Cooperative Execution of Parallel Tasks with Synchronization Constraints

Shams Imam

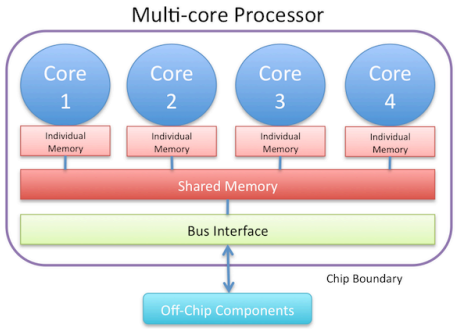
PhD Thesis Defense

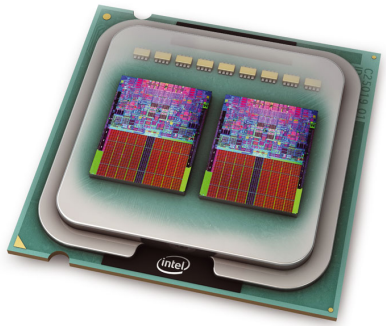
Rice University

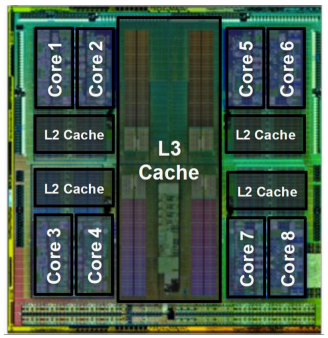
May 14, 2015

Introduction

- Multicore processors ubiquitous

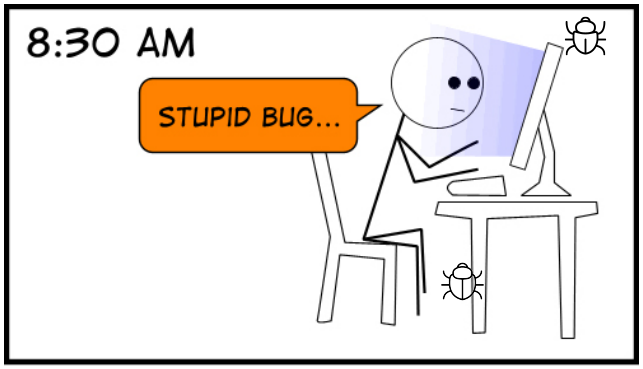






- Application performance requires parallelism

- Notoriously difficult to write performant parallel programs



Introduction…

- Parallel Programming models aid productivity
  - Offer Synchronization and Coordination constructs
  - E.g. Task Parallel Model, Actor Model

Intel® Cilk™ Plus

BERKELEY UPC UNIFIED PARALLEL C

akka

Intel® TBB

CRAY CHAPEL

jetlang
Message based concurrency for Java

ERLANG

IBM X10

OpenMP

Microsoft Asynchronous Agents Library

Issue

- Current implementations for Synchronization and Coordination constructs are either
  - Not performant
    - Poor implementations add overhead in the runtime
    - O/S kernels designed to coordinated resource sharing inter-jobs rather than intra-jobs
    - e.g. future/barrier in Task Parallel Models
  - Not productive
    - Complicate user-level code by restricting synchronization patterns
    - e.g. barriers disallowed/behavior undefined inside OpenMP parallel loops, join only permitted on child tasks in Cilk, …

Thesis Statement

*"Many **modern synchronization and coordination constructs** in parallel programs can incur significant **performance overheads** on current runtime systems, or significant **productivity overheads** when the programmer is forced to restrict the synchronization patterns that they can use.*

*Our thesis is that that the **use of cooperative scheduling techniques** can help **address the performance and productivity challenges** of using modern synchronization and coordination constructs.*

*These techniques can be implemented using **extensions** to **programming models**, **compilers**, and **runtime systems**, depending on the desired trade-offs among performance, productivity, and support for legacy code."*

Thesis Overview

Programming Models

Habanero

Structured Task ||-ism

ξureka!

Speculative ||-ism

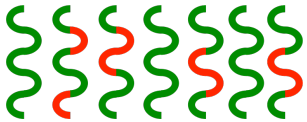
SELECTOR

Actor Extension

Runtimes

Habanero-Java Library

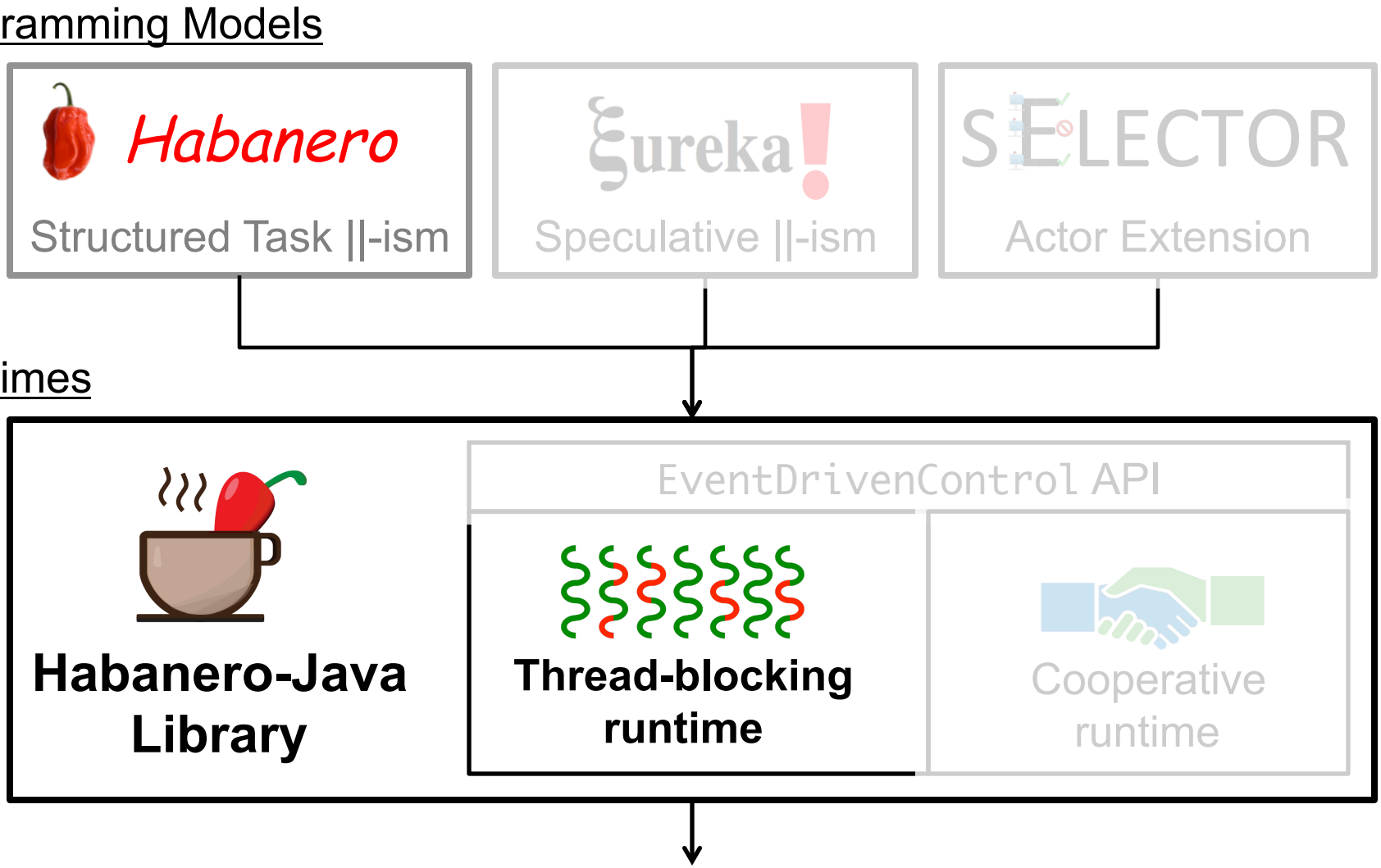
EventDrivenControl API



Thread-blocking runtime

Cooperative runtime

Habanero-Java Library



Habanero-Java Library (HJlib)

RICE

- Implementation of Habanero parallel programming model
- Library-based approach integrates easily with existing code
  - No special compiler support required
  - Java 8 lambdas simplify syntax
  - Thread-blocking runtime
    - Relies on JDK's work-stealing `ForkJoinPool`
  - Users can use IDE and debugger tools of choice
- Feedback capabilities
  - Helps programmer debug applications
  - Abstract metrics report
  - Deadlock detector

HJlib Contributions

- Higher level of abstraction with a wider range of parallel constructs than standard Java
- Pure **from-scratch** library implementation on Java 8
  - Built ground up with no external dependences
- Provides parallel programming framework for everyone
  - Implementation vehicle for concepts presented in this thesis
  - In use in multiple research projects
  - Pedagogic tool for educators

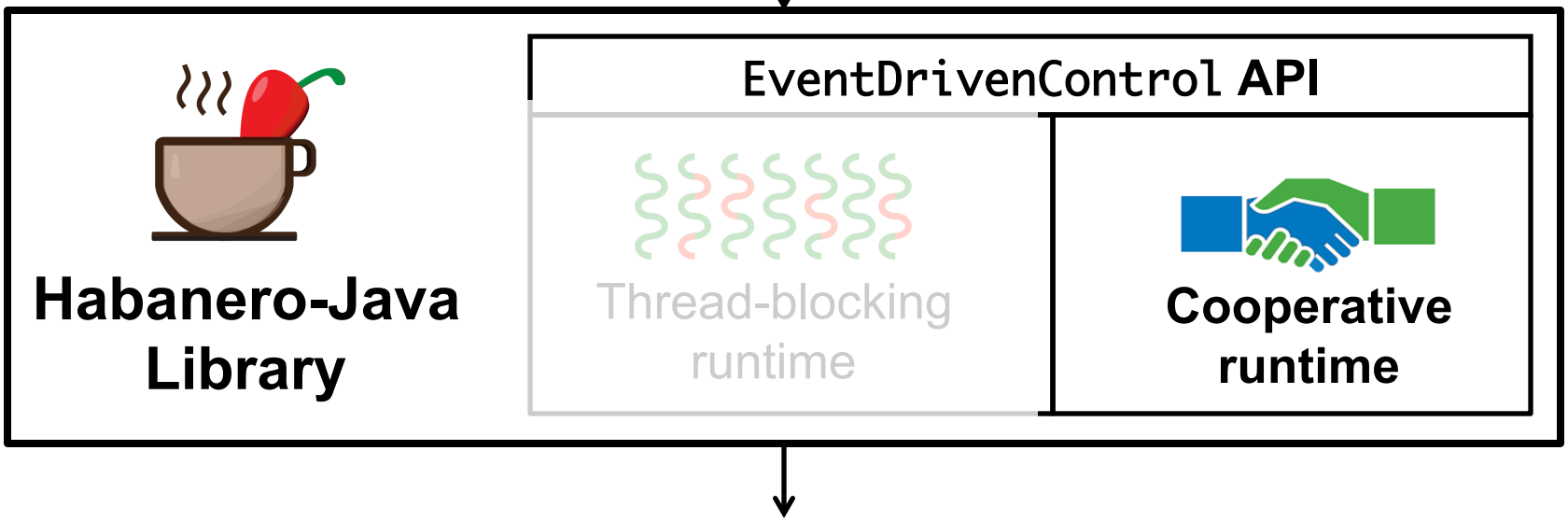
Habanero-Java Library: a Java 8 Framework for Multicore Programming. **Shams Imam**, Vivek Sarkar. 11th International Conference on the Principles and Practice of Programming on the Java Platform (**PPPJ'14**), September 2014.

