JESSICA2:
A Distributed Java Virtual Machine
with Transparent Thread Migration Support

Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
{wzzhu+clwang+fcmlau}@csis.hku.hk

Outline

- Motivations
- Related works
- JESSICA2 features
- Experimental results
- Conclusion & Future works

Motivation

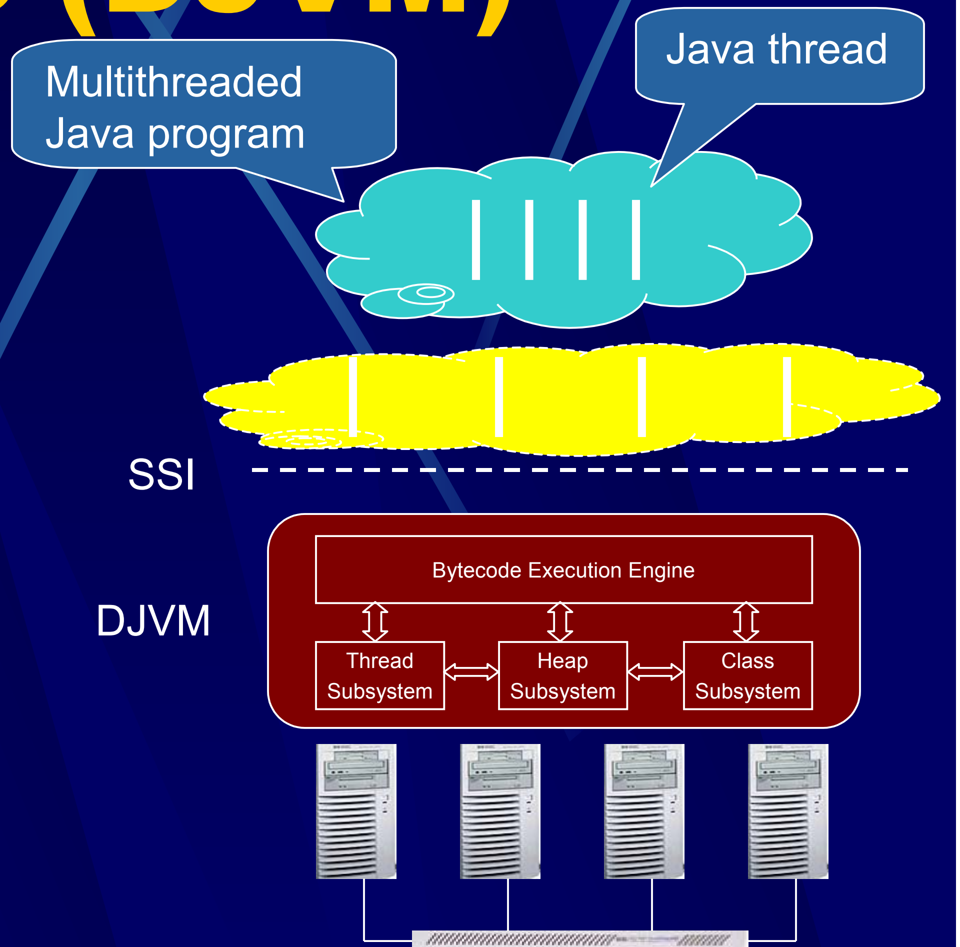
- Why Java?
  - The dominant language for server-side programming
  - Platform independent
  - Built-in multithreading support at language level
  - High-performance with Just-in-Time compilation
- Why cluster?
  - A cluster provides a scalable parallel hardware platform for high performance computing

Parallel/Distributed Computing using Java

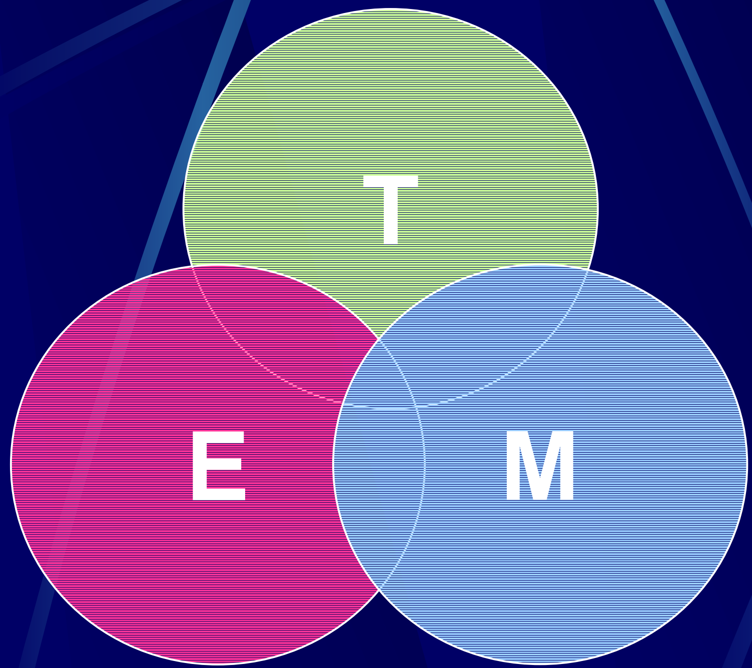
- RMI, Cobra ?
  - Application level
  - Complex programming model
  - Can't take advantage of Java's multithreading features
- Java Multithreading
  - Running a multithreaded Java application on a cluster
  - A Distributed Java Virtual Machine (DJVM) Approach

Distributed Java Virtual Machine (DJVM)

A **distributed Java Virtual Machine (DJVM)** spanning multiple cluster nodes can provide a true parallel execution environment for multithreaded Java applications with a Single System Image illusion to Java threads.



