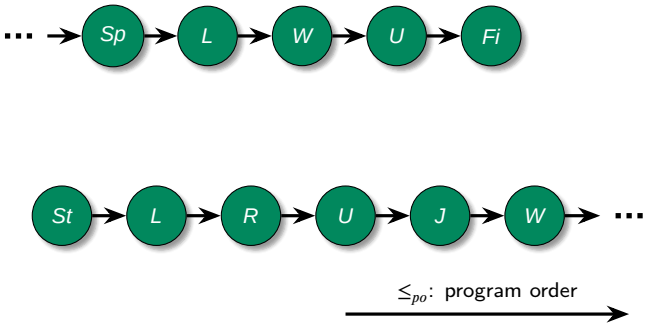
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



$\leq_{po}$: program order

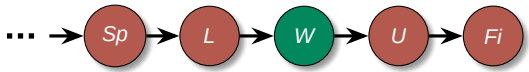
The Java Memory Model

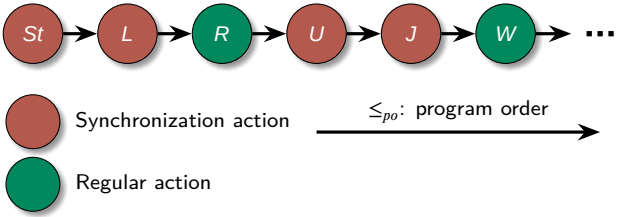
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



The Java Memory Model

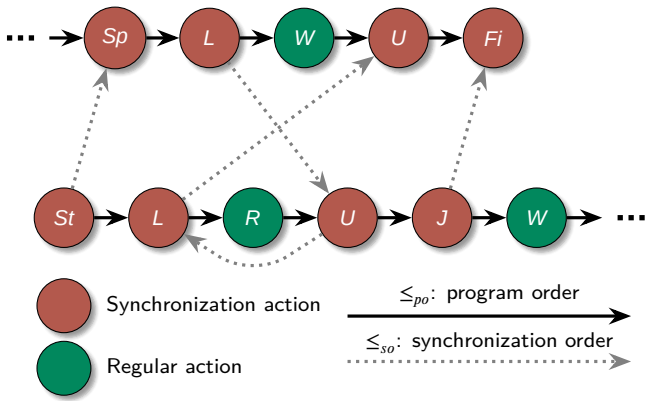
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```





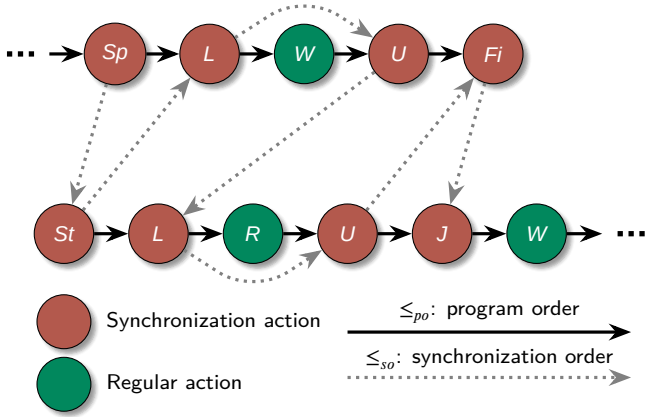
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



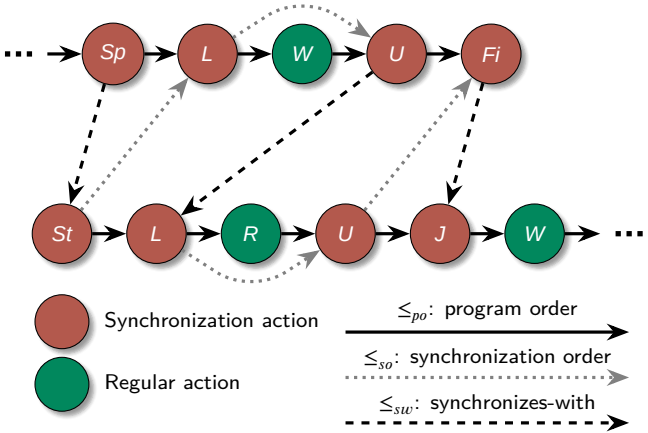
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



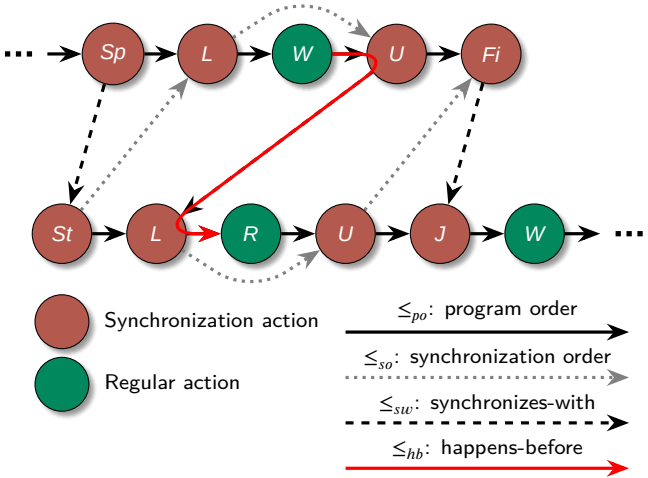
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



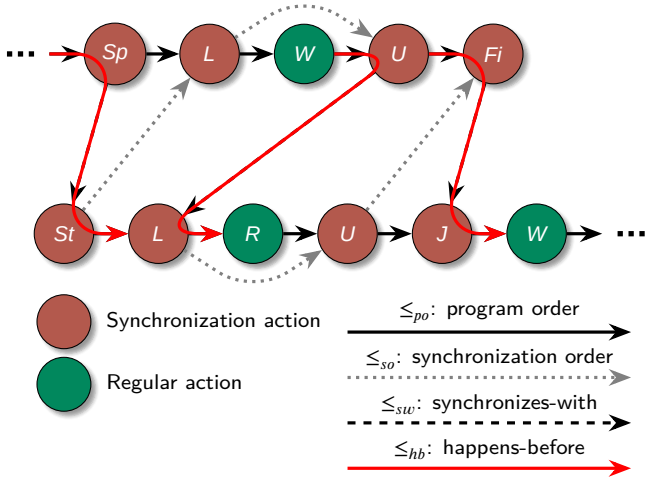
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