Cooperative Runtime for Parallel Tasks

RICE

Programming Models

Runtimes

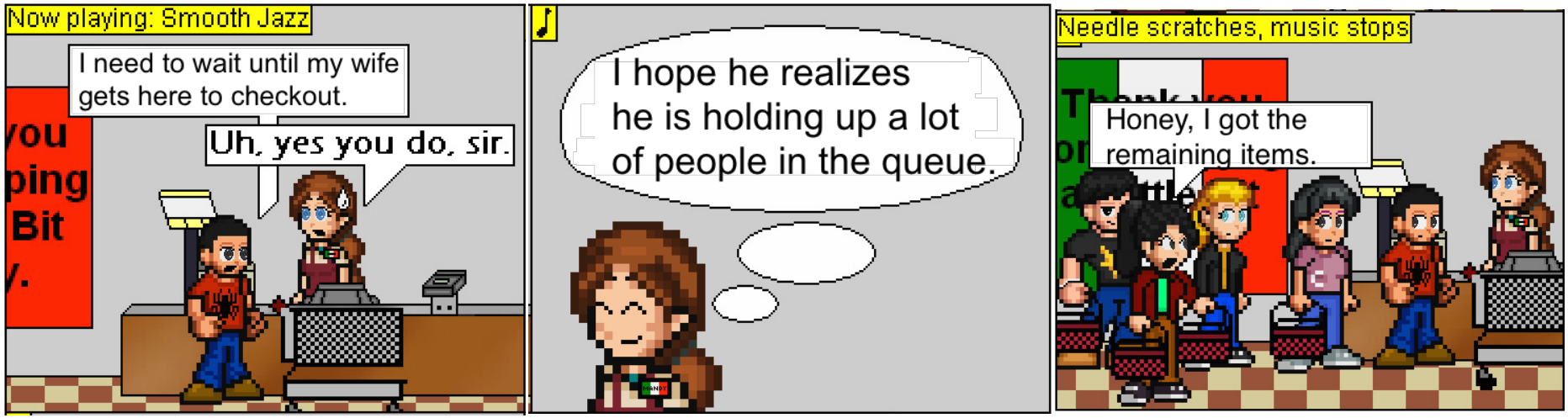


Task-Parallel Model (TPM)

- Tasks, Work Queues, and Worker Threads
- Runtime manages load balancing and synchronization

source: http://www.deviantart.com/art/Randomness-20-178737664

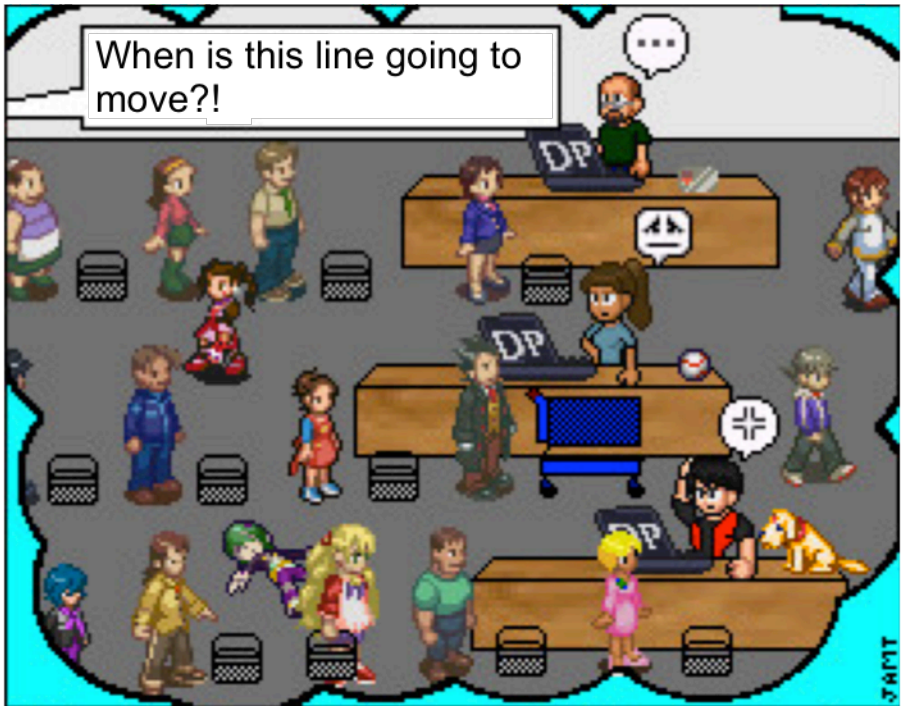
Synchronization Constraints

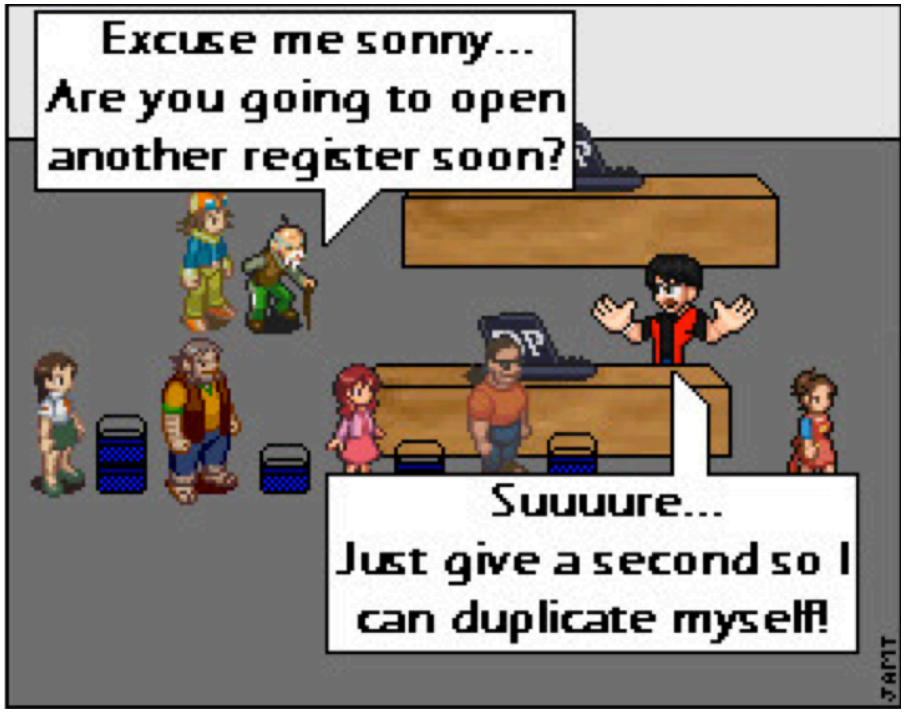


- Dependences between tasks
- Prevent an executing task from making further progress
  - Needs to synchronize with other executing task(s)

source: http://viper-x27.deviantart.com/art/Checkout-Lane-Guest-Comic-161795346

Common TPM Runtime Solution: Spawn Additional Worker Threads



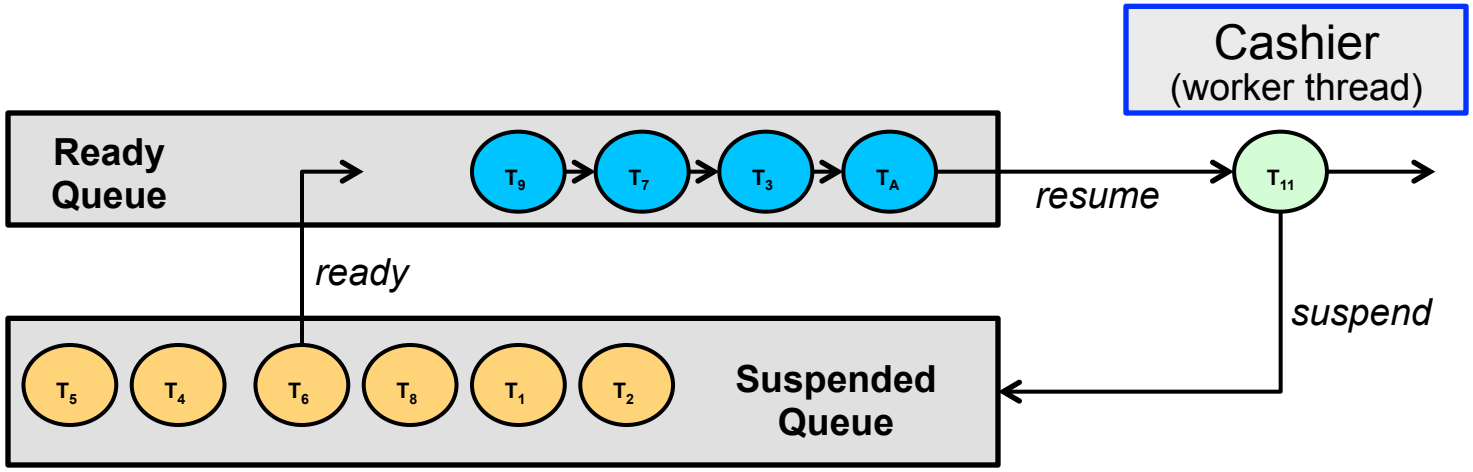


- Each thread needs its own system resources
- Exhaustion of memory or other system resources
- Scheduler tangled in thread scheduling decisions
- Thread-blocking approaches do not scale!

source: http://www.deviantart.com/art/Randomness-5-90424754

Proposed Solution: Cooperative Scheduling

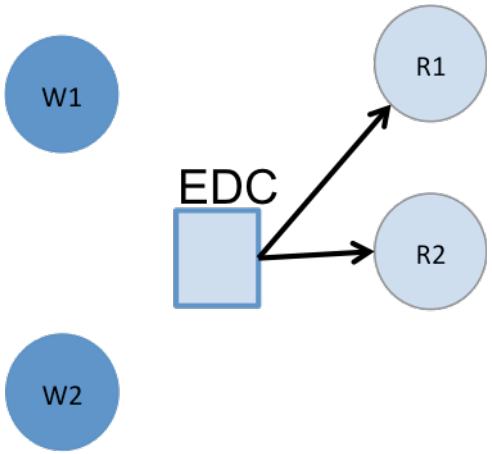
- A **Cooperative** Approach is more efficient
- Task suspended by runtime at synchronization points
  - Task Continuation (TC) moved to the suspended queue
  - Compiler support needed to generate the continuations
- TC moved to ready queue when the task can make progress
- TC resumed when executed on worker thread



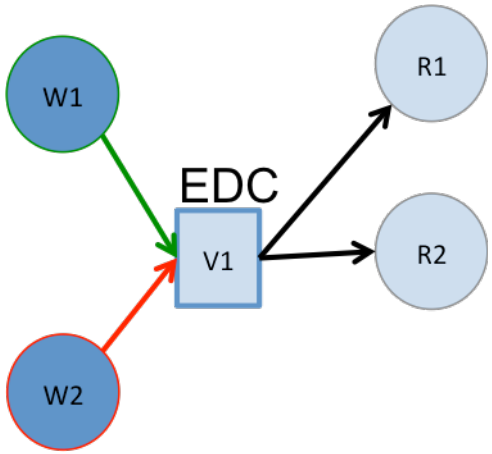
Cooperative Scheduling: http://en.wikipedia.org/wiki/Computer_multitasking#Cooperative_multitasking

Event-Driven Control (EDC)

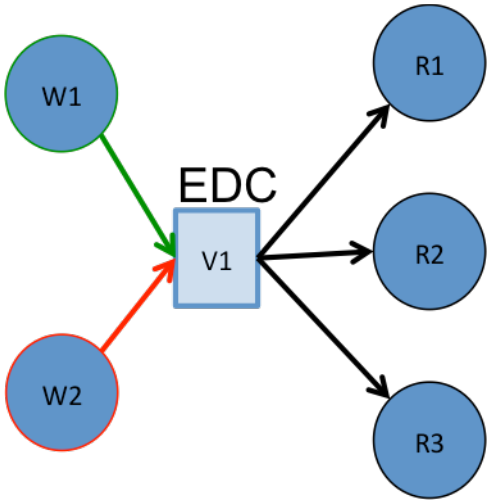
- Delays execution of readers until value available
- Dynamic single-assignment of value (event) by writer