An abstract view of (Distributed) JVM



TEM Model

T: Thread System

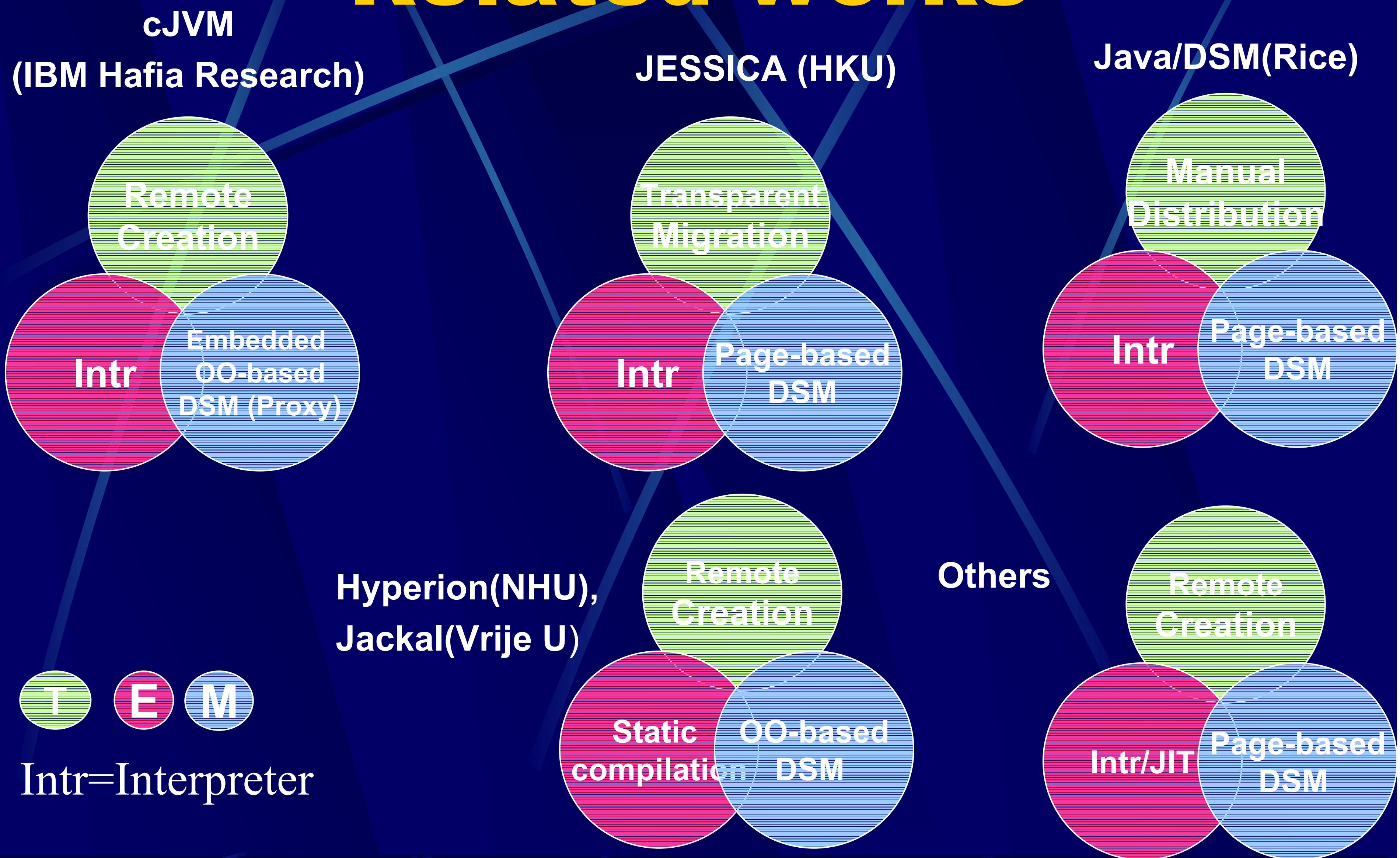
E: Execution Engine

M: Memory Space

Design issues of DJVM

- Extend TEM to distributed environment
  - **T** -> thread creation and migration mechanisms
  - **E** -> execution engine should be aware of the cluster environments
  - **M** -> provide a global object space in a distributed environment

