Adaptive Sampling-Based Profiling Techniques for Optimizing the Distributed JVM Runtime

King Tin Lam, Yang Luo, Cho-Li Wang

Speaker: King Tin Lam
Date: Apr 20, 2010

Systems Research Group
Department of Computer Science
The University of Hong Kong

IPDPS'10, Atlanta, Georgia, USA

Outline

1 Background
2 Challenges and Problems
3 Adaptive Object Sampling
4 Adaptive Stack Sampling
5 Performance Evaluation

Parallel Programming Paradigms

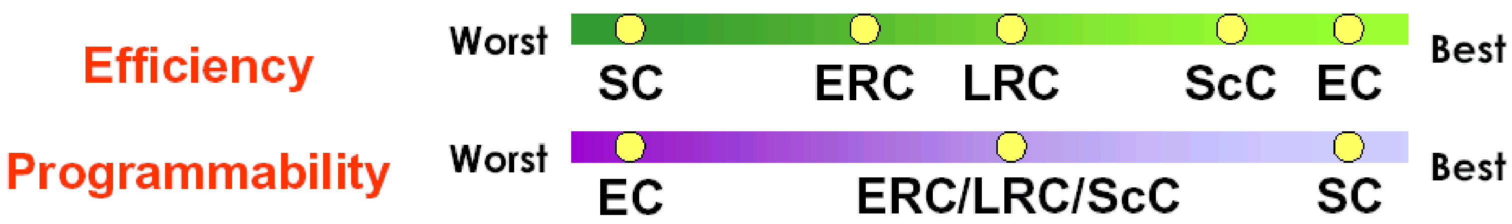
- For a single computer (multiprocessor, multicore),
  - ***Shared memory***
    - e.g. OpenMP
    - Much easier
- For a multicomputer (distributed-memory system),
  - ***Message passing***
    - e.g. MPI, PVM
    - Hard to programmers
  - ***Shared virtual memory (SVM)***
    - a.k.a. Software DSM
    - e.g. Treadmarks, CVM, JiaJia
    - Bind to a memory consistency model
    - Resemble ease of shared memory
    - Less efficient

Parallel Programming Paradigms

System	Developer	Implementation Level	Granularity	Consistency Model
IVY	Yale	Library + OS	Page (1KB)	SC
Munin	Rice	Library + OS	Variable	ERC
TreadMarks	Rice	Library	Page (4KB)	LRC
CVM	Maryland	Library	Page	LRC, SC
Midway	CMU	Library + Compiler	Variable	EC, PC, RC
NCP2	UFRJ, Brail	Library + Hardware support	Page (4KB)	EC, RC
Quarks	Utah	Library	Region, Page	RC, SC
softFLASH	Stanford	OS	Page (16KB)	RC, DIRC
Cashmere-2L	Rochester	Library	Page (8KB)	HLRC
Brazos	Rice	Library	Page	ScC
Shasta	DEC WRL	Compiler	Variable	SC
Mermaid	Toronto	Library+OS	Page (1KB, 8KB)	SC
Mirage	UCLA	OS	512Bytes	SC
JIAJIA	CAS, China	Library	Page (4KB)	ScC
Simple-COMA	SICS (Sweden) and SUN	OS	Page	SC
Blizzard-S	Wisconsin	Library	Cache line	SC
Shrimp	Princeton	OS+Hardware support	Page	AURC, SC
Linda	Yale	Language	Variable	SC
Orca	Vrije Univ., Netherlands	Language	Variable	EC-like

Parallel Programming Paradigms

- Memory consistency models
  - Strict Consistency
  - Sequential Consistency (SC)
  - Release consistency (RC)
    - Eager Release Consistency (ERC)
    - Lazy Release Consistency (LRC)
  - Scope Consistency (ScC)
  - Entry Consistency (EC)



    - Bind to a memory consistency model
    - Resemble ease of shared memory
    - Less efficient

Parallel Programming Paradigms

- Remote memory access is the scalability killer!
- Remote >> local latency (assume in 50-60ns)
  - Infiniband cluster (1-2μs): 20 x slower!
  - Ethernet cluster (100μs): 2,000 x slower!!
  - Grid/Internet (av. 500ms): 10,000,000 x slower!!!

- **"To speed up" ≈ "Reduce as much remote access as possible"**
- **The key is to improve locality**

  - e.g. Treadmarks, CVM, JiaJia
  - Bind to a memory consistency model
  - Resemble ease of shared memory
  - Less efficient

The PGAS Model

- User hints
  - Add annotation
  - Use special API constructs for locality hint inputs (e.g. X10’s *places*)
- *PGAS (Partitioned Global Address Space)*
  - "Hybrid" parallel paradigm
  - Essentially Distributed Shared Memory (DSM)
  - But corporate some MPI-like constructs
  - Research languages:
    UPC, Co-Array Fortran (CAF), Titanium
  - HPCS Languages:
    X10 (IBM), Chapel (Cray)
- A burden to programmers

Our Dream Model: PGPGAS or $(PG)^2AS$

- ***Profile-Guided PGAS (PG^2AS)***
  - A built-in **runtime** profiler instead of humans for digging out the locality hints
- Profile-guided adaptive locality management
  - Thread migration
  - Object home migration
  - Object prefetching

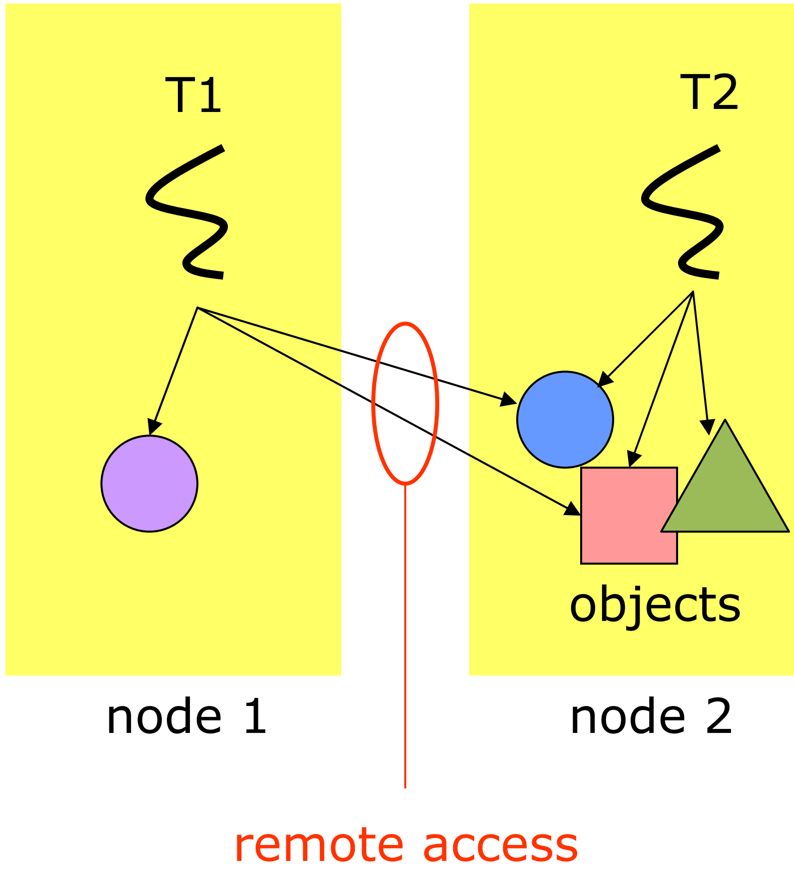
Something new in this paper

- API-free shared virtual memory
  - Transparent clustering and scaling
    - Automatic thread distribution
    - Location-transparent access
  - System instruments cluster-wide logics
  - No modification to existing applications