1. Initially empty, readers suspend



2. Some writer succeeds, value becomes available

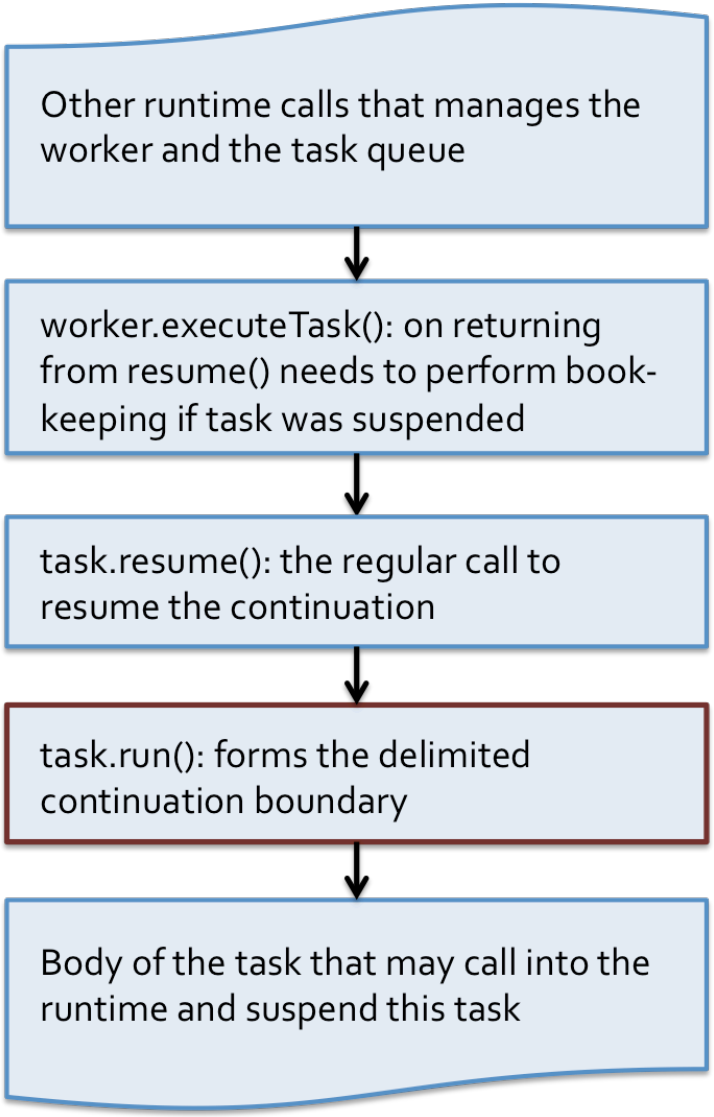


3. All readers execute, subsequent readers do not get suspended

Cooperative Runtime – Call Stack

RICE

- **Help-first** policy
  - Task has a stack of its own
  - Task can be executed by any of the worker threads
- Task wrapped to form One-shot Delimited Continuations
- Worker thread manages when tasks get
  - **cooperatively** added to suspended queue
  - removed from ready queues for execution by worker threads



Cooperative Runtime

- We expose EDCs as an internal API in our runtime
  - Read / Write / Query on value
  - Suspend till value becomes available
- Continuations not exposed to developer
  - Notorious for being hard to use and to understand
- Developers write thread-based code
  - Compiler handles code transformations to support continuations
  - One-shot delimited continuations implemented more efficiently than general continuations

Reducing Overheads compared to Thread-Blocking

RICE

- Creating additional worker threads is contradictory to goal of TPM
- 1100 ns per thread context switch (without cache effects)

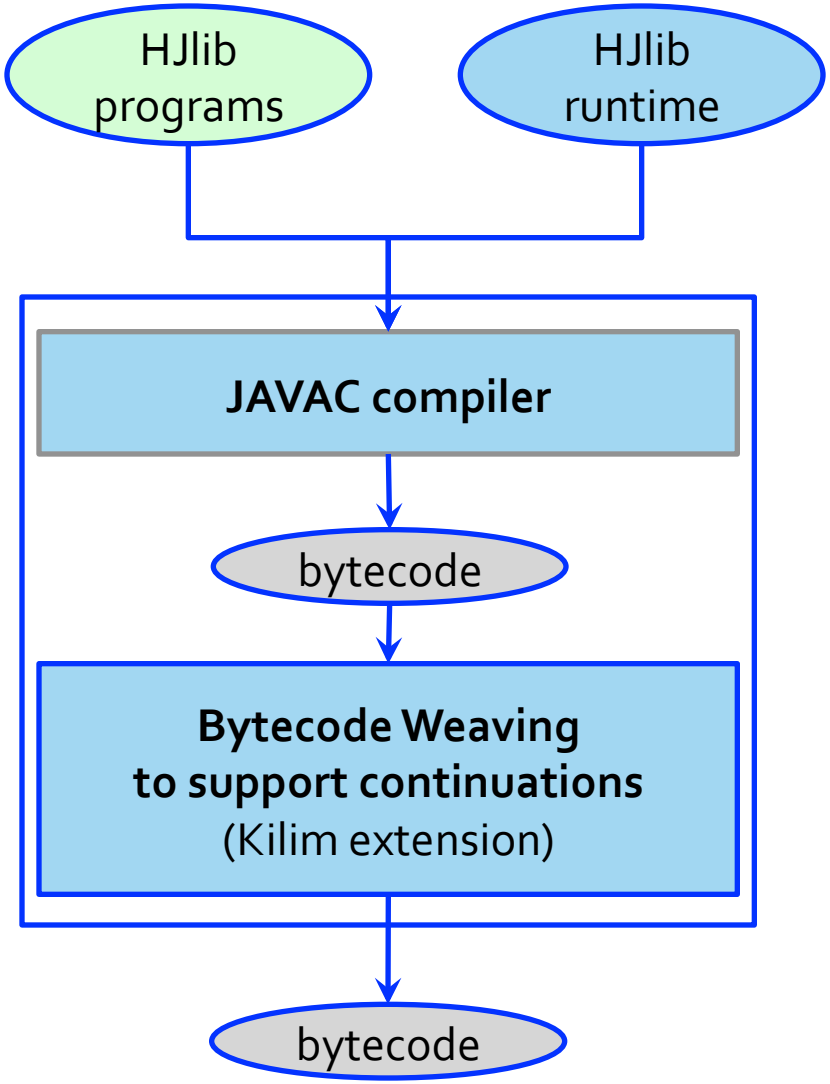
- In contrast
  - Object allocation takes around 30 ns
  - Method call takes around 5 ns
  - Setting fields takes around 1 ns

- Key assumption of this work
  - Creating threads and context switching is expensive
  - Task creation and allocating synchronization objects are cheaper

Implementation: Compiler Infrastructure

RICE

- Standard Java code as input
- First compilation phase uses `javac` compiler to generate bytecode
- Second Compilation phase CPS transforms code to support continuations
  - This phase can also be done during class-loading time
- Output is bytecode that runs on any standard JVM

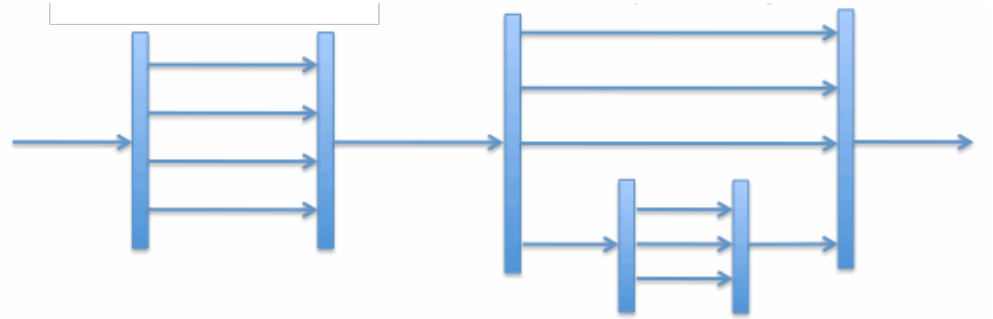


Synchronization Constructs

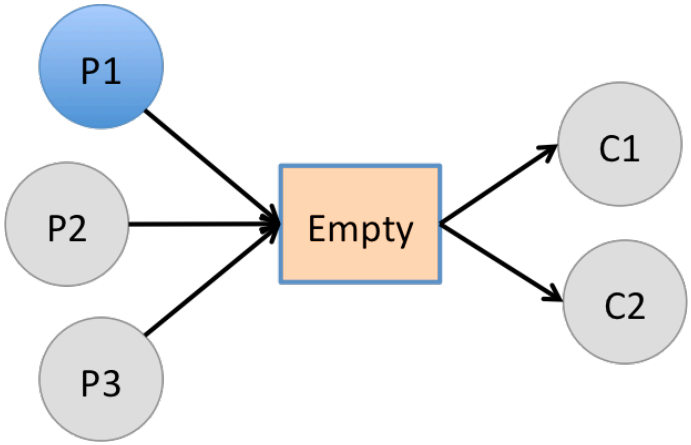
- Key idea is to:
  - Translate the synchronization constraints into producer-consumer constraints on EDCs
  - Use Delimited Continuations to suspend consumers when waiting on item(s) from producer(s)
- Any task-parallel Synchronization Constraint can be supported.
  - Both deterministic and non-deterministic constructs
  - Including `async-finish`, `futures`, barriers, `atomic/isolated` and actors