Related works



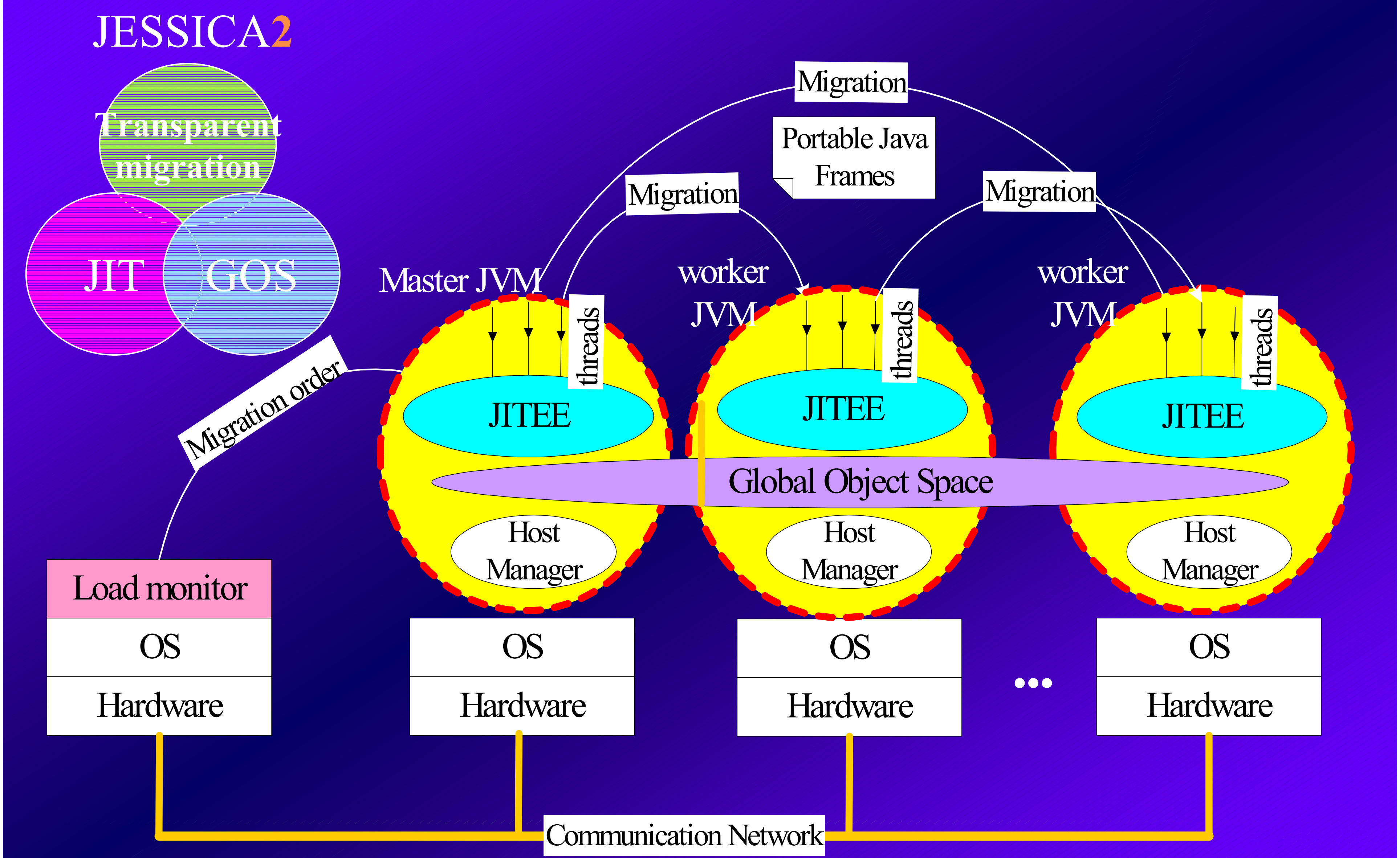
Problems in existing DJVM's

- Can't preserve Java's merits
  - Static compilation (Hyperion, Jackal)=> No dynamic class loading
  - Interpreters(cJVM,Java/DSM,JESSICA) => Can not support JIT compilation
  - Manual distribution (Java/DSM)=>Need to re-write programs
- Layered design using DSM can't be tightly coupled with JVM
  - JVM runtime information can't be channeled to DSM
  - False sharing problem if page-based DSM is employed

Our strategies

- Preemptive transparent Java thread migration in JIT mode
  - No source code modification or bytecode instrumenting
  - Runtime Capturing and Restoring of thread execution context at bytecode boundary
  - Able to be executed in JIT compilation mode
  - Enable dynamic load balancing on clusters
- Embedded Global Object Space layer
  - Take advantage of JVM runtime supports to reduce object access overheads

JESSICA2 Architecture



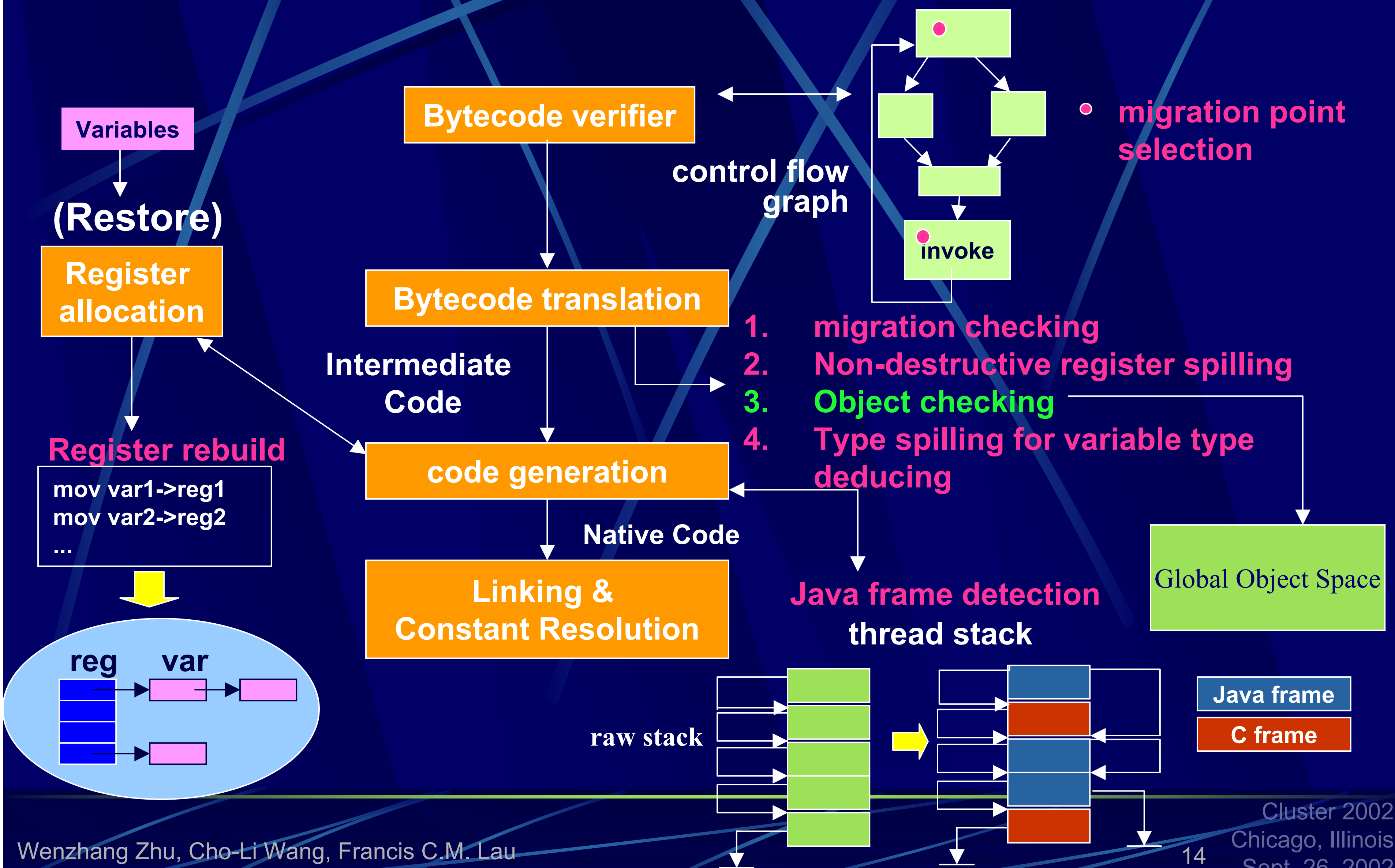
Transparent thread migration in JIT mode?