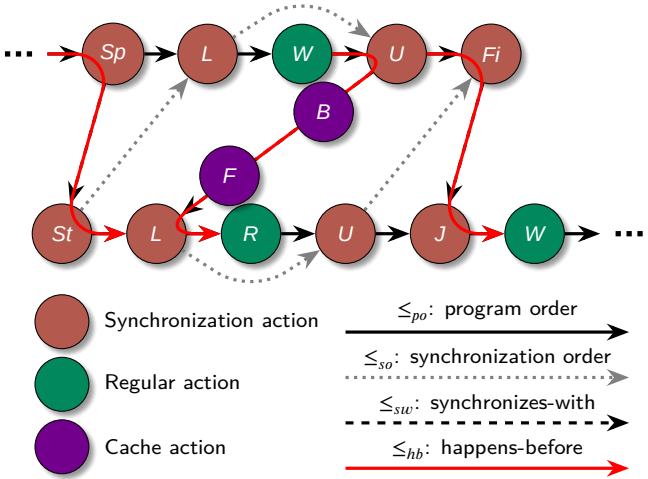
The Java Memory Model

```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```

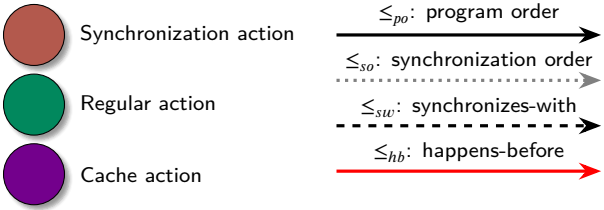


The Java Memory Model

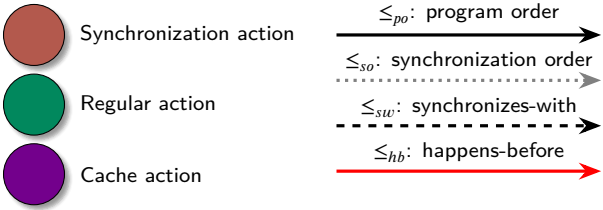
```
run() { // Thread 1
  // ...
  thread2.start();
  synchronized(object1) {
    object1.field1 = 42;
  }
}