Previous distributed JVM research (e.g. cJVM, JavaSplit, JESSICA, …)

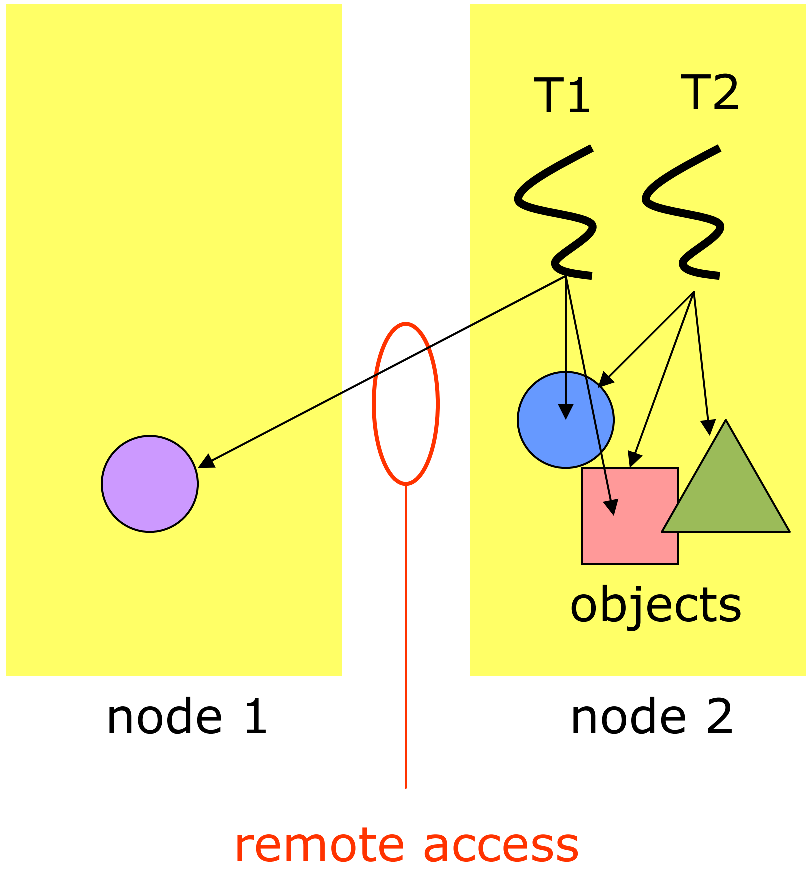
Techniques to improve locality

- Runtime techniques
  - Migration
    - Thread
    - Object (Home)
  - Prefetching
    - Spatial
    - Temporal



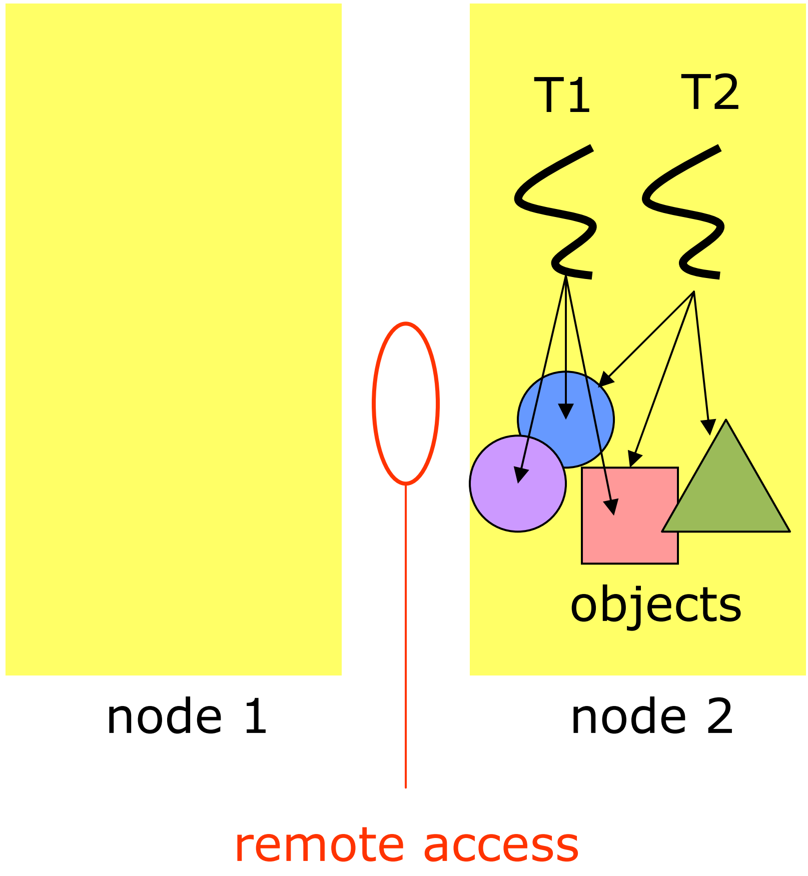
Techniques to improve locality

- Runtime techniques
  - Migration
    - Thread
    - Object (Home)
  - Prefetching
    - Spatial
    - Temporal