Synchronization constructs…

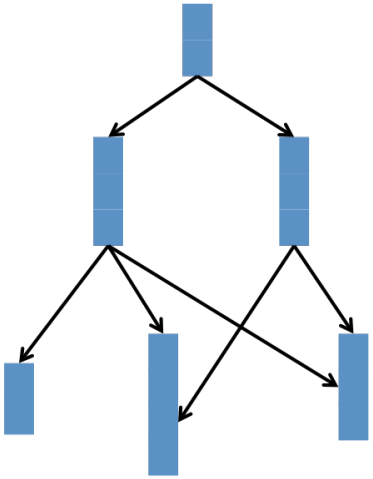
Nested fork-join



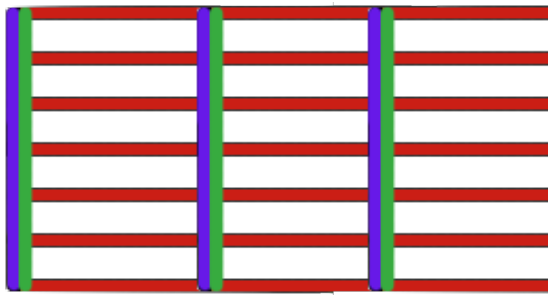
Synchronization Variable



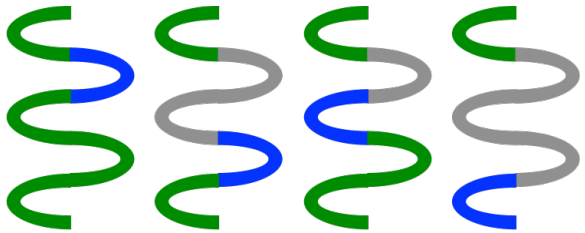
Futures



Barriers

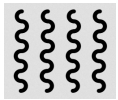


Critical-Section blocks

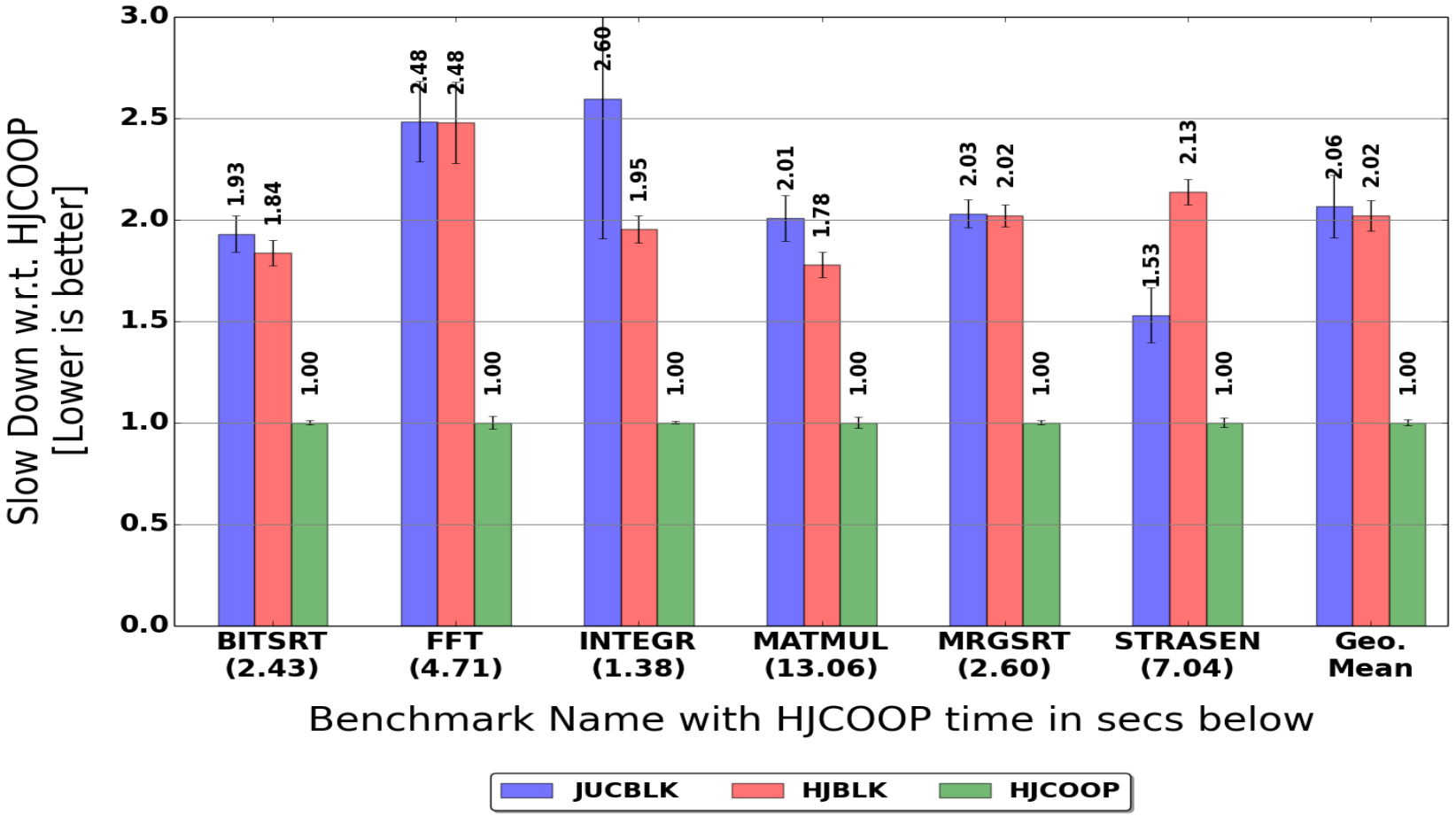


- Recipes for these and other constructs described in thesis…

Experimental Setup

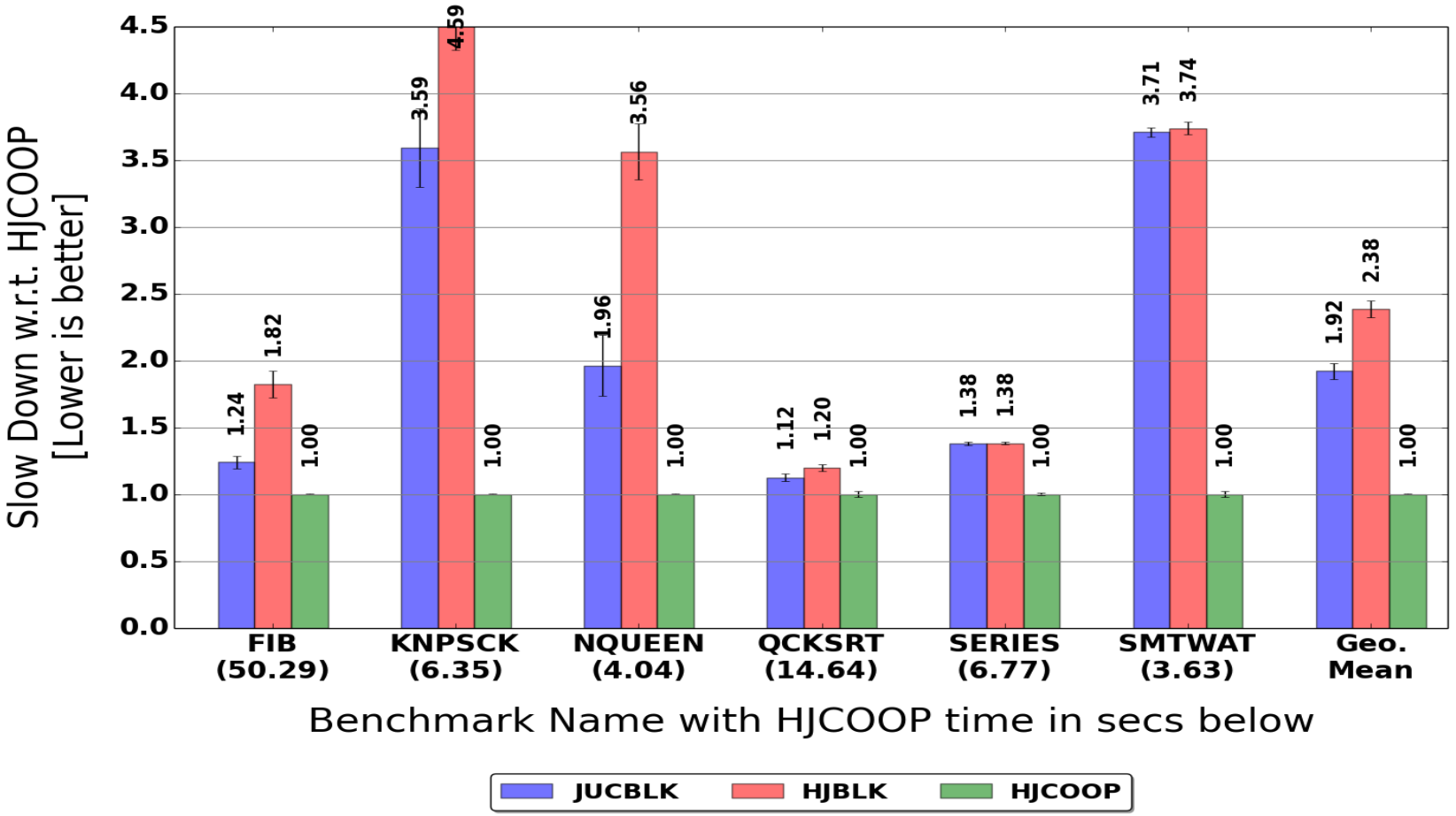
-  Four 8-core 3.8 GHz IBM POWER7
-  256 GB of RAM
-  32 KB L1 Cache
- Software
  -  IBM Java SDK Version 1.8.0
  -  Habanero-Java library 0.1.2
    - Thread-blocking scheduler
    - Cooperative scheduler enabled via option
- Benchmarks run with single place
  -  32 worker threads per place
  - 64 GB memory allocated to JVM
  -  Mean of best 30 out of 100 execution times reported

async-finish Benchmarks



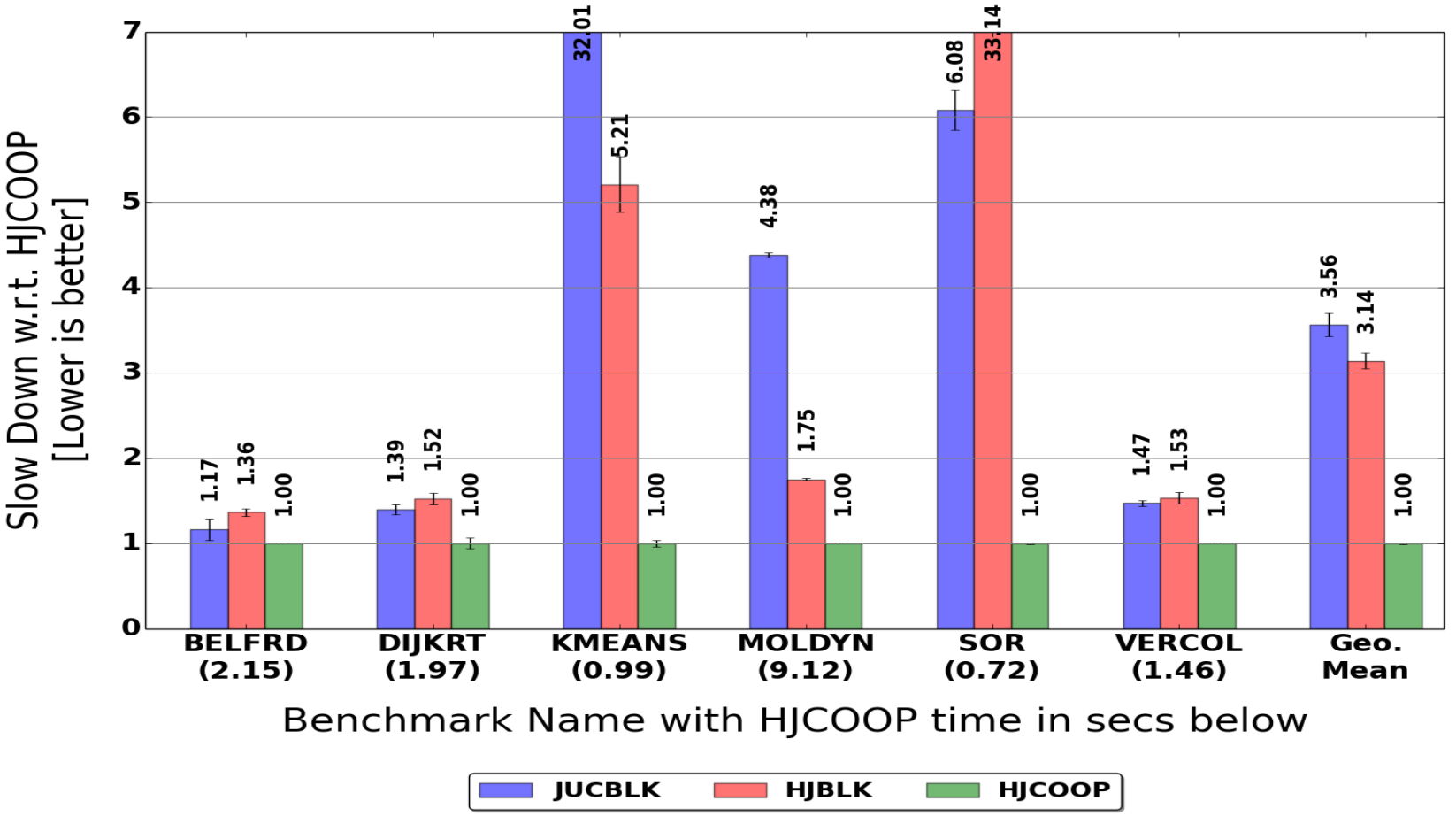
- Benchmarks that create lots of dynamic finish scope instances chosen
- Thread-blocking runtimes crash on larger inputs

future Benchmarks



- Benchmarks that eagerly attempt to resolve futures chosen
- Thread-blocking runtimes crash on larger inputs

Phaser Benchmarks



- Benchmarks chosen such that more than 32 tasks are registered on phaser

Related work

- Qthreads [IPDPS '08]
  - Continuations with lightweight call stack stitching
  - Stack size configured manually
- Glasgow Haskell Compiler [Haskell '07]
  - Provides continuation support directly
  - Uses polling to resume continuations
- C++ implementation of X10 [PPoPP '12]
  - Work-first policy with dedicated implementation for each construct
  - Implementation supports async-finish and futures
  - Implementation did not support clocks (phasers)

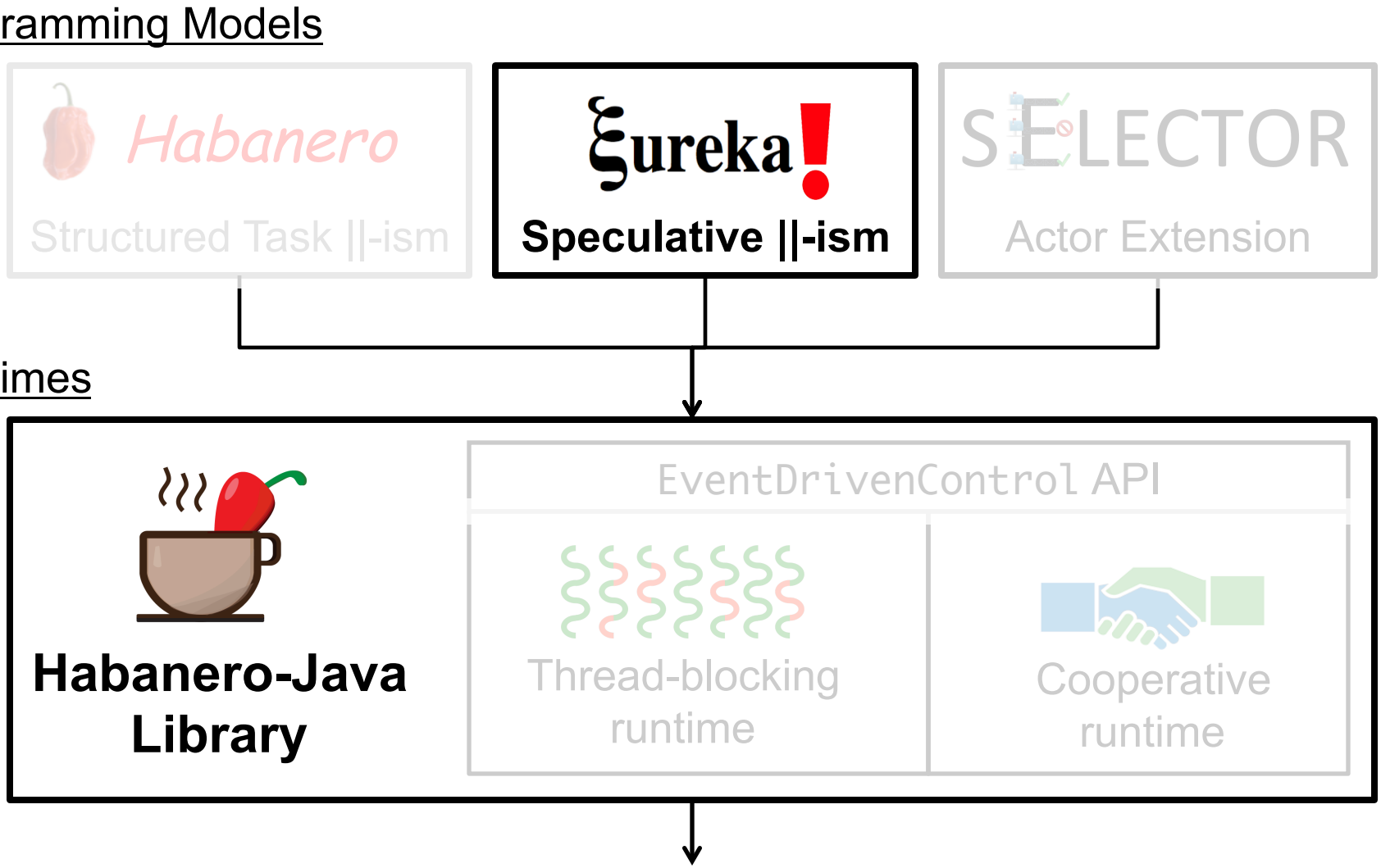
Cooperative Runtime Contributions

RICE

- **Event-Driven Control**s API
  - Can support **any** task-parallel synchronization construct
  - Recipes for various task-parallel synchronization constructs
- **Cooperative** runtime for scheduling tasks
  - Transparent use of One-shot Delimited Continuations
  - Supports triggering the **enablement of multiple** suspended tasks
- **Performs better** than runtimes that use thread-blocking

Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. **Shams Imam**, Vivek Sarkar. 28th European Conference on Object-Oriented Programming (**ECOOP**), July 2014.