run() { // Thread 2
  synchronized(object1) {
    temp1 = object1.field1;
  }
  thread1.join();
  object1.field1 = 24;
  // ...
}
```



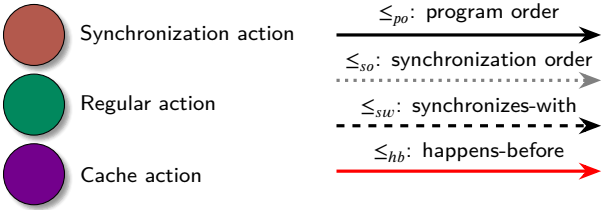
JDMM Introduced Well-Formedness Rules

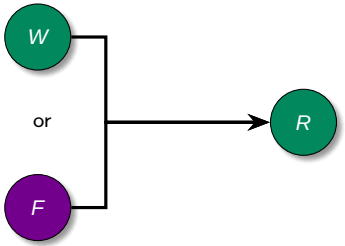


JDMM Introduced Well-Formedness Rules

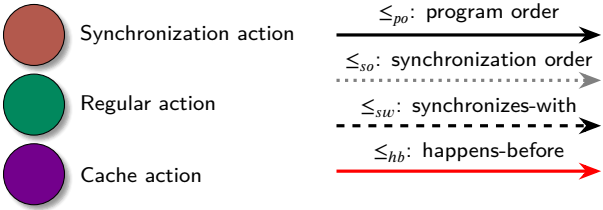


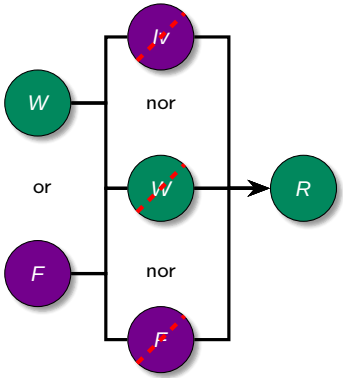
JDMM Introduced Well-Formedness Rules



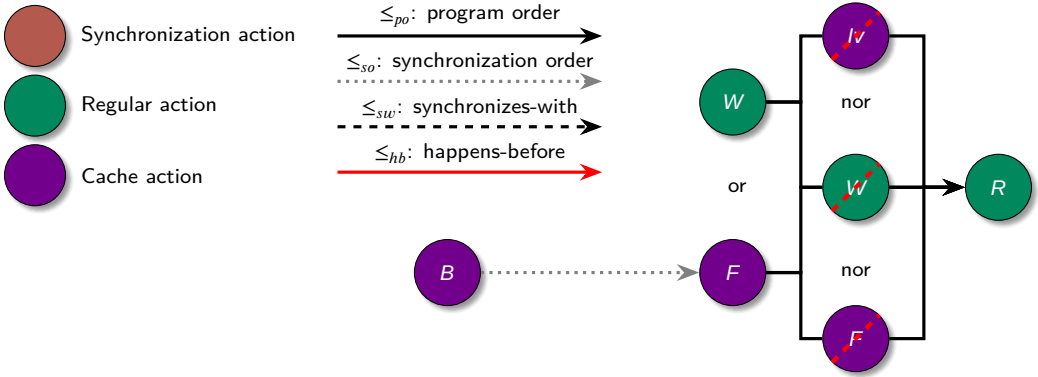


JDMM Introduced Well-Formedness Rules

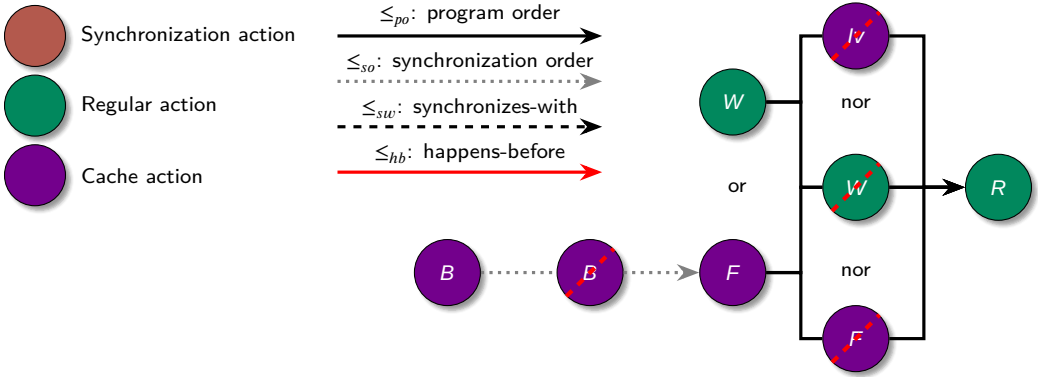




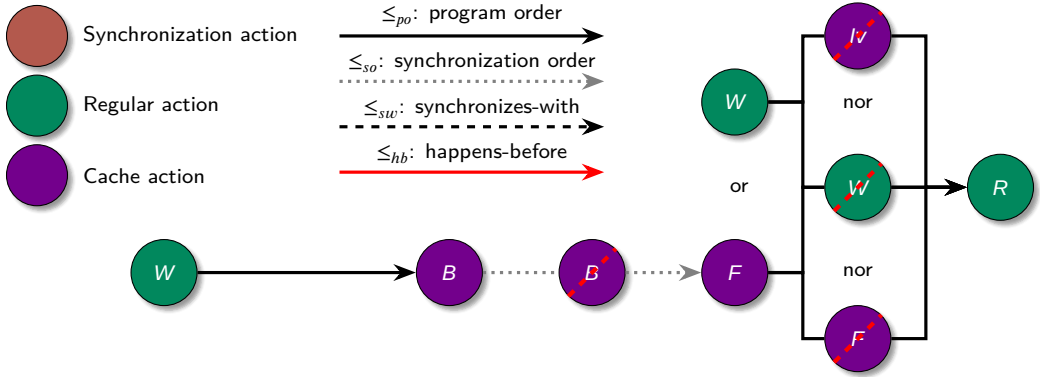
JDMM Introduced Well-Formedness Rules



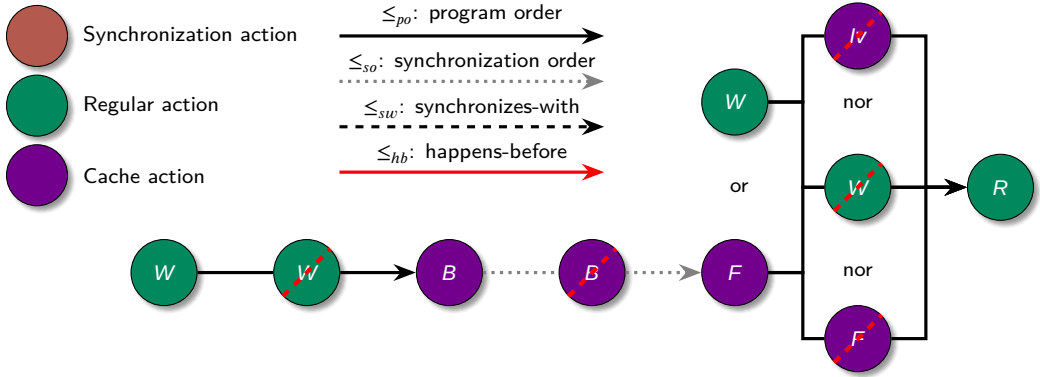
JDMM Introduced Well-Formedness Rules



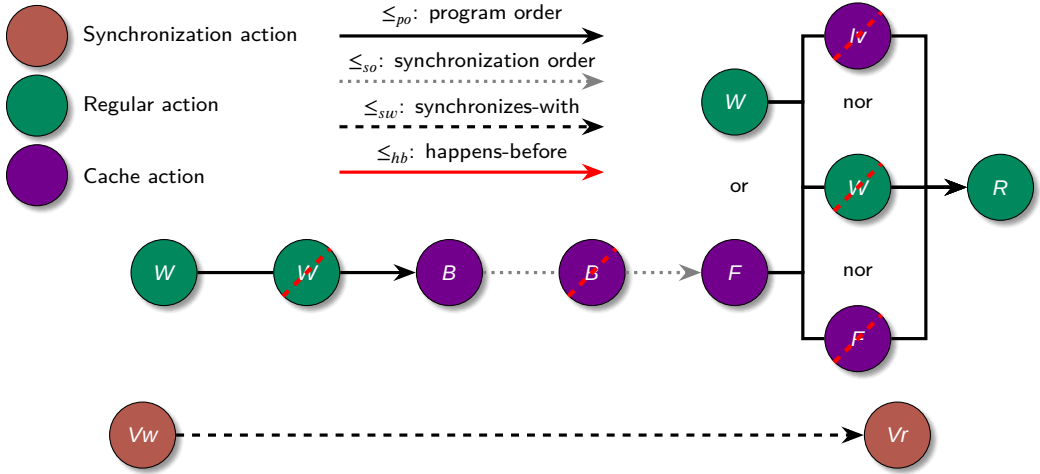
JDMM Introduced Well-Formedness Rules



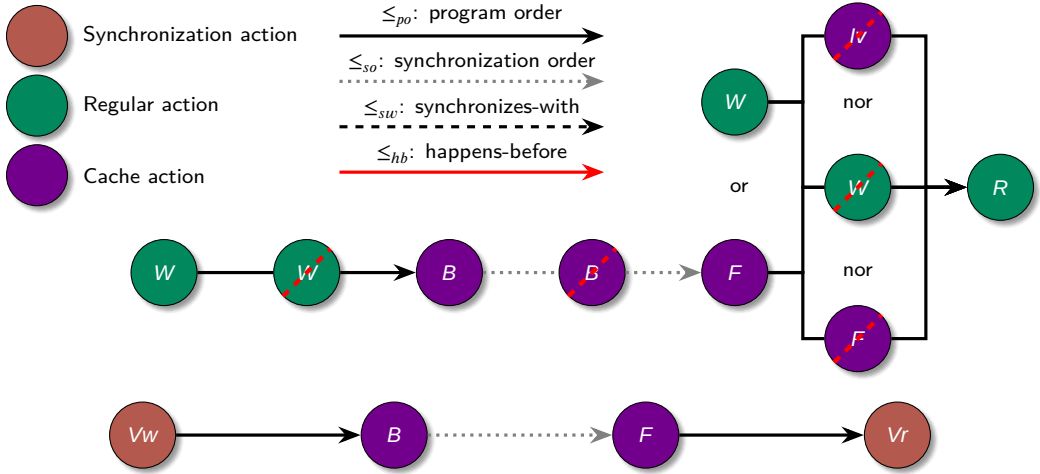
JDMM Introduced Well-Formedness Rules



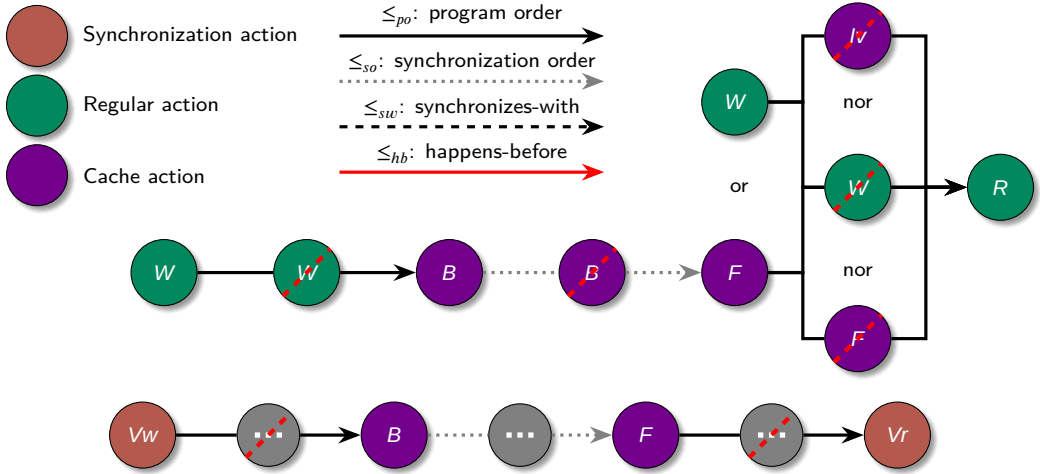
JDMM Introduced Well-Formedness Rules



JDMM Introduced Well-Formedness Rules



JDMM Introduced Well-Formedness Rules



Formalization

JMM Execution

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

JDMM Execution

$$E_D = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W(), V(), \leq^d_{sw}, \leq^d_{hb}, Cs(), Bf(), Ab(), Ai() \rangle$$

JMM & JDMM Rules

$$\forall r \in A_D : \exists x \in A_D : \left(x \leq^d_{po} r\right) \wedge (x.v = r.v) \wedge \left(x.k \in \{W, F\}\right)$$

Formalization

JMM Execution

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

JDMM Execution

$$E_D = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W(), V(), \leq^d_{sw}, \leq^d_{hb}, Cs(), Bf(), Ab(), Ai() \rangle$$

JMM & JDMM Rules

$$\forall r \in A_D : \exists x \in A_D : \left(x \leq^d_{po} r\right) \wedge (x.v = r.v) \wedge \left(x.k \in \{W, F\}\right)$$

Formalization

JMM Execution

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

JDMM Execution

$$E_D = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W(), V(), \leq^d_{sw}, \leq^d_{hb}, Cs(), Bf(), Ab(), Ai() \rangle$$

JMM & JDMM Rules

$$\forall r \in A_D : \exists x \in A_D : \left(x \leq^d_{po} r\right) \wedge (x.v = r.v) \wedge \left(x.k \in \{W, F\}\right)$$

Formalization

JMM Execution

$$E = \langle P, A, \leq_{po}, \leq_{so}, W(), V(), \leq_{sw}, \leq_{hb} \rangle$$

JDMM Execution

$$E_D = \langle P, A_D, \leq^d_{po}, \leq^d_{so}, W(), V(), \leq^d_{sw}, \leq^d_{hb}, Cs(), Bf(), Ab(), Ai() \rangle$$

JMM & JDMM Rules

$$\forall r \in A_D : \exists x \in A_D : \left(x \leq^d_{po} r\right) \wedge (x.v = r.v) \wedge \left(x.k \in \{W, F\}\right)$$

JDMM Properties

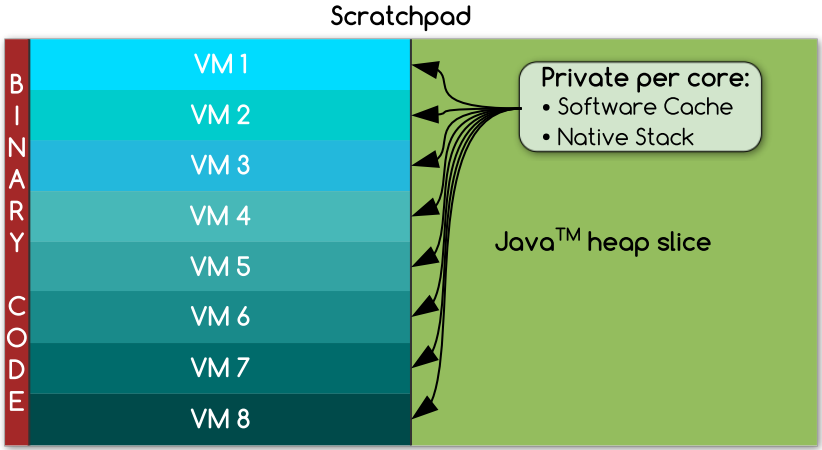
- Allows same optimizations as JMM
- Satisfies the same causality test cases as JMM