- Simple for interpreters (e.g. JESSICA)
  - Interpreter sits in the bytecode decoding loop which can be stopped upon a migration flag checking
  - The full state of a thread is available in the data structure of interpreter
  - No register allocation
- JIT mode execution makes things complex (JESSICA2)
  - No clear bytecode boundary
  - How to deal with machine registers?
  - How to organize the stack frames?
  - How to restore an execution of native codes?

What are those functions?

- **Migration Points Selection**
  - At the head of loop basic block + method
- **Register Context Handler**
  - **Nondestructive register spilling**: spill dirty registers at migration point without invalidation so that native codes can continue the use of registers
  - **Register rebuild**: use register recovering stub at restoring phase
- **Variable Type Deducing**
  - Spill type in stacks using compression
- **Java Frames Detection**
  - Discover consecutive Java frames

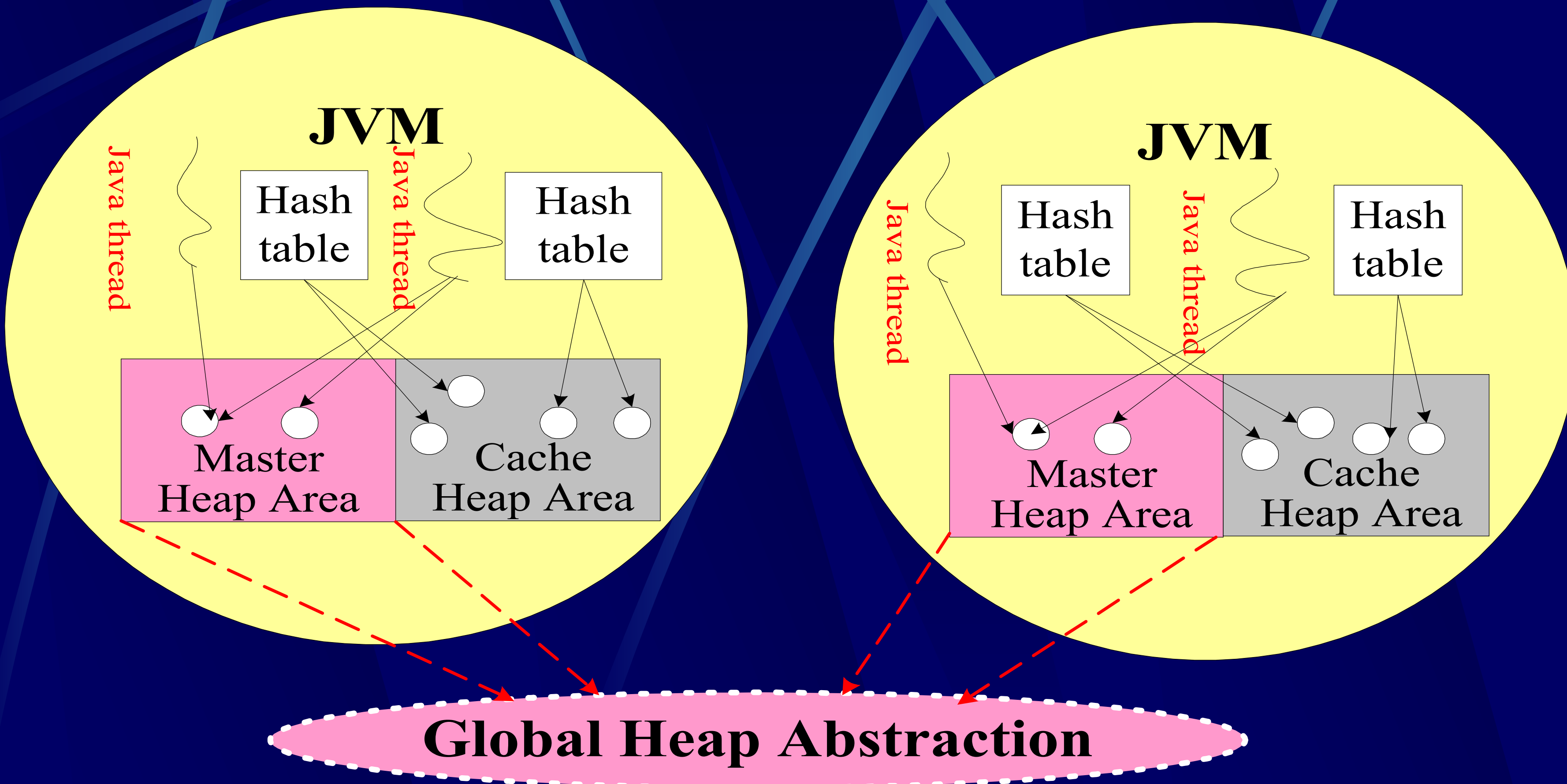
Details of Transparent Java thread migration inside JIT compiler



Global Object Space (GOS)

- Provide global heap abstraction for DJVM
- Home-based object coherence protocol, compliant with JVM Memory Model
  - OO-based to reduce false sharing
- Non-blocking communication
  - Use threaded I/O interface inside JVM for communication to hide the latency
- Adaptive object home migration mechanism
  - Take advantage of JVM runtime information for optimization