Eureka Programming Model



Eureka Style Computation

- Mostly optimization and search problems
- Speculative parallelism abundant
  - Results may or may not be needed
  - Each task attempts to find a result
- Announces result has been found
  - An "aha!" moment – **Eureka** event
  - Can curtail rest of the computation

Image source: http://www.netstate.com/states/mottoes/images/ca_eureka.jpg

Simple Approach: Matrix Search

RICE

```
6   def doComputation(matrix, goal) {
7     val token = atomicRefFactory()
8     finish
9       for rowIndices in matrix.chunks()
10        async
11          for (r in rowIndices)
12            processRow(matrix(r), r, goal, token)
13    // return either [-1, -1] or a valid index [i, j]
14    return token.get()
15  }
16  def processRow(rowData, r, goal, token) {
17    for (c in rowData.indices())
18      if goal.matches(rowData(c))  // eureka!!!
19        token.set([r, c])
20        return
21  }
```

Related Work: Function-Scoped termination

- Cancellation at lexical scope of task creation
- E.g. Cilk *abort*, OpenMP 4.0

```
1   async
2     for (r in rowIndices)
3       processRow(matrix(r), r, goal, token)
4       check(token) // check cancellation

6   def processRow(rowData, r, goal, token) {
7     for (c in rowData.indices())
8       if goal.matches(rowData(c))
9         token.set([r, c])
10        return
11  }
```

Can trigger cancellation here

Cannot trigger cancellation in this lexical block

- Cancellation not possible from nested function call!

Related Work: Exceptions for Control Flow

- Throw exceptions to terminate tasks
- Only edit specific program points
- E.g. MS Task Parallel Library, Java Threads

```
1   async
2     for (r in rowIndices)
3       try
4         processRow(matrix(r), r, goal, token)
5       catch (ex: Exception)
6         exceptionHandling(ex)

8   def processRow(rowData, r, goal, token) {
9     for (c in rowData.indices())
10      if goal.matches(rowData(c))
11        token.set([r, c])
12        throw EurekaException()
13  }
```

EurekaException caught by user!

- User may inadvertently catch exception

Related Work:

Manual termination via Cancellation Tokens

- Manual periodic checks with `returns`
- User controls responsiveness

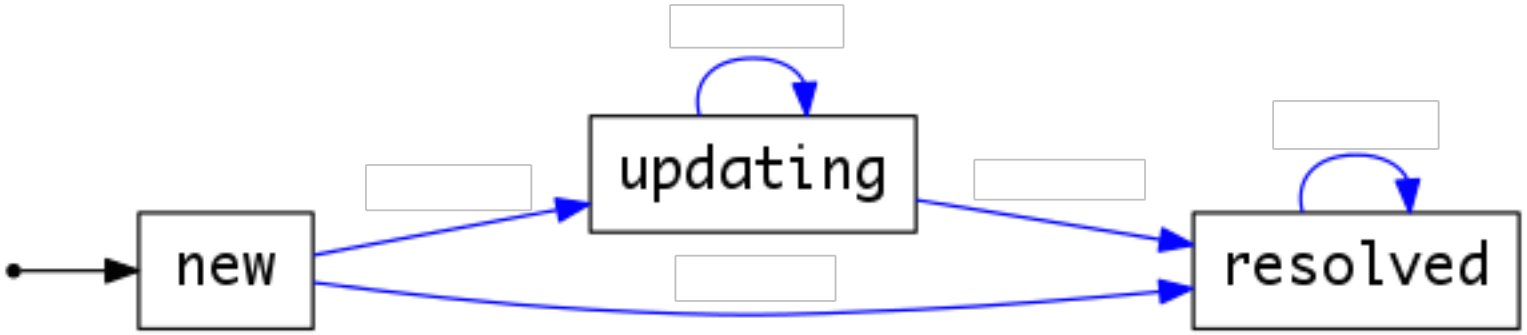
```
1   async
2     for (r in rowIndices)
3       if token.eureka()
4         return
5       processRow(matrix(r), r, goal, token)

7   def processRow(rowData, r, goal, token) {
8     for (c in rowData.indices())
9       if token.eureka()
10        return
11      if goal.matches(rowData(c))
12        token.set([r, c])
13  }
```

Repeated checks which are written manually

- Cumbersome to write
  - Pass-through methods need instrumentation
- Impossible to support inaccessible functions

Detecting eureka events

- **Our solution:** Eureka construct to abstract various patterns
- Simplifies grouping of speculative tasks



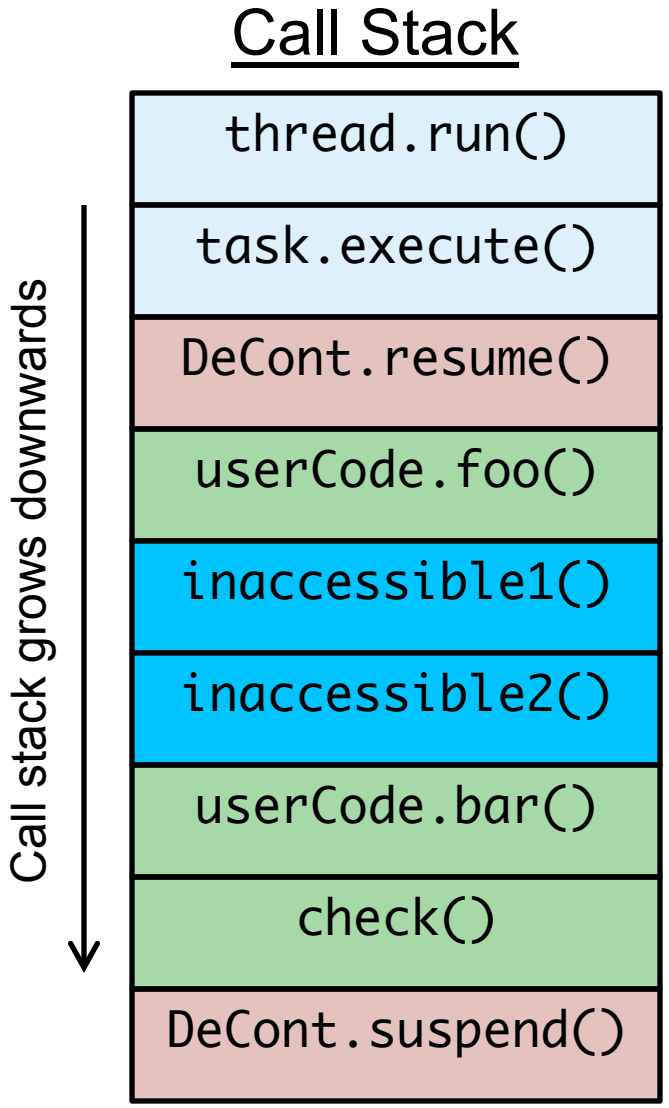
  - NEW: freshly created
  - UPDATING: changeable state
  - RESOLVED: stable state
- Desired invariant: Monotonicity in state transitions
- Once resolved, eureka always stays resolved

Eureka Programming Model

- `finish (eureka) S1`
  - Multiple `finish`es can register on same Eureka
  - Wait for all tasks **or** until eureka is resolved
- `async`
  - Launches an asynchronous task
  - Inherit Eureka registration from immediately-enclosing `finish`
- `offer()`
  - Triggers eureka event on registered Eureka
- `check()`
  - Causes task to terminate if Eureka resolved

Task Termination Technique

- **Our solution**: Use Delimited Continuations!
  - Explained earlier
  - Even cheaper – no need to save state during suspension
- Rely on cooperative task termination techniques
  - **check** status of eureka construct
  - **terminate** task by suspending task



Eureka Version of Example

```
5   def doComputation(matrix, goal) {
6     val eu = eurekaFactory()
7     finish (eu) // eureka registration
8       for rowIndices in matrix.chunks()
9         async
10          for r in rowIndices
11            processRow(matrix(r), r, goal)
12    return eu.get()
13  }
14  def processRow(rowData, r, goal) {
15    for c in rowData.indices()
16      check([r, c]) // cooperative termination check
17      if goal.matches(rowData(c))
18        offer([r, c]) // trigger eureka event
19  } }
```

Eureka Variants

- Both deterministic and nondeterministic computations
- Examples
  - Parallel Search
  - K-Count Eureka
  - N-Version Programming
  - Branch and Bound Optimization
  - Soft Real-Time Deadlines
  - Convergence Iterations
  - Binary Composition of variants
  - Nesting computations
  - …

Eureka Variants…

```
def eurekaFactory() {
  val initValue = [-1, -1]
  return new SearchEureka(initValue)
}
```

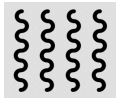
```
def eurekaFactory() {
  val K = 4
  return new CountEureka(K)
}
```

```
def eurekaFactory() {
  // comparator to compare indices
  val comparator = (a, b) -> {
    ((a.x - b.x) == 0) ? (a.y - b.y) : (a.x - b.x)
  }
  val initValue = [INFINITY, INFINITY]
  return new MinimaEureka(initValue, comparator)
}
```

```
def eurekaFactory() {
  val time = 4.seconds
  return new TimerEureka(time)
}
```