  - Small tests, testing the conformance of the implementation and specification to the expected behaviour
  - Behaviors extracted by JVM implementations at the time of JMM specification

Implementation Advice Based on JDMM

- Avoid sync on nested monitor acquisition
- Avoid local caching
- Avoid sync at context switching
- Sync at thread migration
- Direct cache-to-cache transfers

Outline

Memory Management

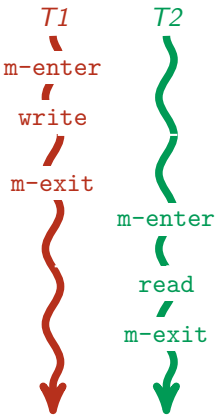


Software Caching

- Private write caches per thread
  - Write-back on capacity misses
  - Write-back at release operations
- Shared read caches per core
  - Invalidate on capacity misses
  - Invalidate on acquire operations
  - Update on write-back

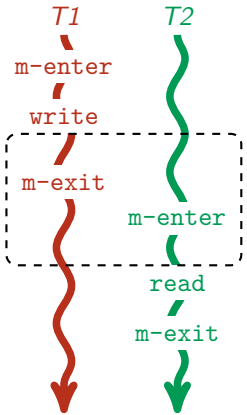
Software Caching

- Private write caches per thread
  - Write-back on capacity misses
  - Write-back at release operations
- Shared read caches per core
  - Invalidate on capacity misses
  - Invalidate on acquire operations
  - Update on write-back



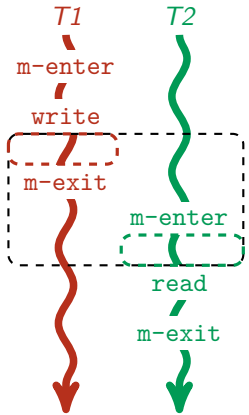
Software Caching

- Private write caches per thread
  - Write-back on capacity misses
  - Write-back at release operations
- Shared read caches per core
  - Invalidate on capacity misses
  - Invalidate on acquire operations
  - Update on write-back



Software Caching

- Private write caches per thread
  - Write-back on capacity misses
  - Write-back at release operations
- Shared read caches per core
  - Invalidate on capacity misses
  - Invalidate on acquire operations
  - Update on write-back



Synchronization

- Synchronization Managers (on dedicated cores)
- Queuing requests when monitor not available
- Co-located threads may reuse monitors to reduce network traffic (combining)

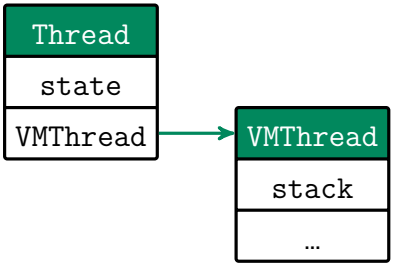
Scheduling

- New thread Start
  1. Pick a random core from the same island
  2. If its deque is full pick a random island
- Work-Stealing (asynchronous) within coherent-islands
  1. Attempt $\left\lceil \sqrt{\#\text{Cores per island}} \right\rceil$ steals
- Work-Dealing (synchronous) across coherent-islands
  1. Attempt $\left\lceil \sqrt{\#\text{Coherent islands}} \right\rceil$ deals
  2. Get half tasks

Thread.start()

1. Weak thread creation
2. Send thread start request to random core
3. Local thread initialization
4. Local stack allocation

- Running thread operates on VMThread class
- Remote threads synchronize with Thread class

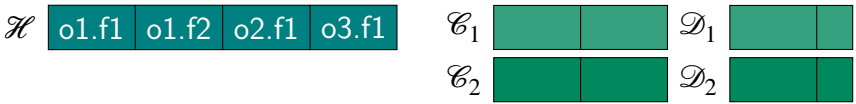


Outline

Approach

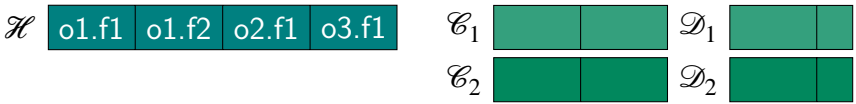
- Definition of Distributed Java Calculus (DJC), a Java core calculus
  - Explicit cache operations
  - Locking & Synchronization actions
- Operational semantics reflecting DiSquawk's design

Modeling DiSquawk