Overview of GOS



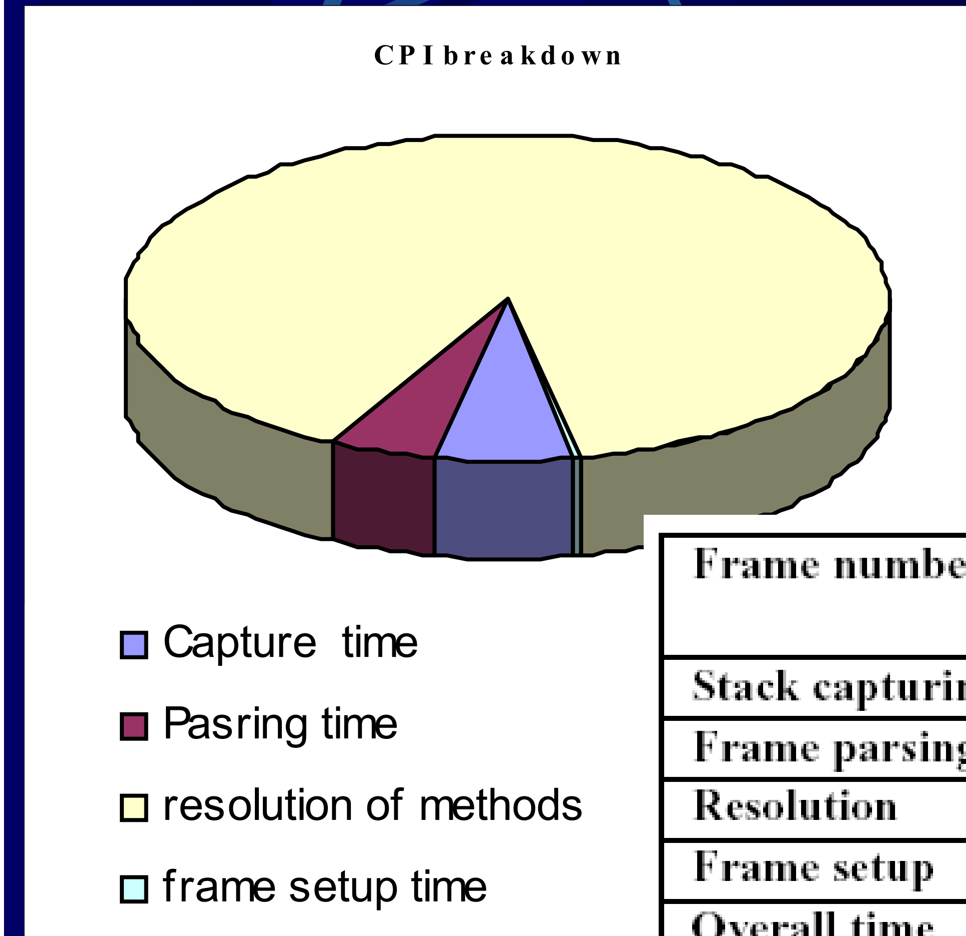
Adaptive object home migration

- Definition
  - "home" of an object = the JVM that holds the master copy of an object
- Problems
  - cache objects need to be flushed and re-fetched from the home whenever synchronization happens
- Adaptive object home migration
  - if # of accesses from a thread dominates the total # of accesses to an object, the object home will be migrated to the node where the thread is running

Experimental Setting

- Pentium II 540MHz, 128MB
- Linux 2.2.1 kernel
- Connected by Fast Ethernet
- Kaffe 1.0.6

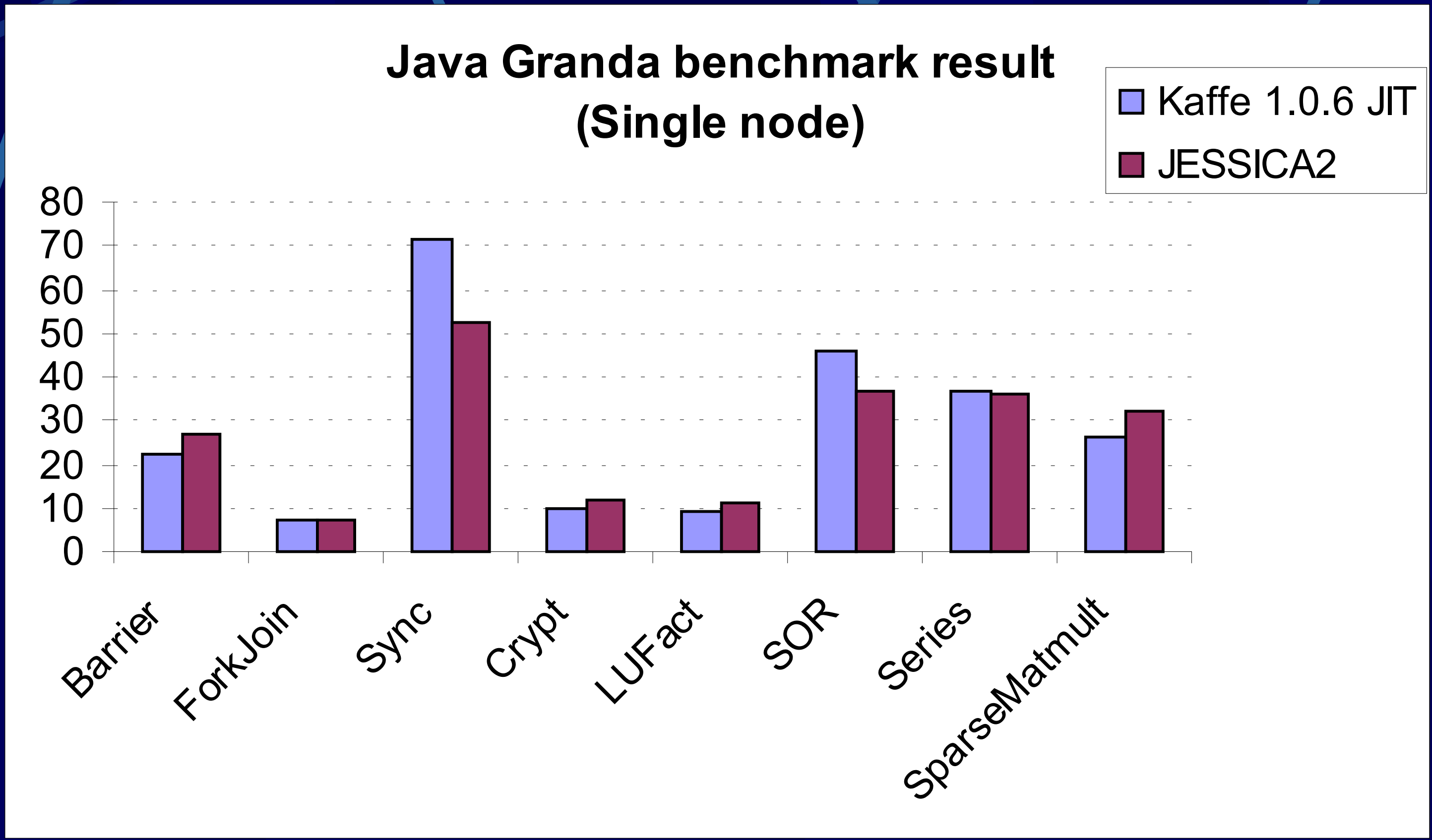
Microbenchmarks(I)