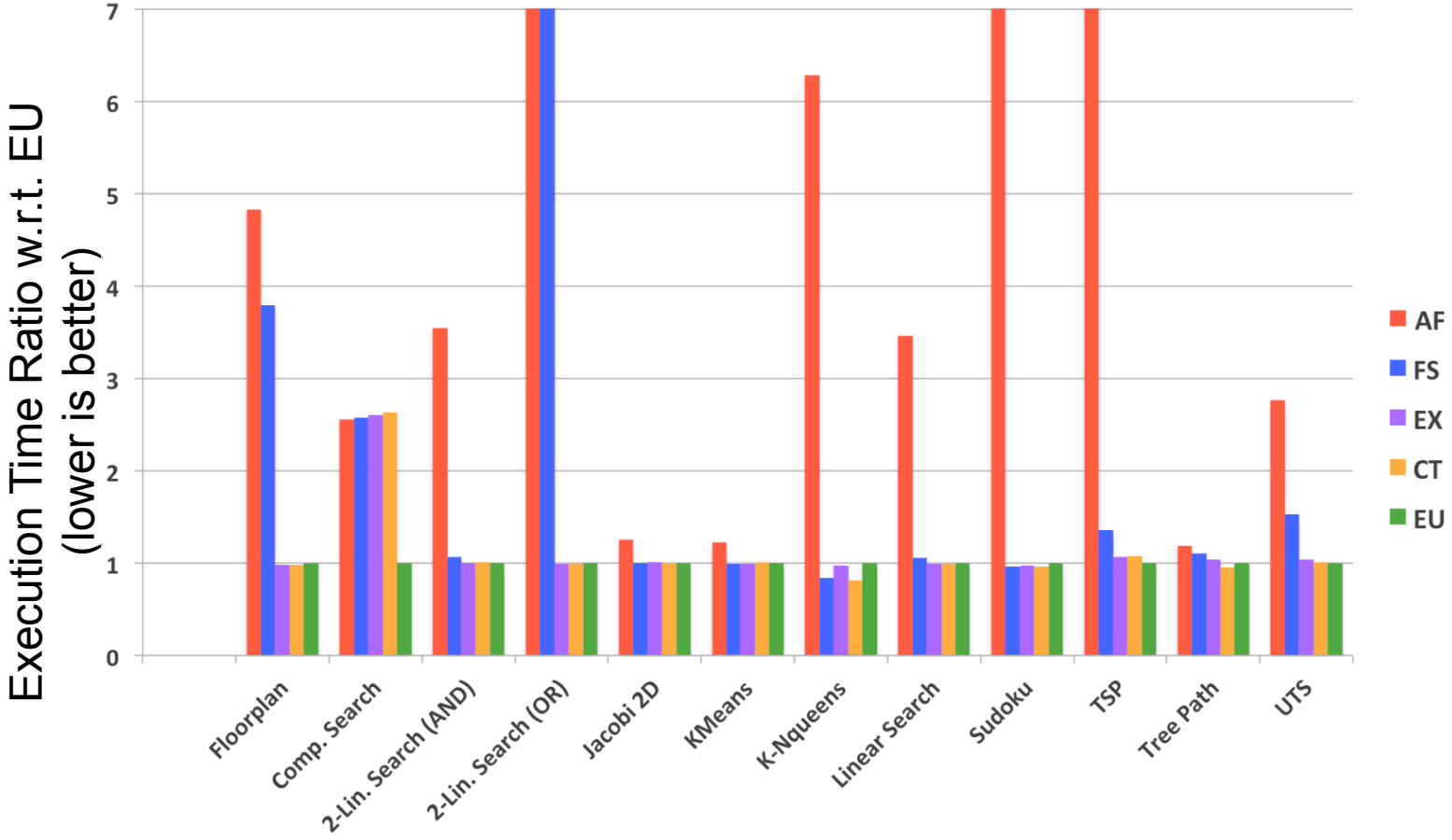
```
def eurekaFactory() {
  val units = 400
  return new EngineEureka(units)
}
```

Experimental Setup

-  Four 8-core 3.8 GHz IBM POWER7
-  256 GB of RAM
-  32 KB L1 Cache
- Software
  -  Habanero-Java library 0.1.4
    - Cooperative scheduler that uses continuations
    - Other variants implemented from scratch
- Benchmarks run with single place
  -  32 worker threads per place
  - 64 GB memory allocated to JVM
  -  Mean of best 50 out of 150 execution times reported

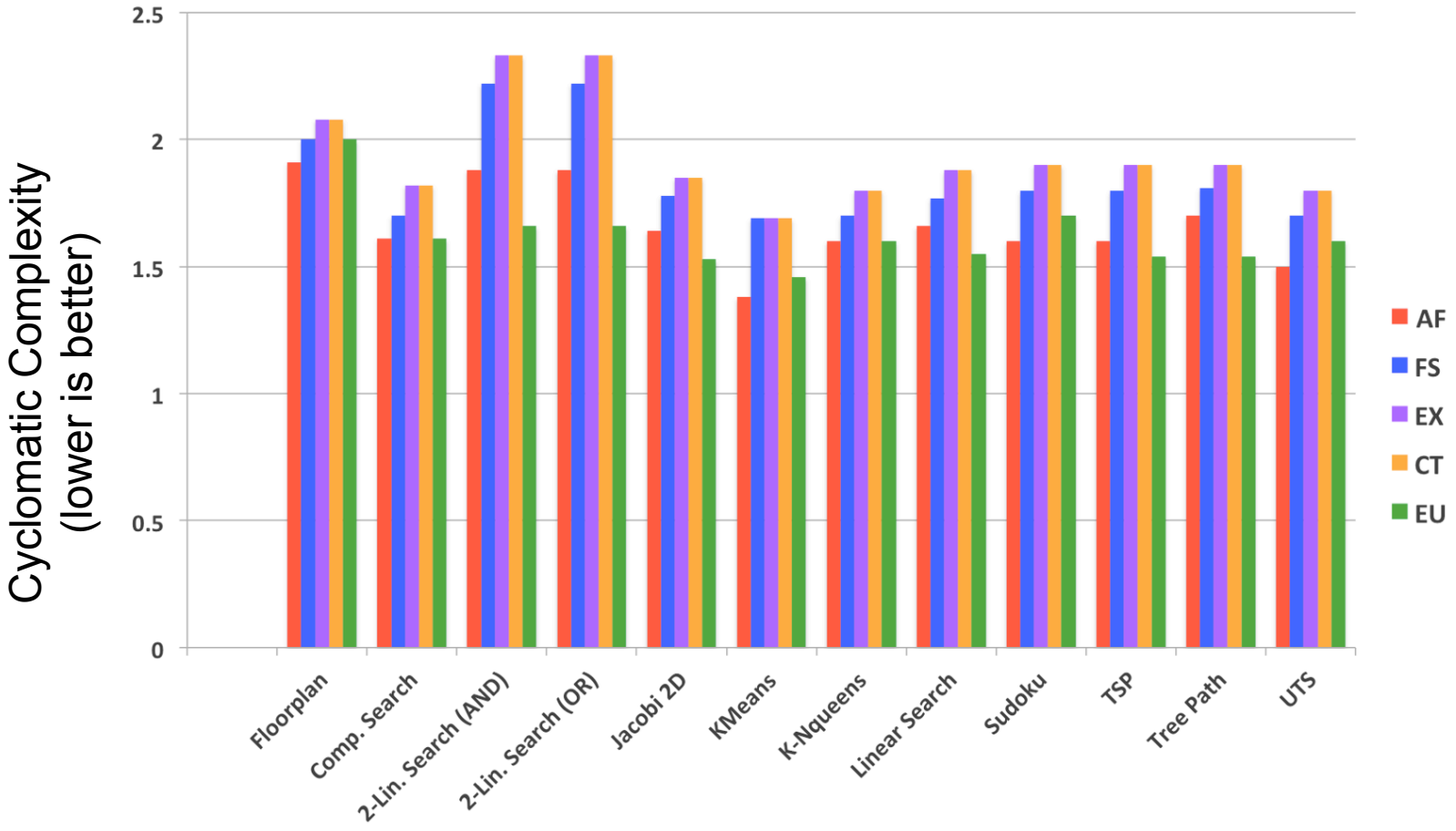
Performance Results

- Legend: AF: `async-finish`; CT: Cancellation-token; EX: Exception-based; FS: Function-scoped; EU: Eureka.



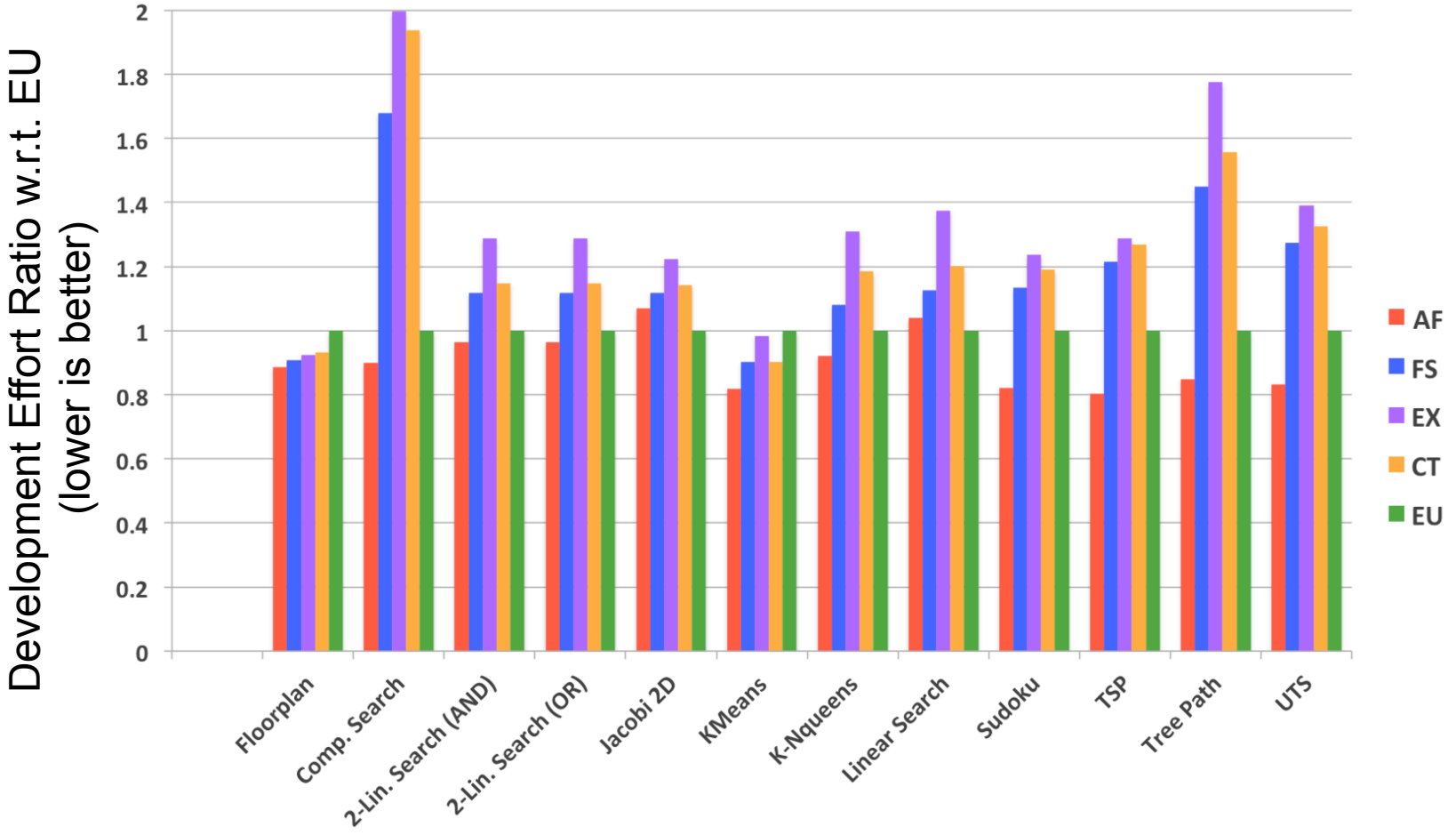
- EU slightly faster than EX and CT, about 50% faster than FS

Productivity – Cyclomatic Complexity



- EU similar to AF, at least 10% better than FS, EX, and CT

Productivity – Halstead's Development Effort



- EU 10% more than AF, 15% lower than FS, EX, and CT.

Eureka Contributions

- **Generalized** Eureka-Style Computations
  - Identified various Eureka patterns
- Proposed Eureka Programming Model
  - Rely on **cooperative** termination techniques
  - Approach works in the presence of inaccessible functions
- Evaluation shows implementation is **effective**
  - Despite using unmodified VM
- Displayed **productivity** benefits via metrics (CC & DE)

The Eureka Programming Model for Speculative Task Parallelism.
Shams Imam, Vivek Sarkar. 29th European Conference on Object-Oriented Programming (**ECOOP**), July 2015.

Selectors Programming Model

RICE

Programming Models

Habanero

Structured Task ||-ism

ξureka!