Techniques to improve locality

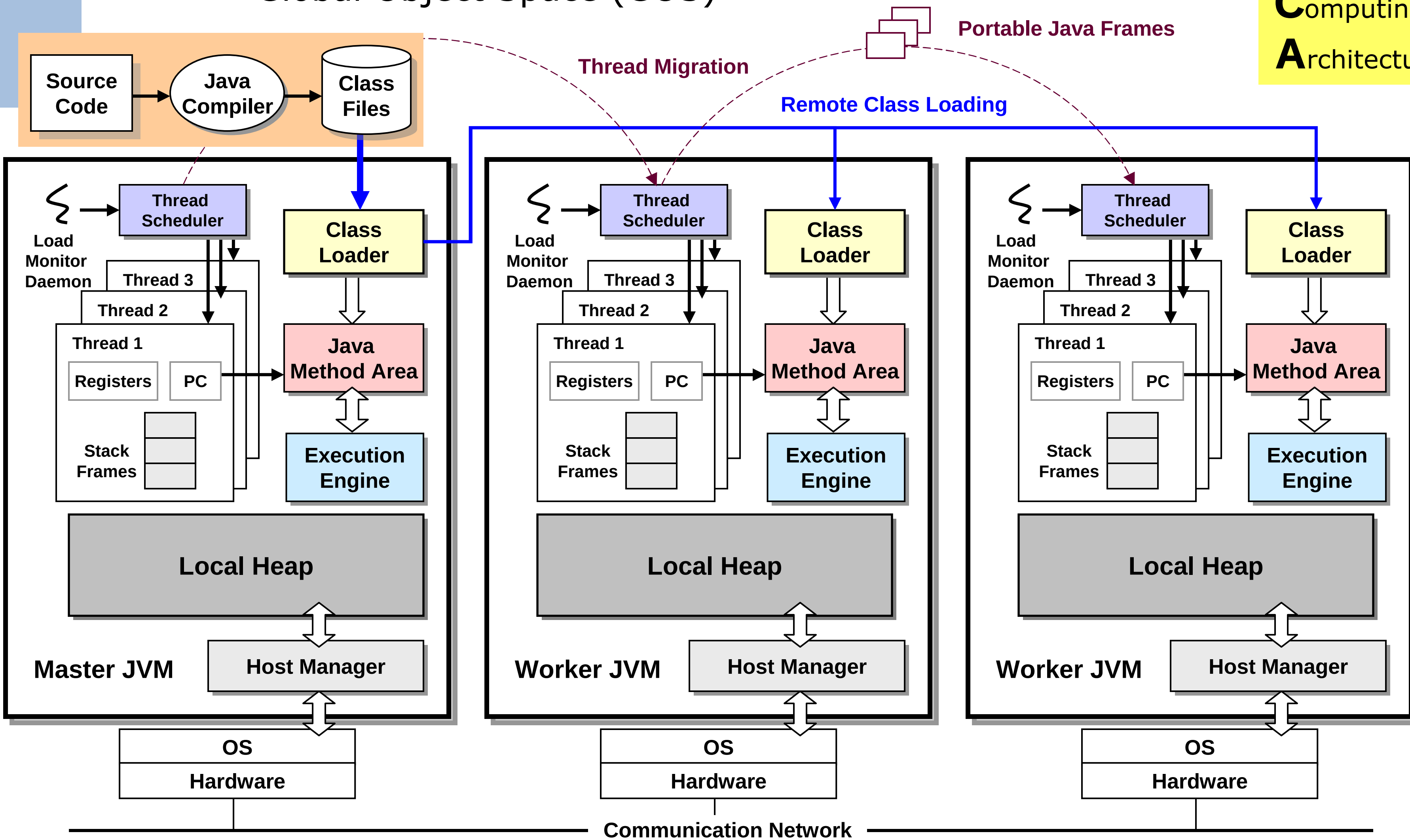
- Runtime techniques
  - Migration
    - Thread
    - Object (Home)
  - Prefetching
    - Spatial
    - Temporal



JESSICA Distributed Java VM

- A cluster-wide JVM with
  - Dynamic thread mobility in JIT mode
  - Global Object Space (GOS)

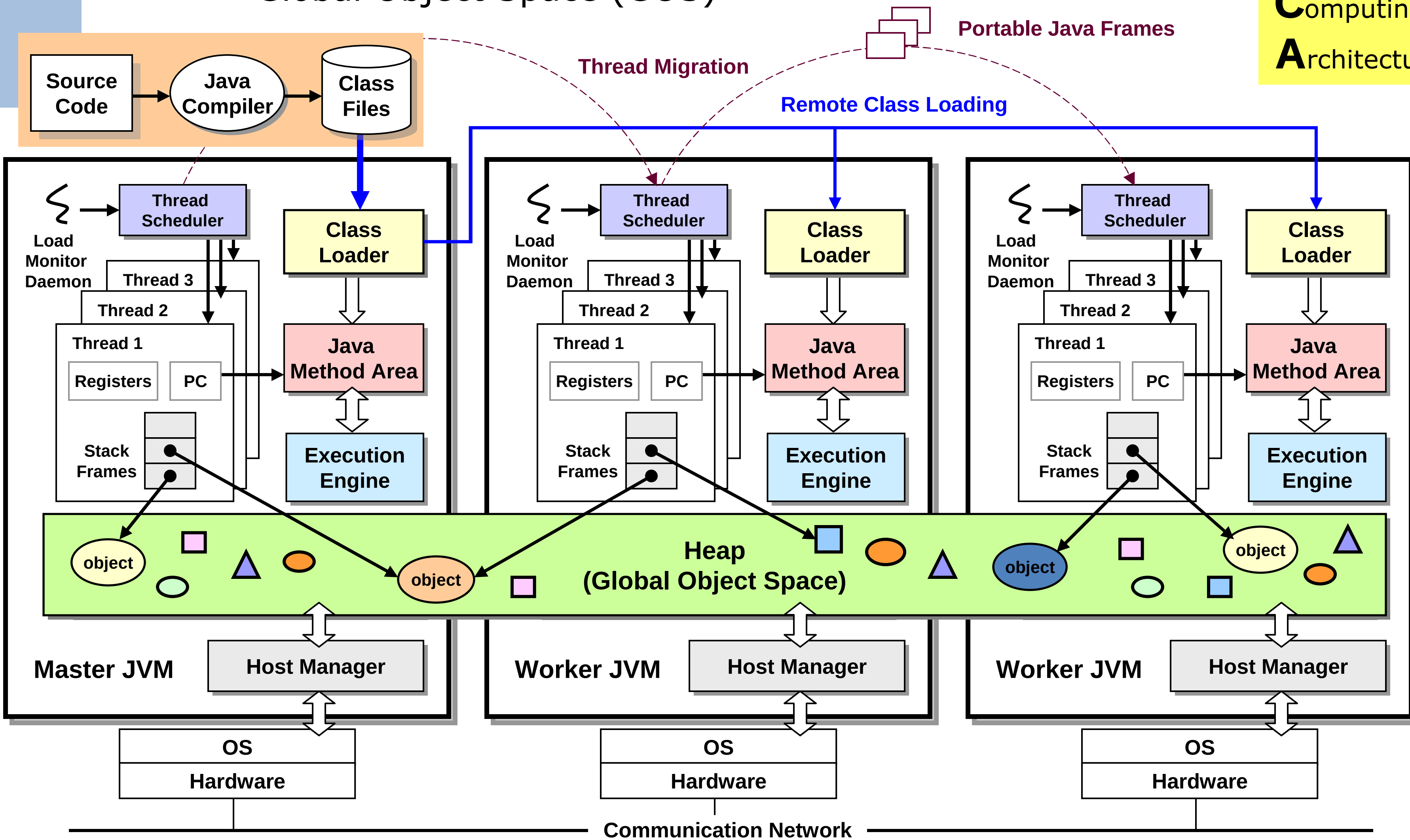
Java
Enabled
Single
System
I mage
Computing
Architecture