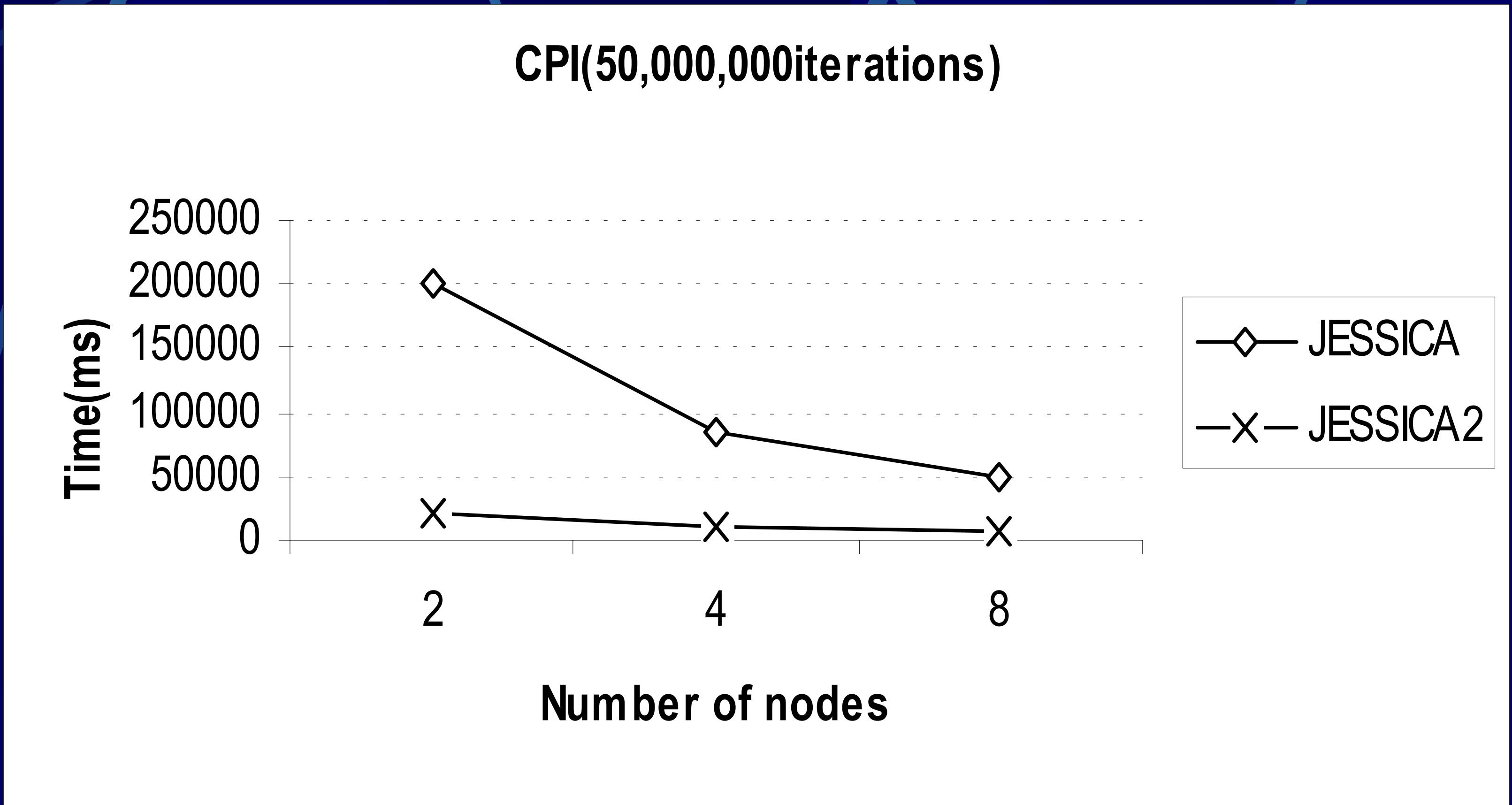
Frame number	1 frame (475Bytes)	2 frames (482Bytes)	11 frames (3,049Bytes)
Stack capturing	232	437	12,993
Frame parsing	166	328	1,383
Resolution	3,431	13,747	227,587
Frame setup	9	13	49
Overall time	3,838	14,525	242,012

Microbenchmark(II)