```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

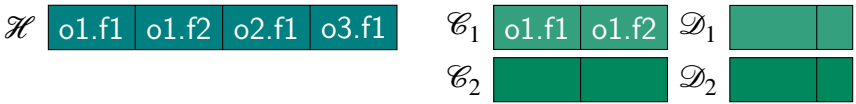
Modeling DiSquawk



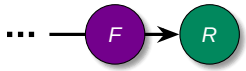

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

...

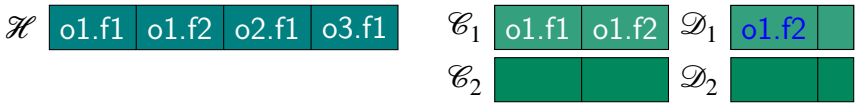
Modeling DiSquawk



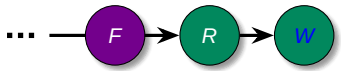

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



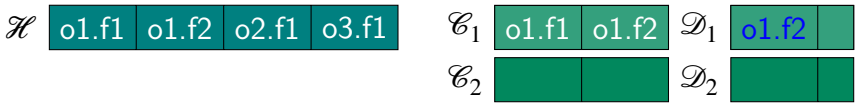
Modeling DiSquawk



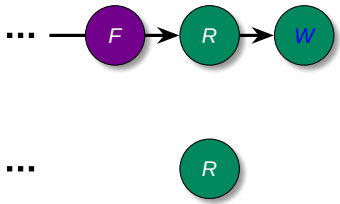

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



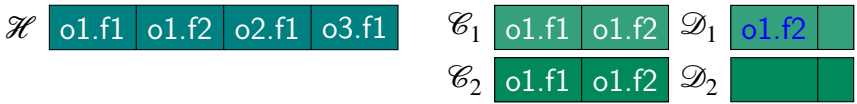
Modeling DiSquawk



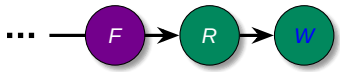

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

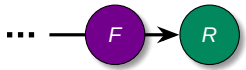


Modeling DiSquawk

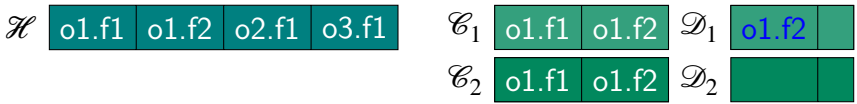



```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

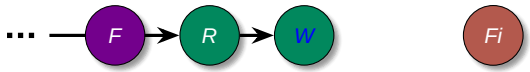


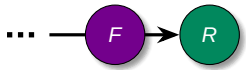


Modeling DiSquawk

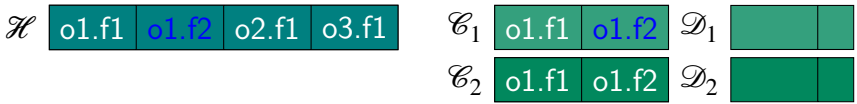