JESSICA Distributed Java VM

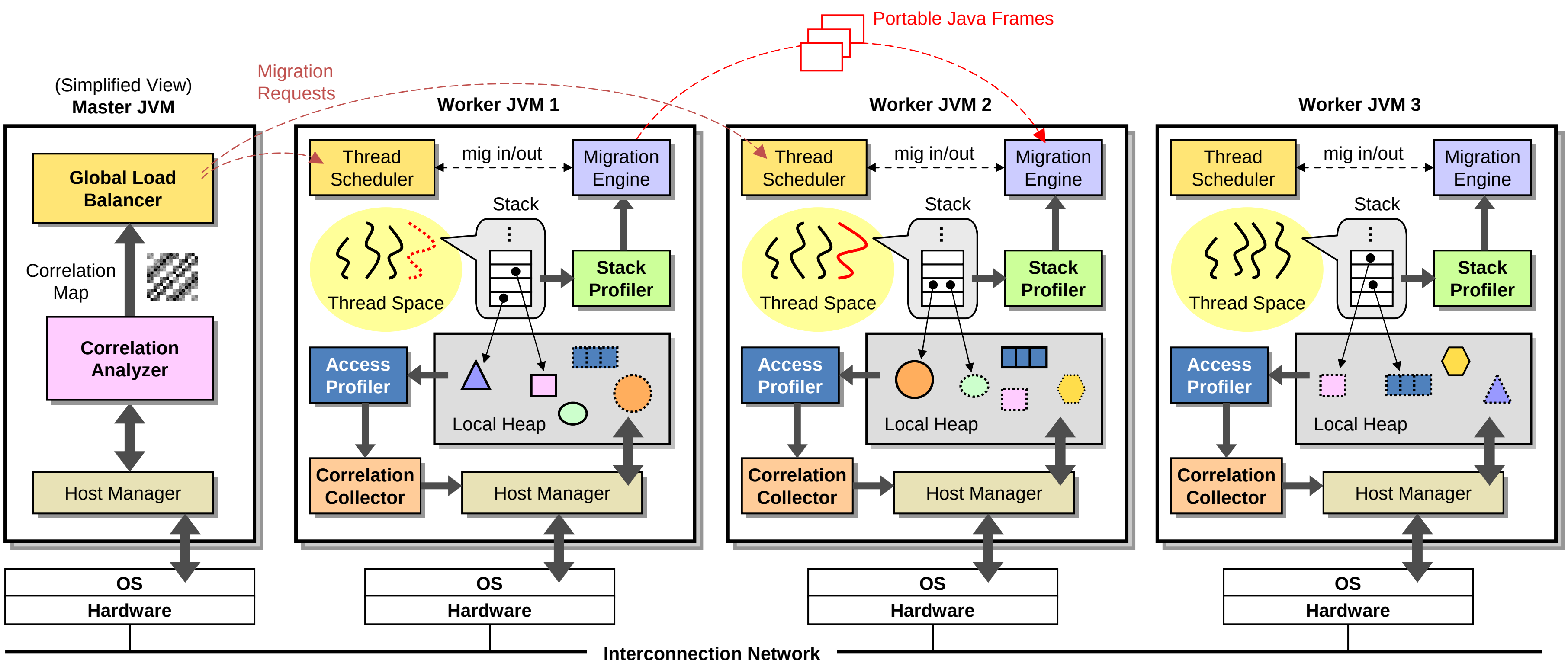
Java
Enabled
Single
System
I mage
Computing
Architecture

- A cluster-wide JVM with
  - Dynamic thread mobility in JIT mode
  - Global Object Space (GOS)



PG-JESSICA: Profile-Guided Version

- Now equipped with
  - ***Access profiler***: track object access over heap to deduce inter-thread sharing -> *thread-thread relation*
  - ***Stack profiler***: track the set of frequent objects accessed by each thread -> *thread migration cost*
  - ***Correlation analyzer***: profile-guided decisions on dynamic thread migration -> global locality improvement



Outline

1. Background
2. **Challenges and Problems**
3. Adaptive Object Sampling
4. Adaptive Stack Sampling
5. Performance Evaluation

Challenge 1

- How does the runtime know which threads to migrate can make the most locality benefit?
- Difficult to decide if no global inter-thread sharing information
- Solution: Track sharing % threads
  - T1 accesses O1, O3, O5, …
  - T2 accesses O1, O2, O3, …
  - Sharing % T1 & T2: O1, O3

Thread Correlation Map (TCM)

- Thitikamol and Keleher; D-CVM (1999)
  - Proposed "Active Correlation Tracking"
- Visualize correlation % threads by a 2D map
  - Grayscale(x,y) = sharing amount of thread x and y
  - TCM(1,1) = TCM(2,2) = TCM(3,3) = … = 0

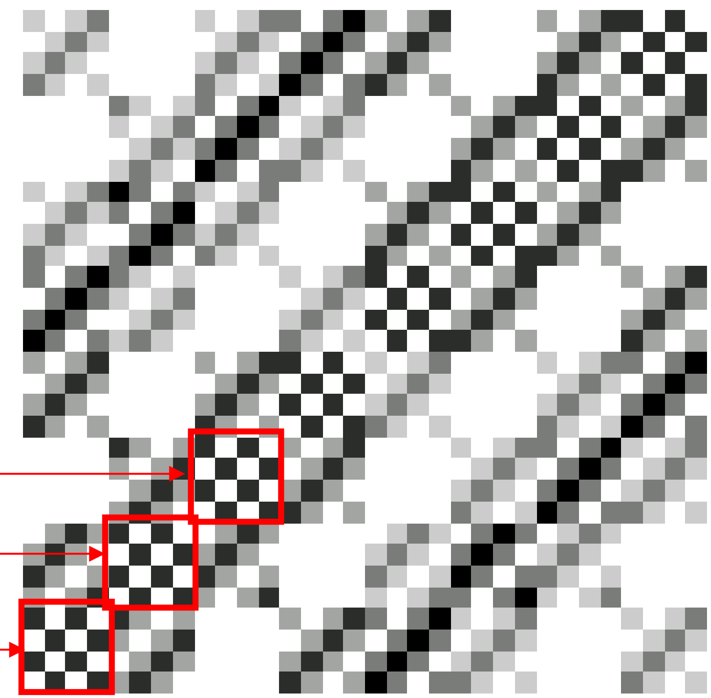
e.g. Water-Spatial
32 threads placed
on 8 nodes