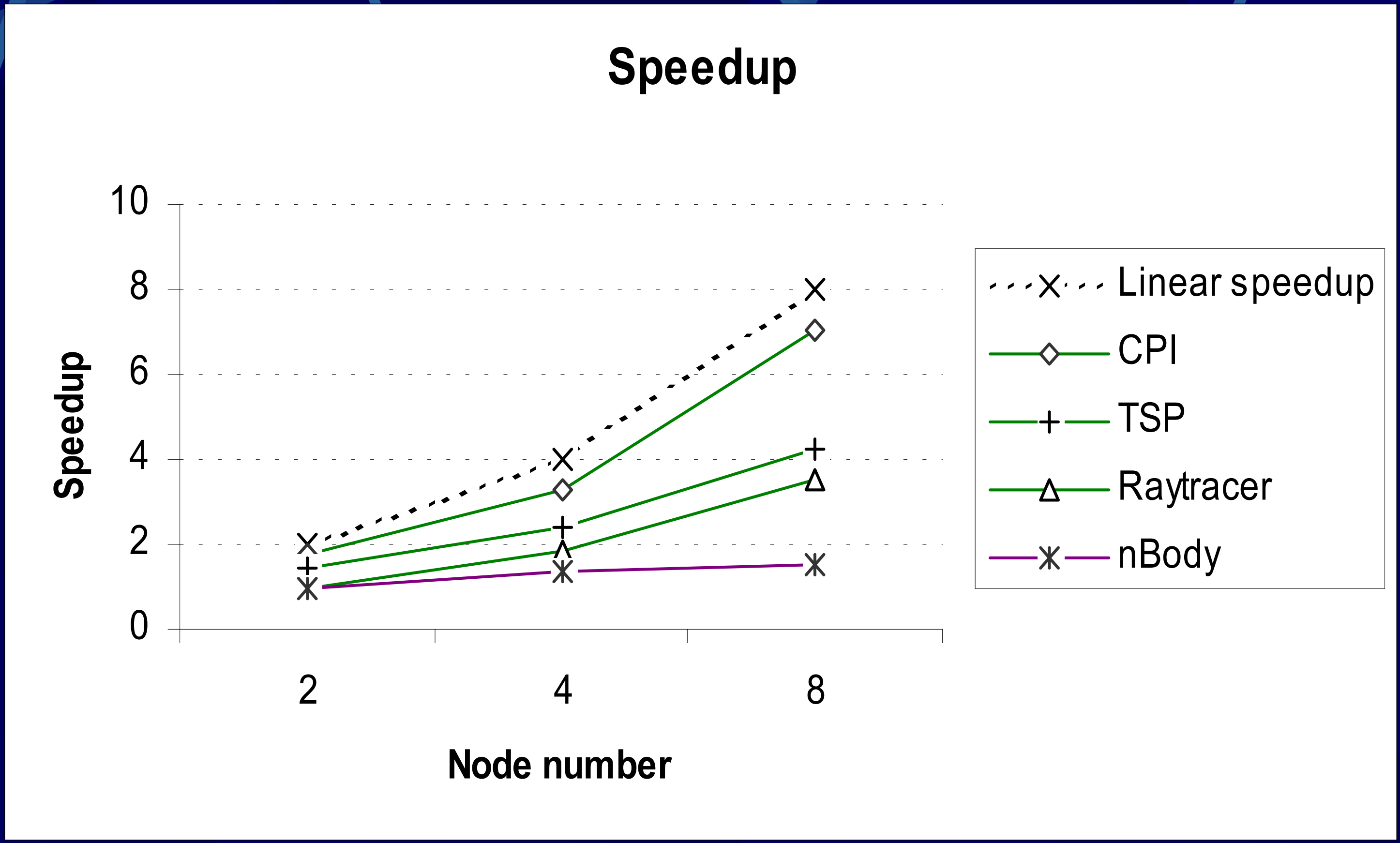
(Execution time in microseconds)



JESSICA2 vs JESSICA (CPI)