Speculative ||-ism

SELECTOR

Actor Extension

Runtimes

Habanero-Java Library

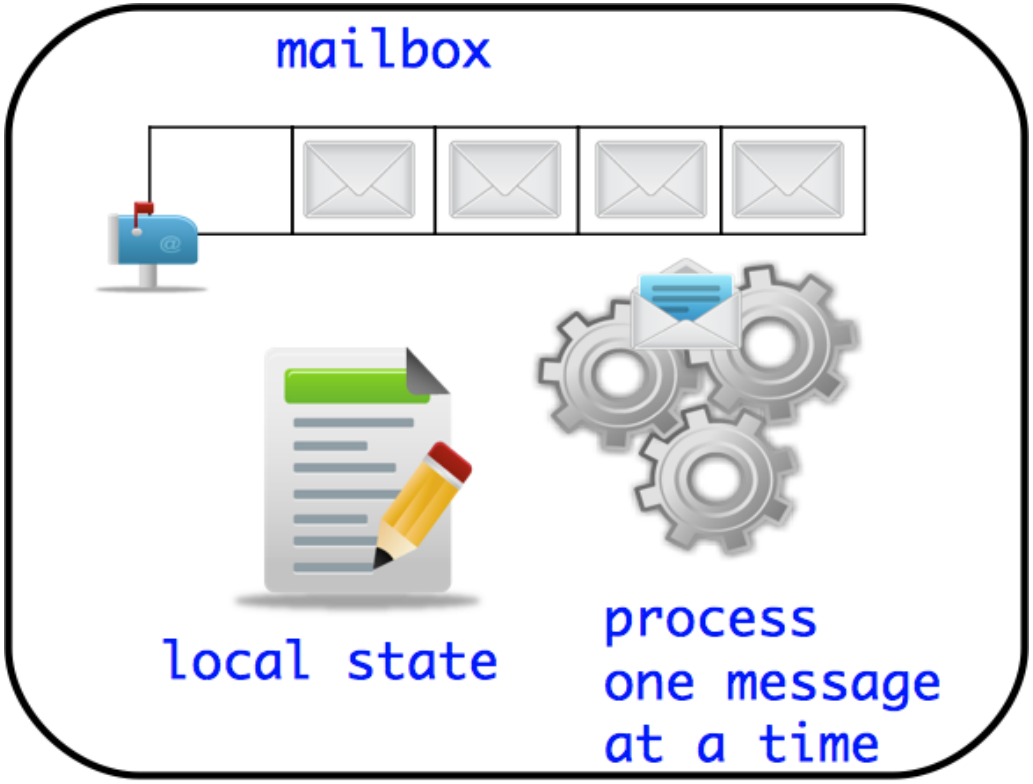
EventDrivenControl API

Thread-blocking runtime

Cooperative runtime

Actors

- Asynchronous message-based concurrency model
- Processes one message at a time



Motivation for Selectors

- Actors give strong guarantees about asynchrony and concurrency, but make synchronization and coordination harder
  - Compared to shared-memory model
  - Coordination patterns involving multiple actors are particularly difficult
  - Actor Model (AM) is not a silver bullet
- Until message is processed, solutions may require the actor to
  - Buffer messages
  - Resend messages to itself and rely on fairness guarantees

Problematic Synchronization Patterns

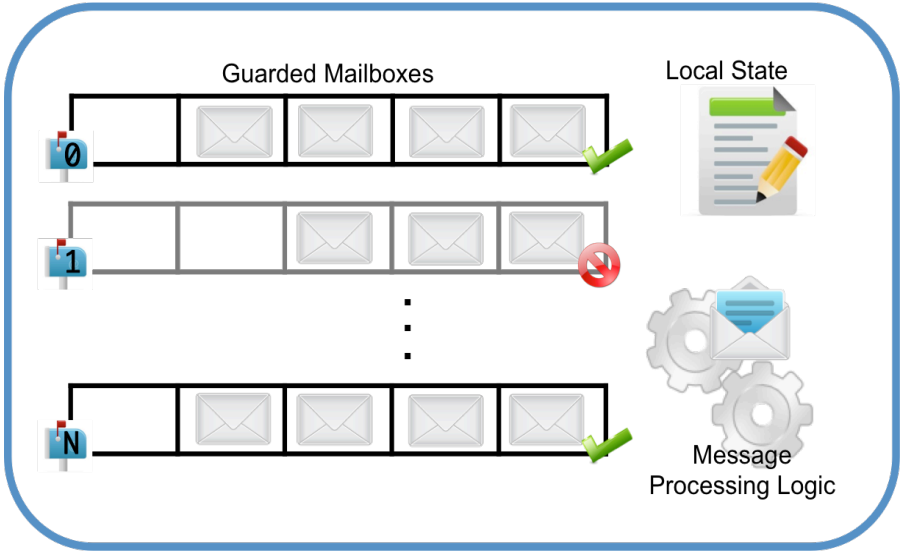
RICE

- Patterns of interest
  - **Join patterns common in streaming applications**
  - **Synchronous request-reply**
  - Priorities in message processing
  - Variants of reader-writer concurrency
  - Producer-consumer with bounded buffer
- Two of these patterns were studied in Master's thesis
  - Relied on different solution of integrating actors and task parallelism

Integrating Task Parallelism with Actors. **Shams Imam**, Vivek Sarkar. ACM International Conference on Object oriented programming systems languages and applications (**OOPSLA '12**), October 2012.

Selector

- **Multiple** mailboxes
  - Messages can be concurrently sent to different mailboxes
- Each mailbox maintains a mutable **guard**
  - Mailbox can always receive messages
  - Internal guard changed **cooperatively**
  - Affects which mailboxes provides next message to process



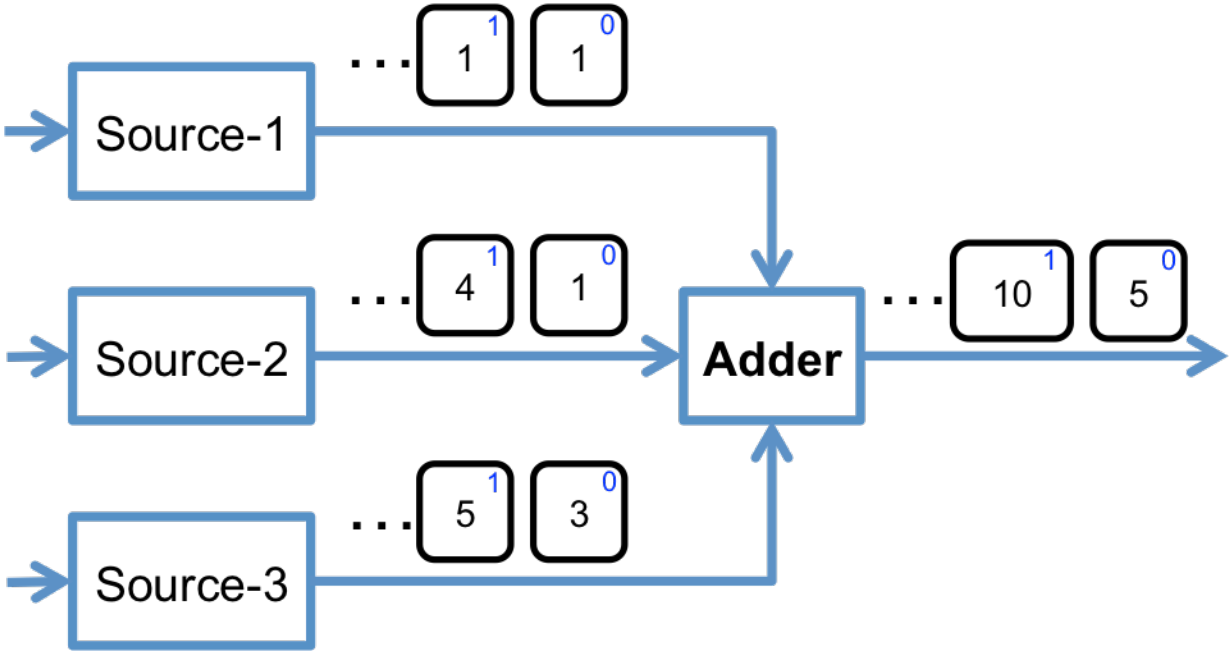
Actor is a Selector with a single mailbox!

- Guard on the mailbox always enabled

Join Patterns in Streaming Applications

RICE

- Messages from two or more data streams are combined together into a single message
- Joins need to match inputs from each source
- Wait until all corresponding inputs become available



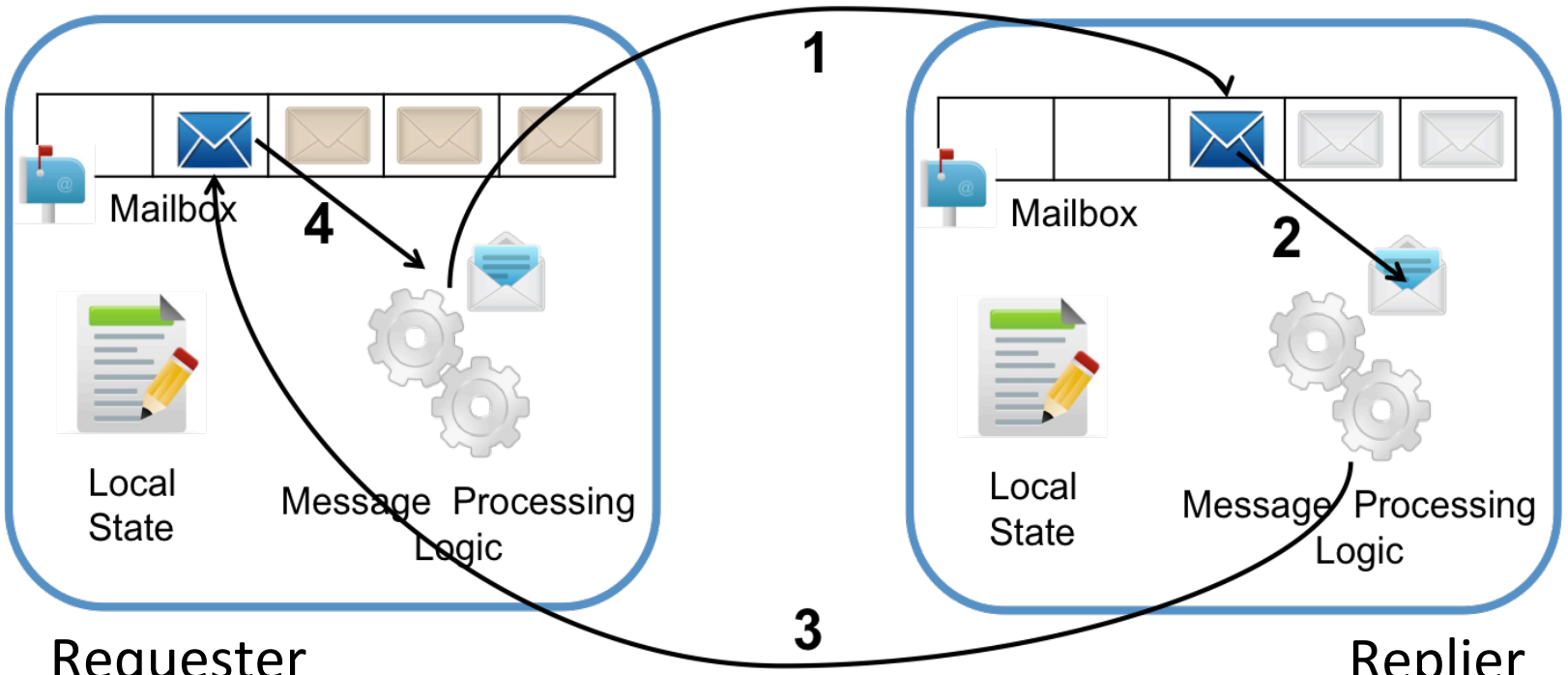
Join Pattern: Selector-based Solution

RICE

- One mailbox for each source
- Sources send their messages to corresponding mailboxes
- Two policies
  - Round-Robin order (RR)
    - Receive messages from ordered source for the current sequence
    - Useful if the join actor is performing associative reduction
  - Arbitrary Order (AO)
    - Receive messages from any source for the current sequence
    - Especially useful for non-associative reductions

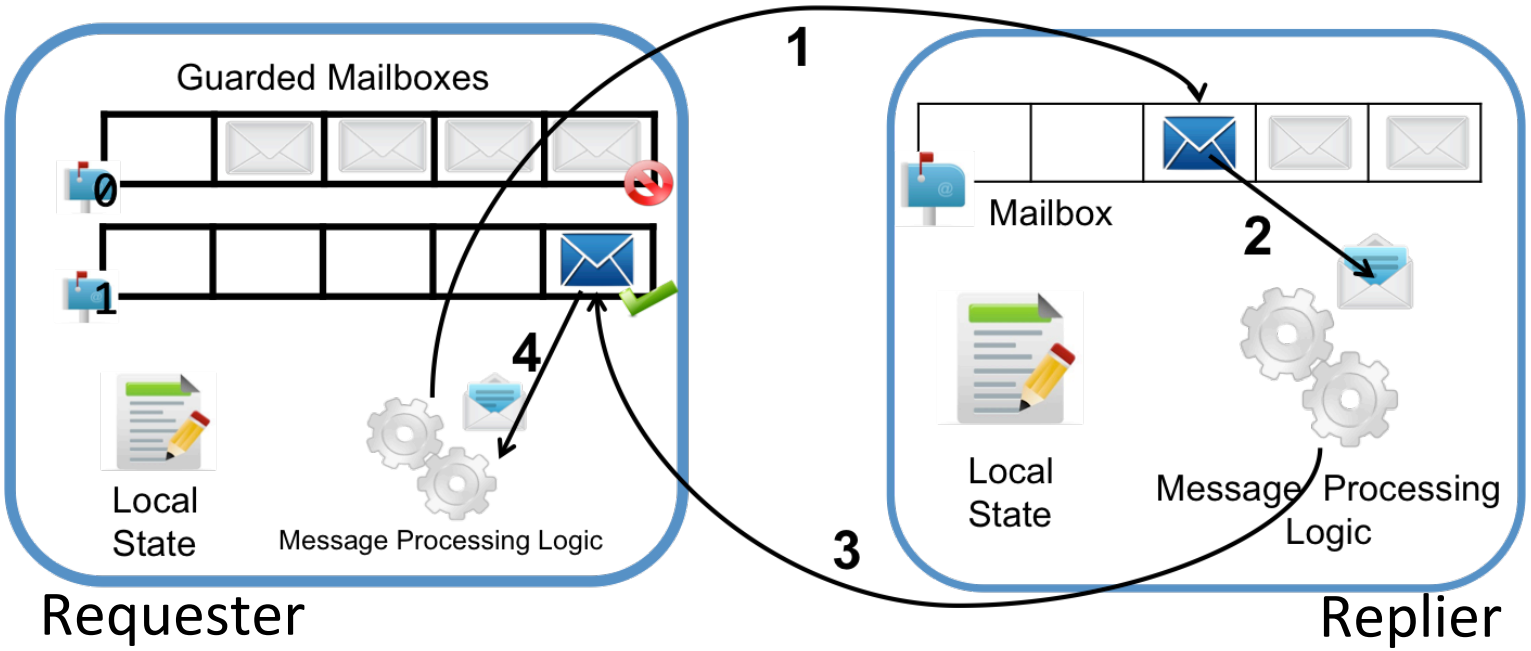
Synchronous Request-Response Pattern

1. Requester sends a message to a replier system
2. Replier *eventually* receives and processes the request
3. Replier returns a message in response
4. Requester can make further progress after receiving response