```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

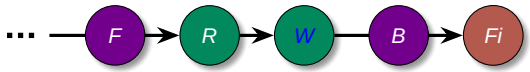


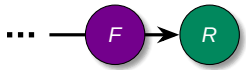


Modeling DiSquawk

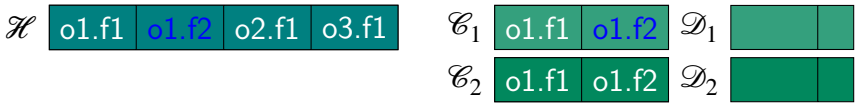



```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```

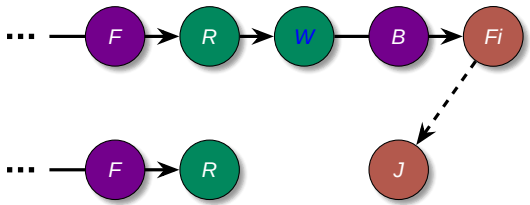




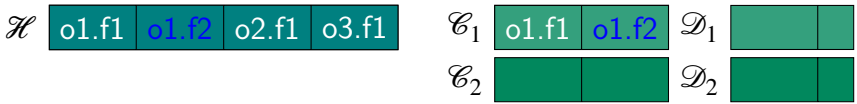
Modeling DiSquawk



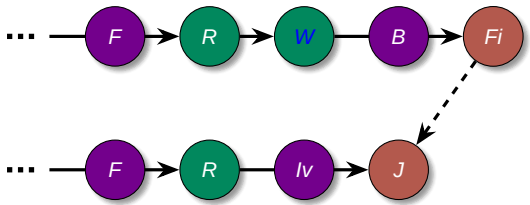

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



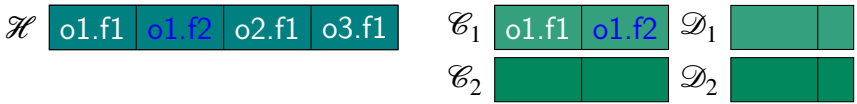
Modeling DiSquawk



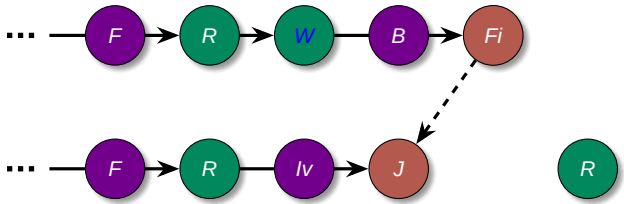

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



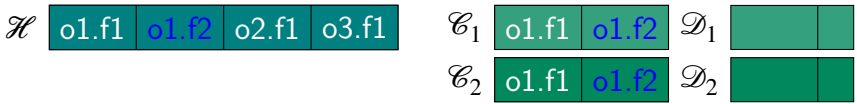
Modeling DiSquawk



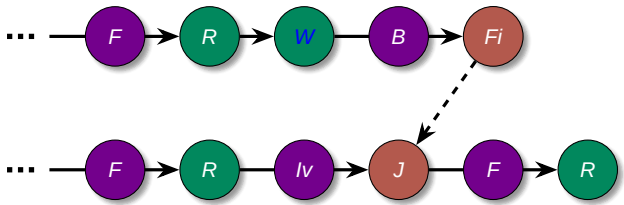

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



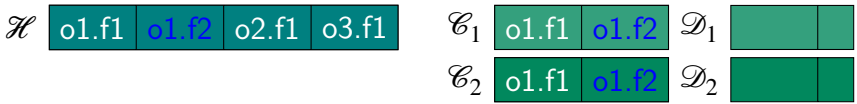
Modeling DiSquawk



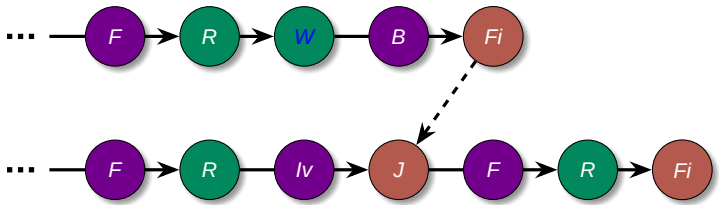

```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



Modeling DiSquawk




```
run() { // Thread 1
  // ...
  o1.f2++;
}

run() { // Thread 2
  int temp;
  // ...
  temp = o1.f1;
  thread1.join();
  temp = o1.f2;
}
```



Distributed Java Calculus

DJC Memory State

$$\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}}$$

DJC Execution Step

$$\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash T \xrightarrow[\vec{c}]{\vec{a}} \mathscr{H}';\vec{\mathscr{C}}';\vec{\mathscr{D}}' \vdash T'$$

Distributed Java Calculus

DJC Memory State

$$\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}}$$

DJC Execution Step

$$\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash T \xrightarrow[\vec{c}]{\vec{a}} \mathscr{H}';\vec{\mathscr{C}}';\vec{\mathscr{D}}' \vdash T'$$

Distributed Java Calculus

DJC Memory State

$$\mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}}$$

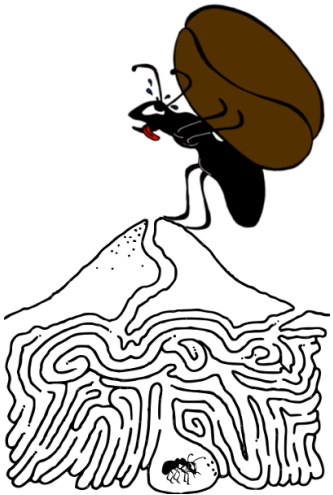
DJC Execution Step

$$\mathscr{H}; \vec{\mathscr{C}}; \vec{\mathscr{D}} \vdash T \xrightarrow[\vec{c}]{\vec{\alpha}} \mathscr{H}'; \vec{\mathscr{C}}'; \vec{\mathscr{D}}' \vdash T'$$