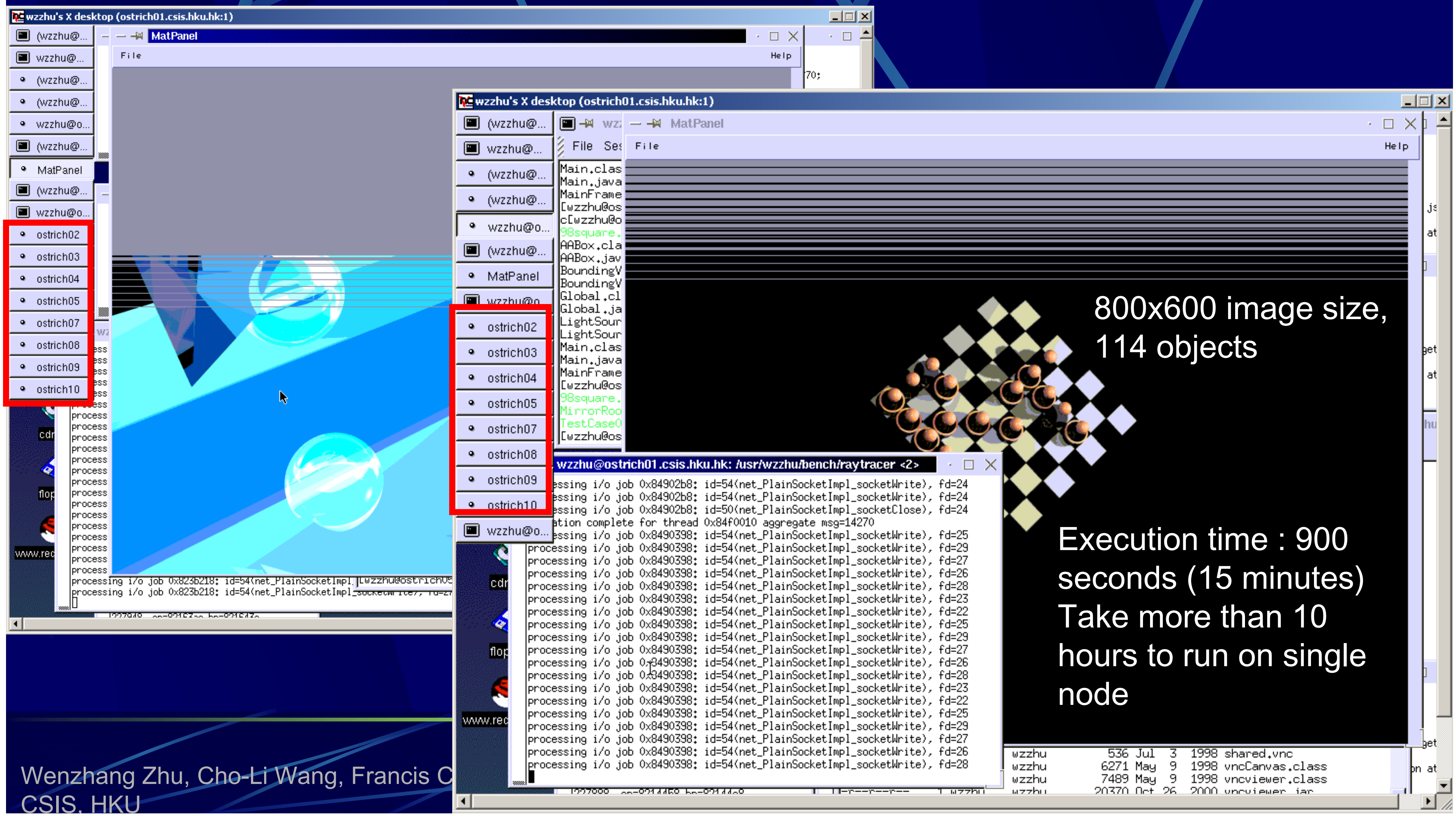


Application benchmark

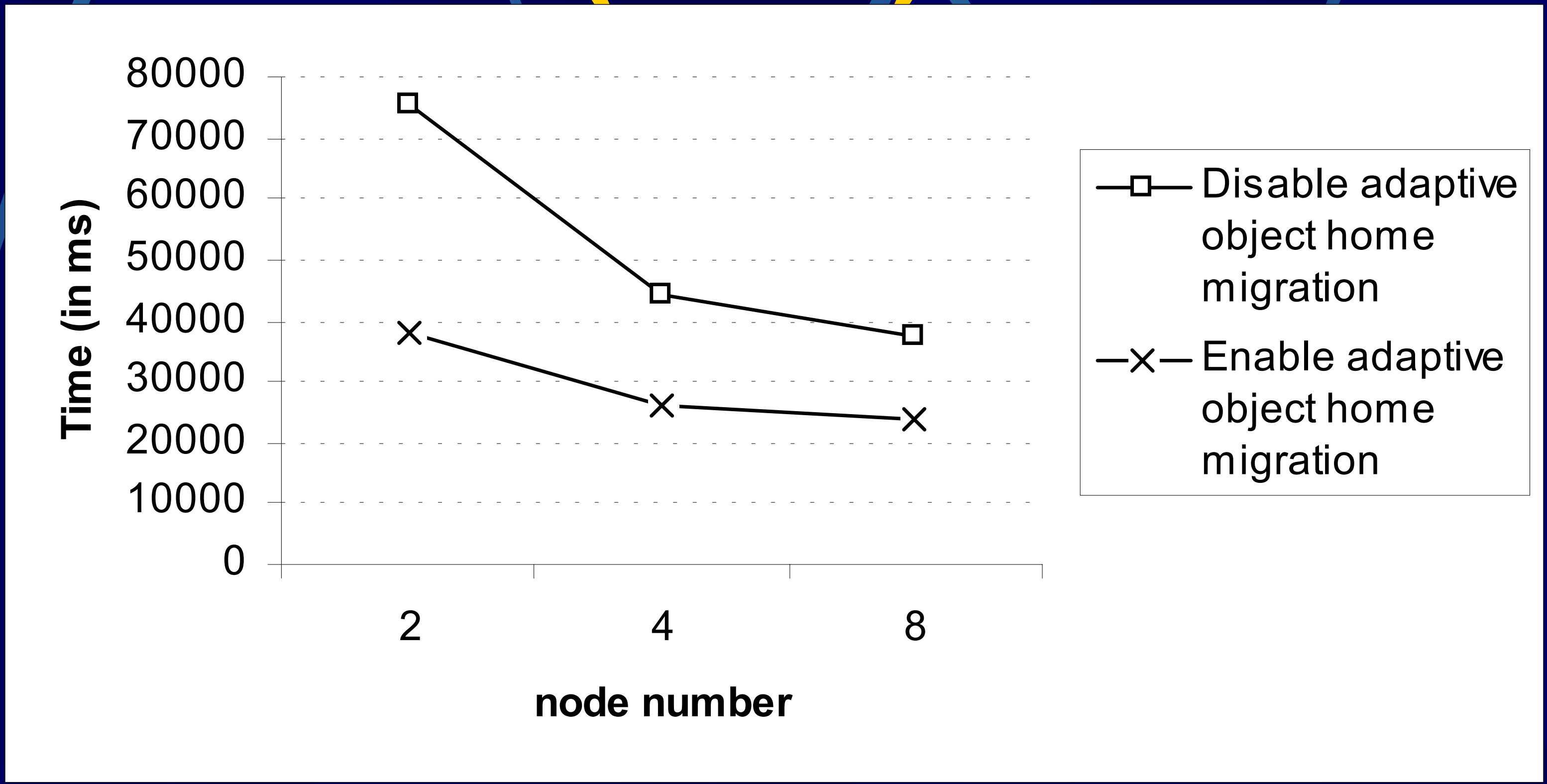


Parallel Ray Tracing on JESSICA2

(Running at 8-node P-III cluster)



Effect of Adaptive object home migration (SOR)



Conclusions

- Transparent Java thread migration in JIT compiler enables the high-performance execution of multithreaded Java application on clusters
- An embedded GOS layer can take advantage of the JVM runtime information to reduce communication overhead

Works in Progress

- Exploit new optimization techniques on GOS
- Incremental Distributed GC
- Add load balancing module
- Enhanced Single I/O Space to benefit more real-life applications
- Parallel I/O Support

Thanks

- Q & A