⋮

node 3

node 2

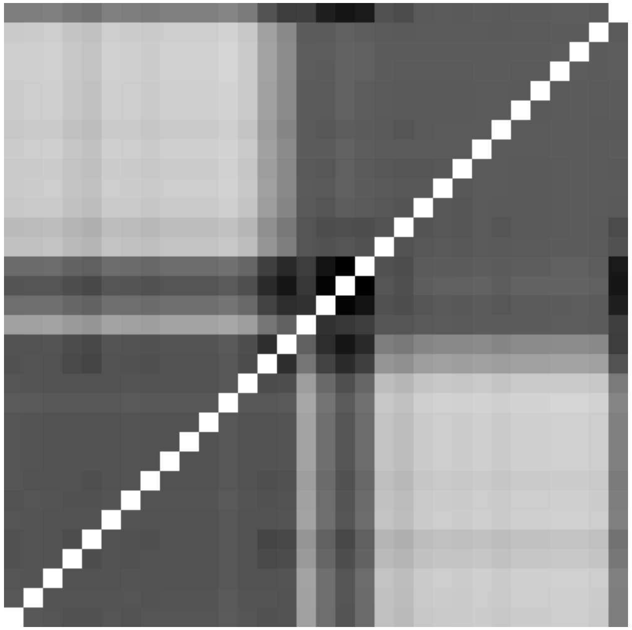
node 1



Problems for OO-Based Systems

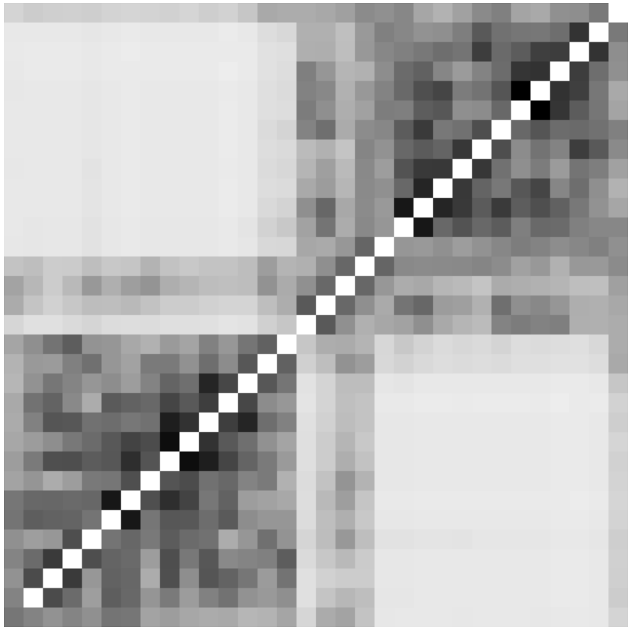
Simulation
Barnes-Hut: 32 threads, 4K bodies (<100 bytes each), dist=7.0

Page size: 4KB



- Low tracking overhead
- But suffer false sharing
- *Induced* sharing pattern
- Can't be used at all

Page size: 128 byte



- No or little false sharing
- *Inherent* sharing pattern
- But at much higher cost: 32 times more tracking

Challenge 2

- Thread migration cost is ill-modeled in past research.
  - Suppose thread *T* has *n* frames

$$t_{mig}(T) = \sum_{i=1}^{n}\left[t_{capture}(i) + t_{restore}(i)\right] + \alpha + \frac{\sum_{i=1}^{n} L_{frame}(i)}{\beta} \quad \dots (1)$$

network latency & bandwidth

- Did not consider ***indirect*** cost of subsequent object misses after migration → inaccurate decisions
- How about including cost of shipping the thread's working set?

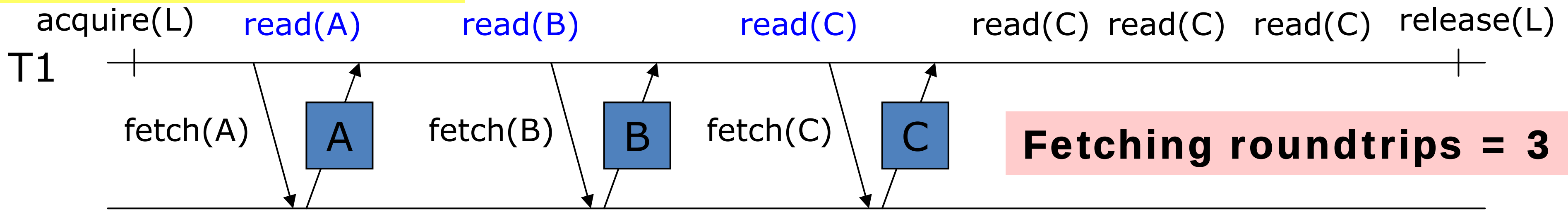
$$t_{mig}(T) = \sum_{i=1}^{n}\left[t_{capture}(i) + t_{restore}(i)\right] + \alpha + \frac{\sum_{i=1}^{n} L_{frame}(i) + W_T(t,\tau)}{\beta} \quad \dots (2)$$

- Yes! But not the best model for the migration cost

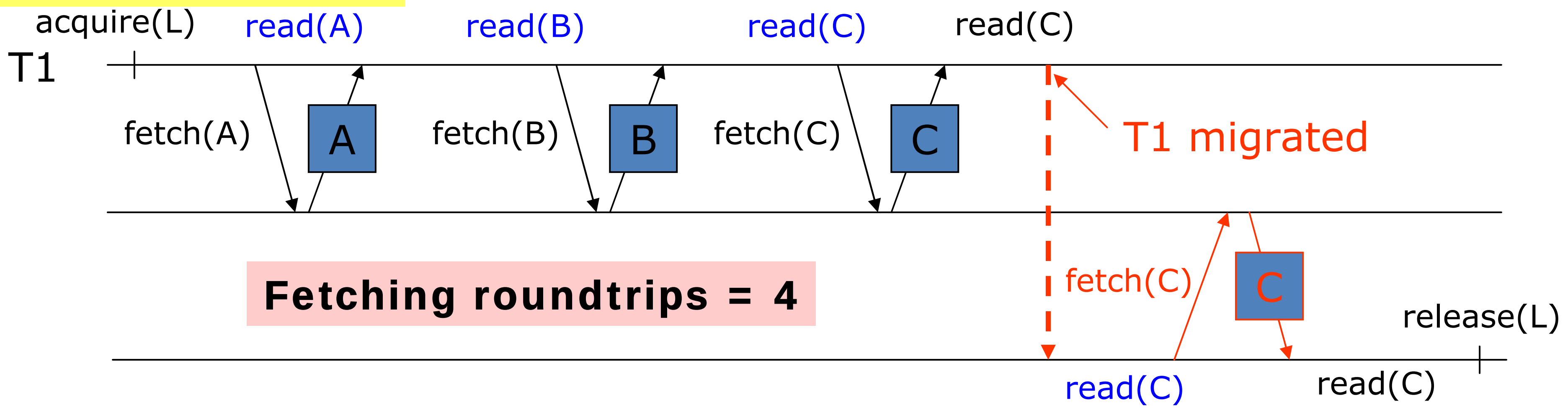
Challenge 2 (Cont')

- Suppose T1 accesses within the same interval:
  - A (1 time), B (1 time), C (4 times)
  - W_{T1}={A, B, C}

(1) Without migration:



(2) With migration:

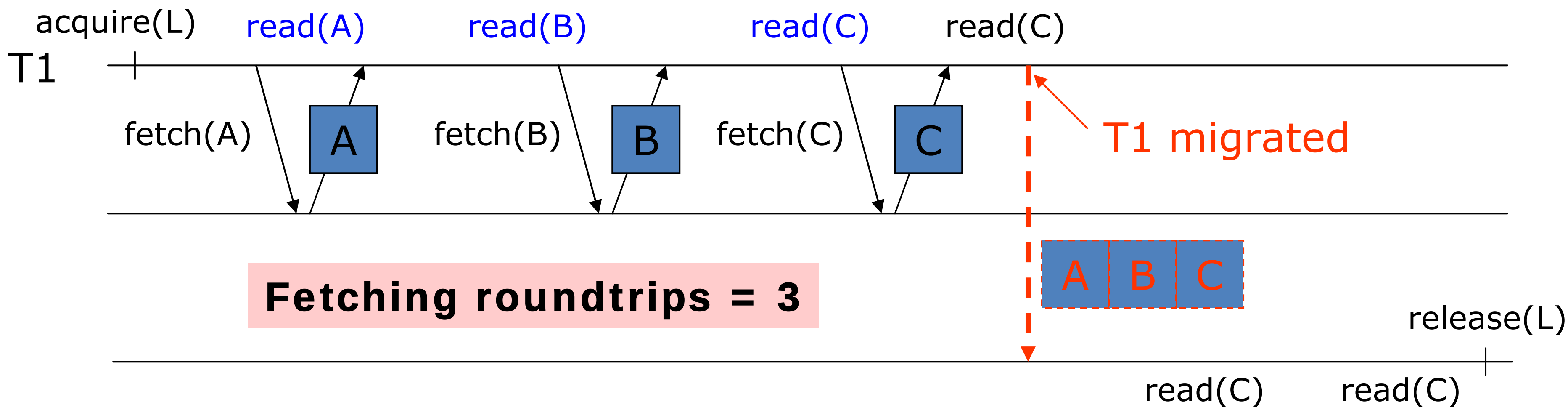


Challenge 2 (Cont')

(3) With migration prefetching W_{T1}:

W_{T1}={A, B, C}

A (1 time), B (1 time), C (4 times)



However, prefetching A and B are unnecessary overheads. We need prefetch of C only.
How can we know that?

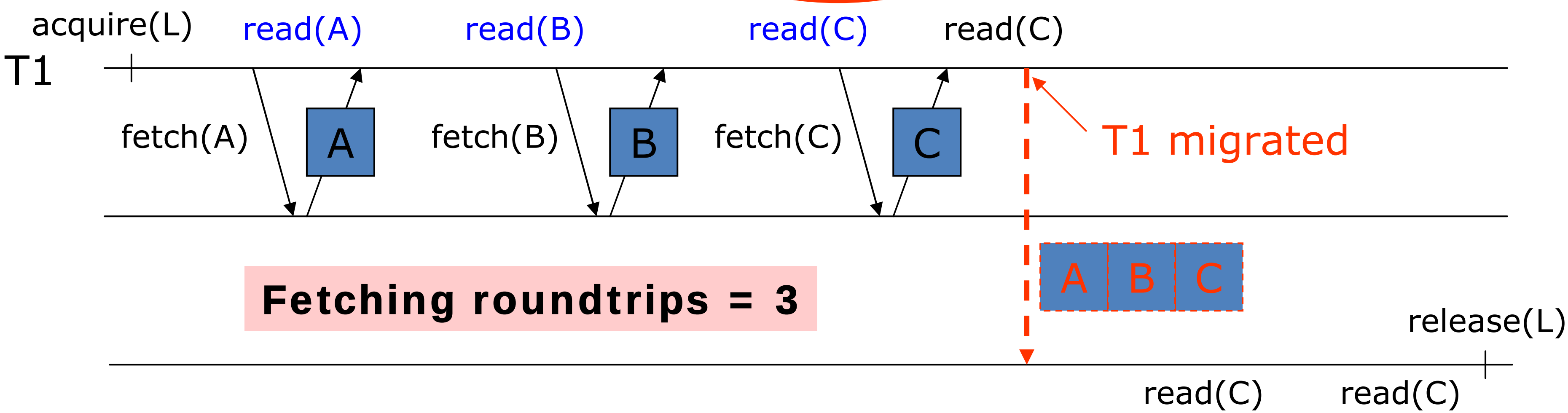
Challenge 2 (Cont’)

(3) With migration prefetching W_{T1}:

W_{T1}={A, B, C}

Track access frequency

A (1 time), B (1 time), C (4 times)



However, prefetching A and B are unnecessary overheads. We need prefetch of C only.
How can we know that?

Sticky Set

- We define the ***sticky set (SS)*** of a thread as a subset of working set that includes only those frequently used objects.
- "Sticky" in the sense that if the thread is migrated, this set of objects should be prefetched along to save most object misses to follow.
- Objects in SS are more likely to be fetched again after migration.
- Size of SS serves as a good estimate of indirect cost of thread migration.

How to Detect Sticky Set

- Compiler can only give qualitative answer
  - Pointer analysis, shape analysis, …
- Detecting SS at **runtime**
  - Our approach
  - Much more accurate
  - But tracking object access frequency is also costly
  - How to cut costs?

Summary of Our Solution

- What we want to do:
  1. Model thread sharing (inter-thread correlation)
  2. Model indirect thread migration cost
- Profiling results:
  1. Thread correlation map (TCM)
  2. Per-thread sticky set (SS)
- Use both to design new migration policy
  1. Correlation-driven
  2. Cost-aware
- How we profile them efficiently? (Our main contribution: lightweight techniques)
  1. **Adaptive object sampling** → TCM
  2. **Adaptive stack sampling** → SS

New Thread Migration Policy

- Correlation-Driven
  - TCM(T1, T2) > threshold → migrate T1 to T2 or T2 to T1
- Cost-aware
  - But T1 to T2 or T2 to T1?
    - Depends on which of SS(T1), SS(T2) is bigger?
    - Also need to compare with correlation with other local threads

Outline

Thread Correlation Tracking

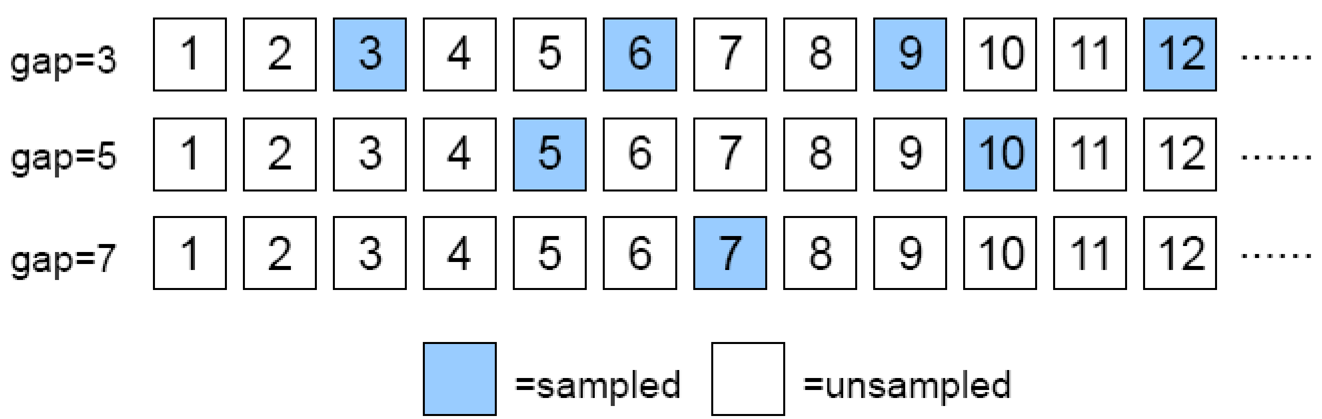
- Our mechanism is OO-based
- **OAL**: Object Access List
  - We need to obtain thread-object relation first.
- **TCM**: Thread Correlation Map
  - Collect OALs from all threads cluster-wide
  - Compute each element of TCM from OALs
- How to obtain OAL?
  - Passive: only when object checks see invalid object states (i.e. access faults)
  - **Active**:
    - Real object states are stored separately
    - Purposefully set object states to "falsely invalid" → trigger *correlation faults* → logging into OALs
    - Real states are restored after serving correlation faults; access faults are handled normally