Outline

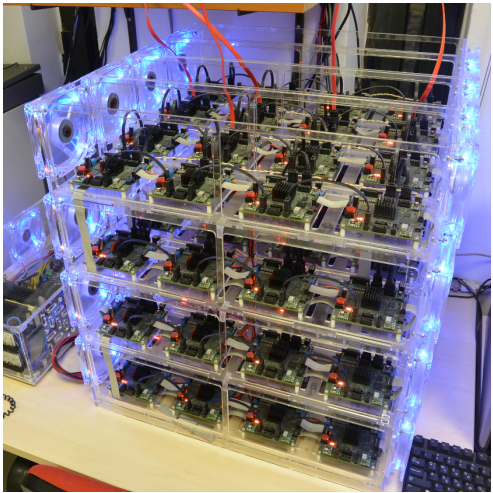
DiSquawk

- Runs on the Formic-Cube [Lyberis et al. 2012]
- Bare-metal
- Single System Image (SSI)
- Based on Squawk [Simon et al. 2006]
  - ✓ Not OS dependent
  - ✗ Targets single-core embedded devices
- Uses Myrmics' core libraries [Lyberis 2013]



The Formic-Cube

- 64-boards
- 8-cores per board
- 128MiB per board
- 512-cores in total
- 8GiB in total
- No memory coherence
  (not even on the same board)
- Emulator (CPU clock @10MHz)



Limitations

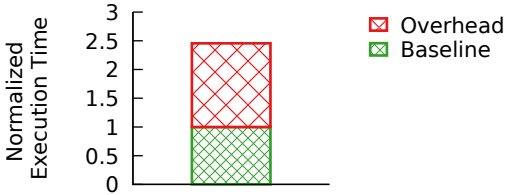
- Garbage Collection
- Files / File-system
- Sockets / Network
- `java.util.concurrent`
- Coherent Islands
- JIT compilation

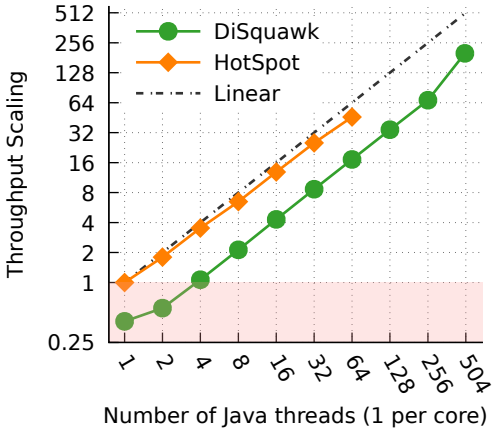
Methodology

- 4 benchmarks (Java Grande and PARSEC) + microbenchmarks
- Compare scale factor against the HotSpotVM
  - x86 64-core NUMA machine
  - Disabled JIT compilation (`-Xint`)
  - Run in server mode (`-server`)
  - Tuned heap size to avoid garbage collection
- Evaluate weak scaling

SOR (Successive Over-Relaxation)

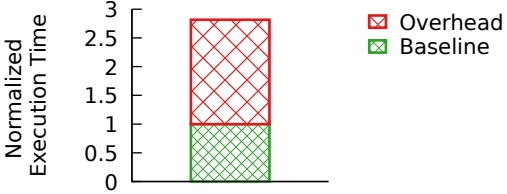
- Stencil computation
- Volatile accesses at every iteration
- Memory intensive
- 143% DiSquawk overhead on 1 thread

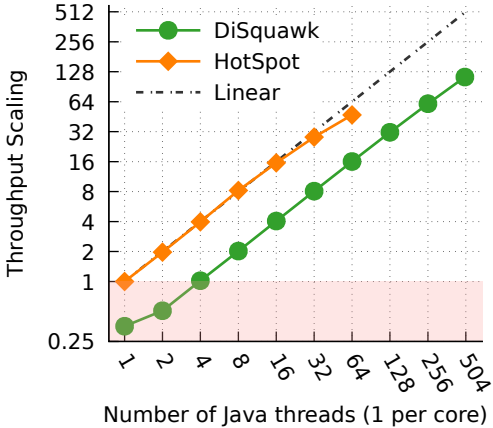




Crypt

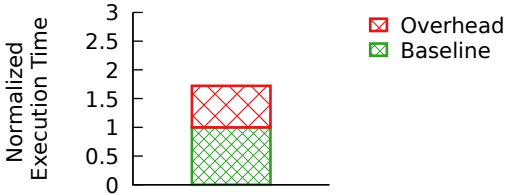
- 2-phase (1. Encrypt 2. Decrypt)
- Synchronization through barrier
- Memory intensive
- 181% DiSquawk overhead on 1 thread

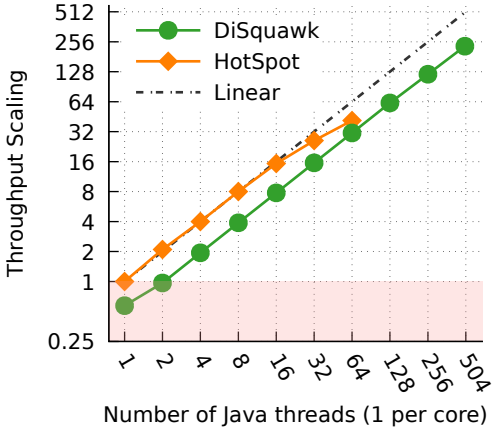




Black-Scholes

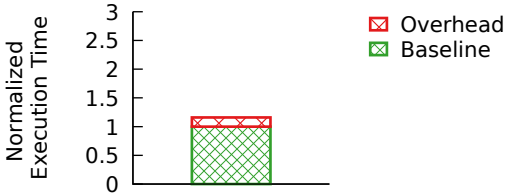
- Embarrassingly parallel
- Computation intensity > Memory intensity
- 72% DiSquawk overhead on 1 thread