Synchronous Request/Response: Selector-based Solution

- Two mailboxes
  - one to receive regular messages
  - one to receives synchronous response messages
- Whenever expecting a synchronous response
  - disables the regular mailbox
  - ensures next message processed is from reply mailbox



Experimental Results

- 

12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node
- 
Java Hotspot JDK 1.8.0
- Our implementation:
  - 
Selector (HS), pure library implementation on Java 8
- Other libraries:
  - 
Actors (HA) 0.1.2
  - 

2.11.0 actors (SC)
  - 

2.3.2 (AK)
  - 

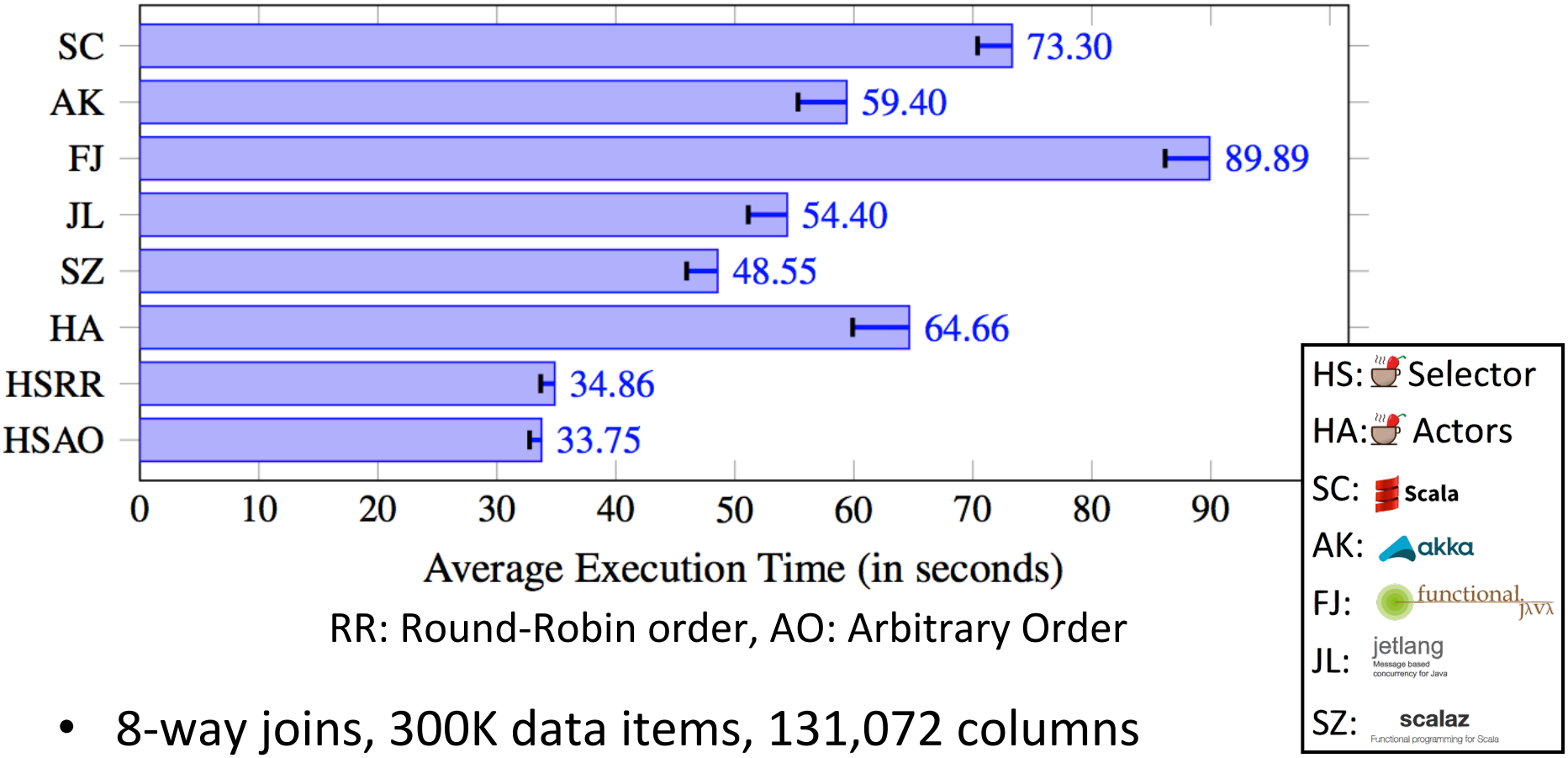
4.1 (FJ)
  - 

0.2.12 (JL)
  - 

7.1.0-M6 (SZ)

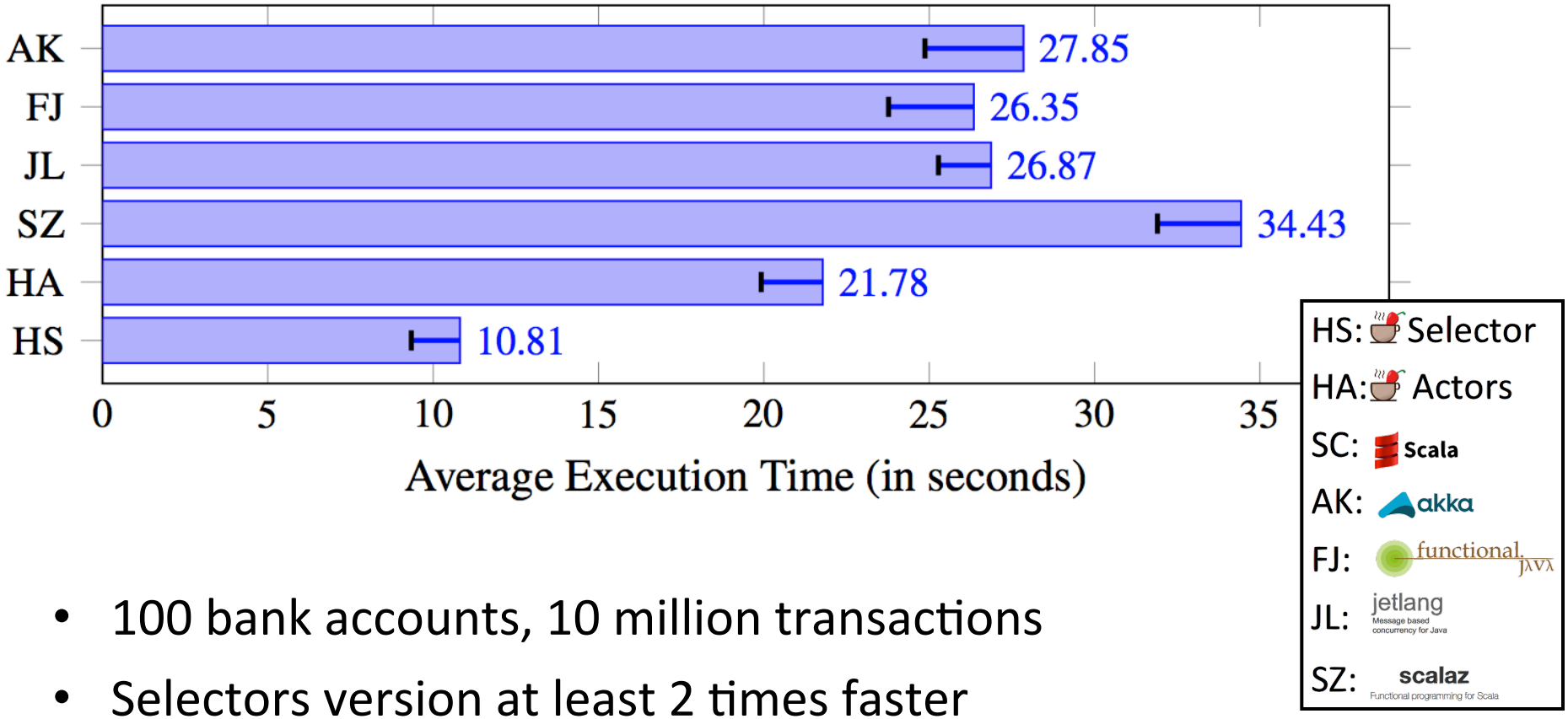
- Benchmarks from: *Savina – An Actor Benchmark Suite*. **Shams Imam**, Vivek Sarkar. 4th Int'l Workshop on Programming based on Actors, Agents, and Decentralized Control (**AGERE! 2014**), October 2014.

Filter Bank benchmark



- 8-way joins, 300K data items, 131,072 columns
- Selectors version at least 40% faster
  - Avoids local dictionary lookup overheads

Bank Transaction benchmark



- 100 bank accounts, 10 million transactions
- Selectors version at least 2 times faster
  - Avoids message recycling overheads

Related Work for Selectors

- Pattern matching on receive
  - Enabled-sets by Tomlinson and Singh [OOPSLA '89]
  - Scala Actors by Haller [COORDINATION 2007]
- Aggregator Pattern from Akka
  - Does not match sender
- Message priorities
  - SALSA provides two-level priority [ACM SIGPLAN Notices, Dec 2001]
- Parallel Actor Monitors [Science of Comp. Prog., Feb 2014]
  - Solves the symmetric reader-writer problems
  - Does not support priorities, hence other variants

Selectors Contributions

- **Simplify** writing of **synchronization and coordination patterns**
  - Using a **simple extension** to Actors
  - Multiple **cooperatively guarded** mailboxes
- Effective solution to synchronization/coordination patterns
  - Join patterns common in streaming applications
  - Synchronous request-reply
  - Variants of reader-writer concurrency
  - Priorities in message processing
  - Producer-consumer with bounded buffer

Selectors: Actors with Multiple Guarded Mailboxes. **Shams Imam**, Vivek Sarkar. 4th Int'l Workshop on Programming based on Actors, Agents, and Decentralized Control (**AGERE! 2014**), October 2014.

RICE

Thesis Summary

- **Cooperative techniques** help parallel programs
  - In presence of modern **synchronization and coordination constructs**
- **Cooperative Runtime**
  - Avoiding thread-blocking operations
  - Enable implementing future synchronization constructs
- Programming Model **Extension: Eureka**
  - Efficient collective cooperative task termination
  - Performance by reducing redundant work
  - Productivity by simplifying user-level code
- Programming Model **Extension**: **Selector**
  - Performance by reducing book-keeping performed by user
  - Productivity by simplifying user-level code

Acknowledgments

- Vivek Sarkar
- Other Committee members
- Habanero Research Group
- Colleagues at Rice CS Department

Questions

- Cooperative techniques help parallel programs
  - In presence of modern synchronization and coordination constructs
- Cooperative Runtime
  - Avoiding thread-blocking operations
  - Enable implementing future synchronization constructs
  - Efficient collective cooperative task termination
  - Performance by reducing redundant work
  - Productivity by simplifying user-level code
- Programming Model Extension: Selector
  - Performance by reducing book-keeping performed by user
  - Productivity by simplifying user-level code

```
import defense.audience.Questions;
import defense.audience.Comments;
```