Object Sampling

- CPU/comm. overhead of TCM/OAL can be substantial
  - Too many objects to track in a fine-grained app!
  - Can’t compute TCM in time as system scales up
- Need **object sampling** – i.e. only a portion of heap (selected objects) will undergo access tracking.
- But how much heap portion to sample?
  - Traditional (fixed rate):
    - Keep a global counter *k* of #bytes accessed over the heap
    - Each object header has a "sample" flag;
    - Upon an object creation, mark the flag whenever *k* > threshold

Adaptive Object Sampling (AOS)

- Each object has a "sequence number"
- Sample the object if sequence # is divisible by the current "sampling gap"
- Sampling gap can be selected and change at runtime
- Strike a balance of cost and accuracy
- Sampling rate definition
  - 1X = Sample 1 object per page of heap
  - 1024X means "full sampling"



Accuracy of AOS

- Because of sampling, we miss to track some objects in the heap.
- So we will see error.
- Let $A = [a_{ij}]_{N \times N}$ and $B = [b_{ij}]_{N \times N}$ be two TCMs and B is obtained by full sampling.
- A contains a % error defined by:

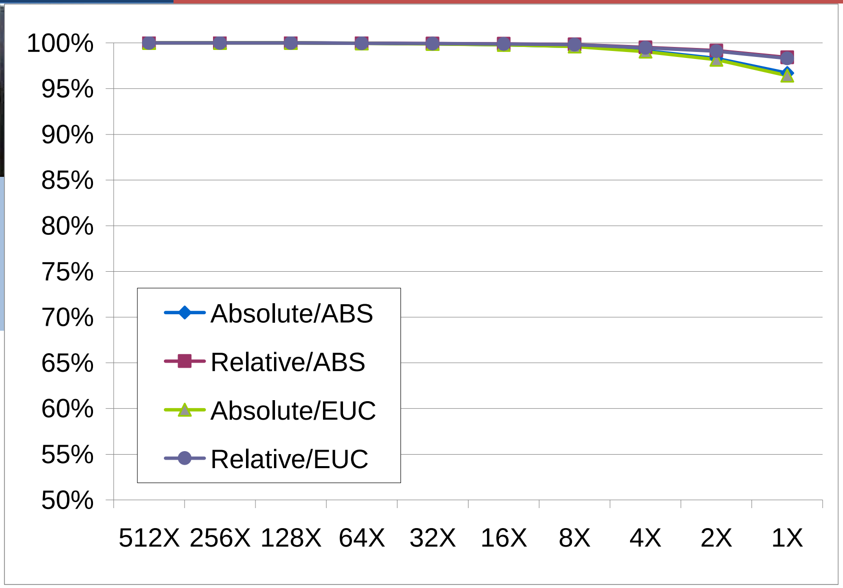
$$E_{EUC} = \frac{\sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} (a_{ij} - b_{ij})^2}}{\sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} (b_{ij})^2}}$$

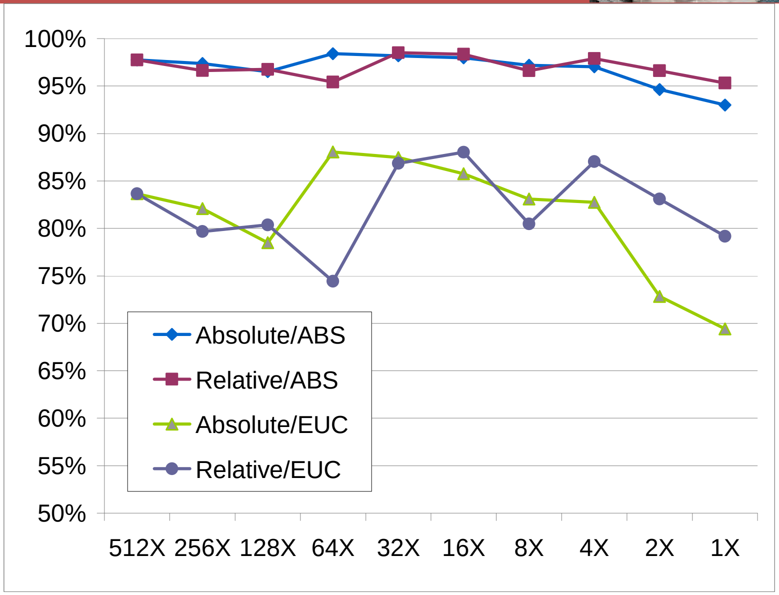
(Euclidean distance)

$$E_{ABS} = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} \left| a_{ij} - b_{ij} \right|}{\sum_{i=1}^{N} \sum_{j=1}^{N} \left| b_{ij} \right|}$$

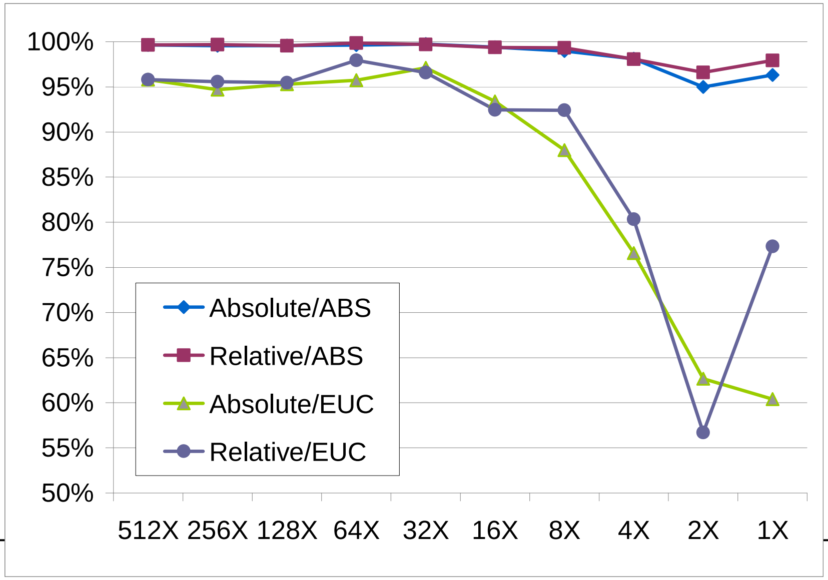
(Absolute distance)