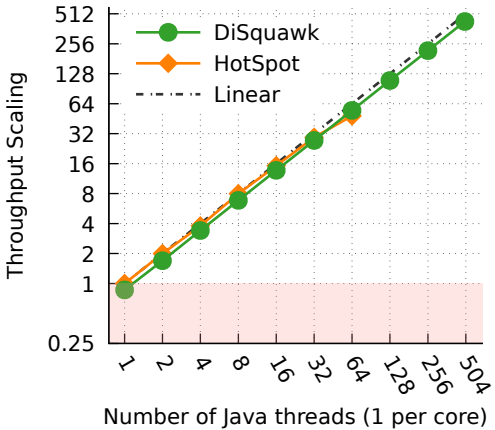




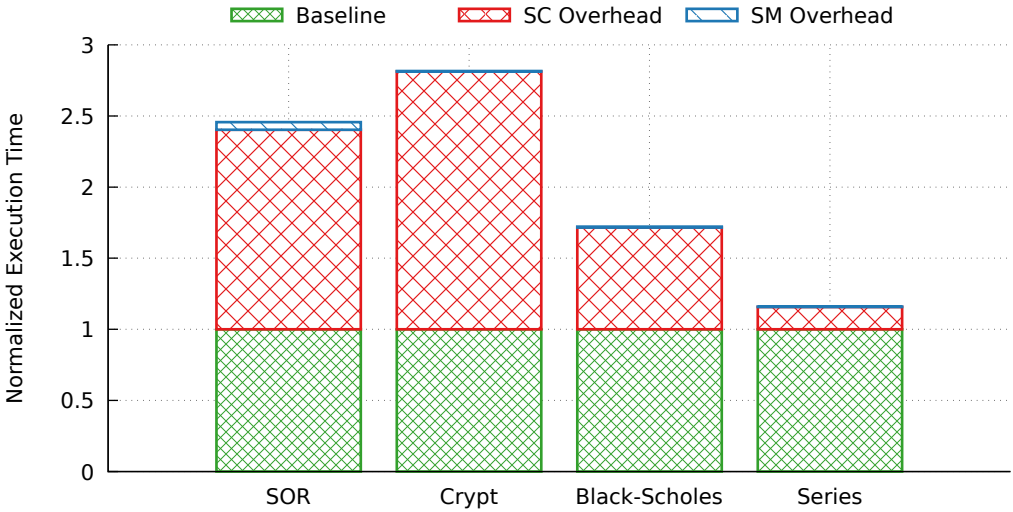
Series

- Fourier coefficients calculation
- Embarrassingly parallel
- Compute intensive
- 16% DiSquawk overhead on 1 thread





Overhead Breakdown



Outline

Takeaways

- We can run Java on non-cache-coherent architectures with hundreds of cores
- If you are willing to follow our example:
  - download DiSquawk and port it to your platform
    or
  - start by understanding JDMM
  - study our design and let us know of any optimizations you might thing of
  - use DJC to argue about your implementation's correctness

Open Research Problems

- Machine checked proofs
- Implementation & Evaluation on partially coherent architectures (*e.g.*, EUROSERVER [Durand et al. 2014])
- Use of a state-of-the-art JVM as the base
- On-the-fly/Concurrent Garbage Collection
- Study of `java.util.concurrent` and porting to future many-core architectures (started in GreenVM project)
- Java Memory Model update (ongoing JEP-188)

Thank You!

Open Research Problems

- Machine checked proofs
- Implementation & Evaluation on partially coherent architectures (*e.g.*, EUROSERVER [Durand et al. 2014])
- Use of a state-of-the-art JVM as the base
- On-the-fly/Concurrent Garbage Collection
- Study of `java.util.concurrent` and porting to future many-core architectures (started in GreenVM project)
- Java Memory Model update (ongoing JEP-188)

Thank You!

Publications

Peer reviewed

- Java Distributed Memory Model in ISMM'14
- Distributed Java Calculus in PPPJ'16
- DiSquawk Design in JTRES'16

Other

- DiSquawk open-source @github
- DJC technical report

Conferences

ISMM'14

- International Symposium on Memory Management
- Co-located with PLDI

Conferences

PPPJ'16

- 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools
- Part of Managed Languages & Runtimes week '16

Conferences

JTRES'16

- 14th International Workshop on Java Technologies for Real-time and Embedded Systems
- Part of Managed Languages & Runtimes week '16

Synchronization

- Synchronization Managers
  (on dedicated cores)
- Queuing requests when monitor not available
- Co-located threads may reuse monitors to reduce network traffic (combining)

Formalization

DJC Local Execution Step Example

$$[\textsc{Assign}]\ \frac{r \in dom(\mathscr{H}) \qquad \neg volatile(r.f) \qquad \mathscr{D}' = \mathscr{D}[r.f \mapsto v]}{\mathscr{H};\mathscr{C};\mathscr{D} \vdash c\langle r_t, r.f := v\rangle \xrightarrow{W} \mathscr{H};\mathscr{C};\mathscr{D}' \vdash c\langle r_t, v\rangle}$$

DJC Global Execution Step Example

$$[\textsc{Spawn}]\ \frac{\mathscr{H}(r_{t'}) = \langle C, \overrightarrow{f \mapsto v}\rangle \qquad \mathscr{H}'(r_{t'}) = \langle C, \overrightarrow{f \mapsto v}, \text{spawned}\rangle \qquad \text{run()}\{\text{return } e;\} \in C \qquad \vec{\mathscr{D}}(c) = \emptyset \qquad c' \in \textsc{Cids}}{\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash c\langle r_t, r_{t'}.\text{start}()\rangle \xrightarrow[\{c\}]{\{S_p\}} \mathscr{H}';\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash c\langle r_t, ()\rangle \parallel c'\langle r_{t'}, \text{start}\rangle}$$

Formalization

DJC Local Execution Step Example

$$[\textsc{Assign}]\ \frac{r \in dom(\mathscr{H}) \qquad \neg volatile(r.f) \qquad \mathscr{D}' = \mathscr{D}[r.f \mapsto v]}{\mathscr{H};\mathscr{C};\mathscr{D} \vdash c\langle r_t, r.f := v\rangle \xrightarrow{W} \mathscr{H};\mathscr{C};\mathscr{D}' \vdash c\langle r_t, v\rangle}$$

DJC Global Execution Step Example

$$[\textsc{Spawn}]\ \frac{\begin{array}{c}\mathscr{H}(r_{t'}) = \langle C, \overrightarrow{f \mapsto v}\rangle \qquad \mathscr{H}'(r_{t'}) = \langle C, \overrightarrow{f \mapsto v}, \text{spawned}\rangle \\ \text{run()}\{\text{return } e;\} \in C \qquad \vec{\mathscr{D}}(c) = \emptyset \qquad c' \in \textsc{Cids}\end{array}}{\begin{array}{c}\mathscr{H};\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash c\langle r_t, r_{t'}.\text{start()}\rangle \xrightarrow[\{c\}]{\{Sp\}} \\ \mathscr{H}';\vec{\mathscr{C}};\vec{\mathscr{D}} \vdash c\langle r_t, ()\rangle \parallel c'\langle r_{t'}, \text{start}\rangle\end{array}}$$