Accuracy of AOS (Cont')





(a) SOR (b) Barnes-Hut (c) Water-Spatial



Outline

1 Background

2 Challenges and Problems

3 Adaptive Object Sampling

4 **Adaptive Stack Sampling**

5 Performance Evaluation

Tracking sticky sets

- Common belief is that we need to pay per-access overhead to maintain LRU/LFU/…, etc
- We use an elegant stack profiling approach: take and compare snapshots of stack states
  - no overhead for object access
  - background profiling is cheap and flexible

o1	o5	int	o1
o5	o2	int	o5
int	o1	int	int
o4	double	int	int
float	float	o2	o100
int	int	o1	o9
o2	o0	int	o10
stack	**stack**	**stack**	**stack**
t0	t0	t1	t1
p0	p1	p0	p1

Time: t0, t0, t1, t1
Processor: p0, p1, p0, p1

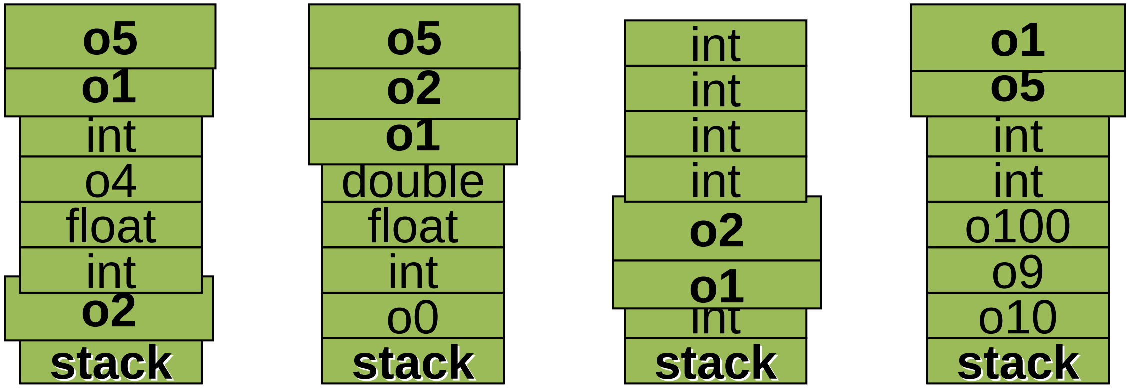
Tracking sticky sets

- Common belief is that we need to pay per-access overhead to maintain LRU/LFU/…, etc
- We use an elegant stack profiling approach: take and compare snapshots of stack states
  - no overhead for object access
  - background profiling is cheap and flexible

	o5	**o5**	int	**o1**
	o1	o2	int	**o5**
	int	**o1**	int	int
	o4	double	int	int
	float	float	o2	o100
	int	int	**o1**	o9
	o2	o0	int	o10
	stack	**stack**	**stack**	**stack**
Time:	t0	t0	t1	t1
Processor:	p0	p1	p0	p1

Tracking sticky sets

- Common belief is that we need to pay per-access overhead to maintain LRU/LFU/…, etc
- We use an elegant stack profiling approach: take and compare snapshots of stack states
  - no overhead for object access
  - background profiling is cheap and flexible

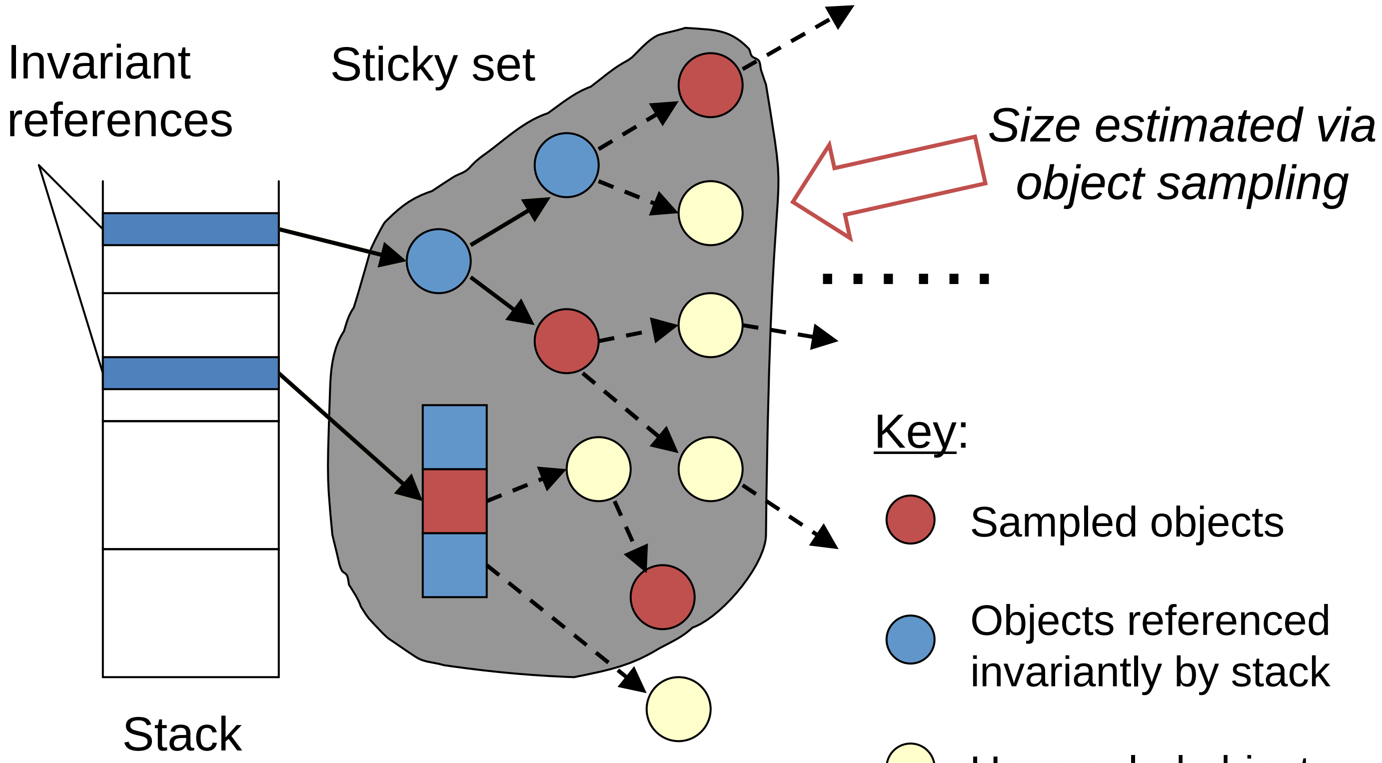


Time:	t0	t0	t1	t1
Processor:	p0	p1	p0	p1

Stack Invariants

- Because JVM is a "stack machine"
  - Stack variables can be hint of constantly accessed objects
  - Temporary variables are useless
  - Those references constantly stay in the stack across snapshots taken (we call them ***stack invariants***) are good hints of SS.
  - Usually stack invariants are the entry points of SS and important data structures like Hashmap, TreeMap, Linked List

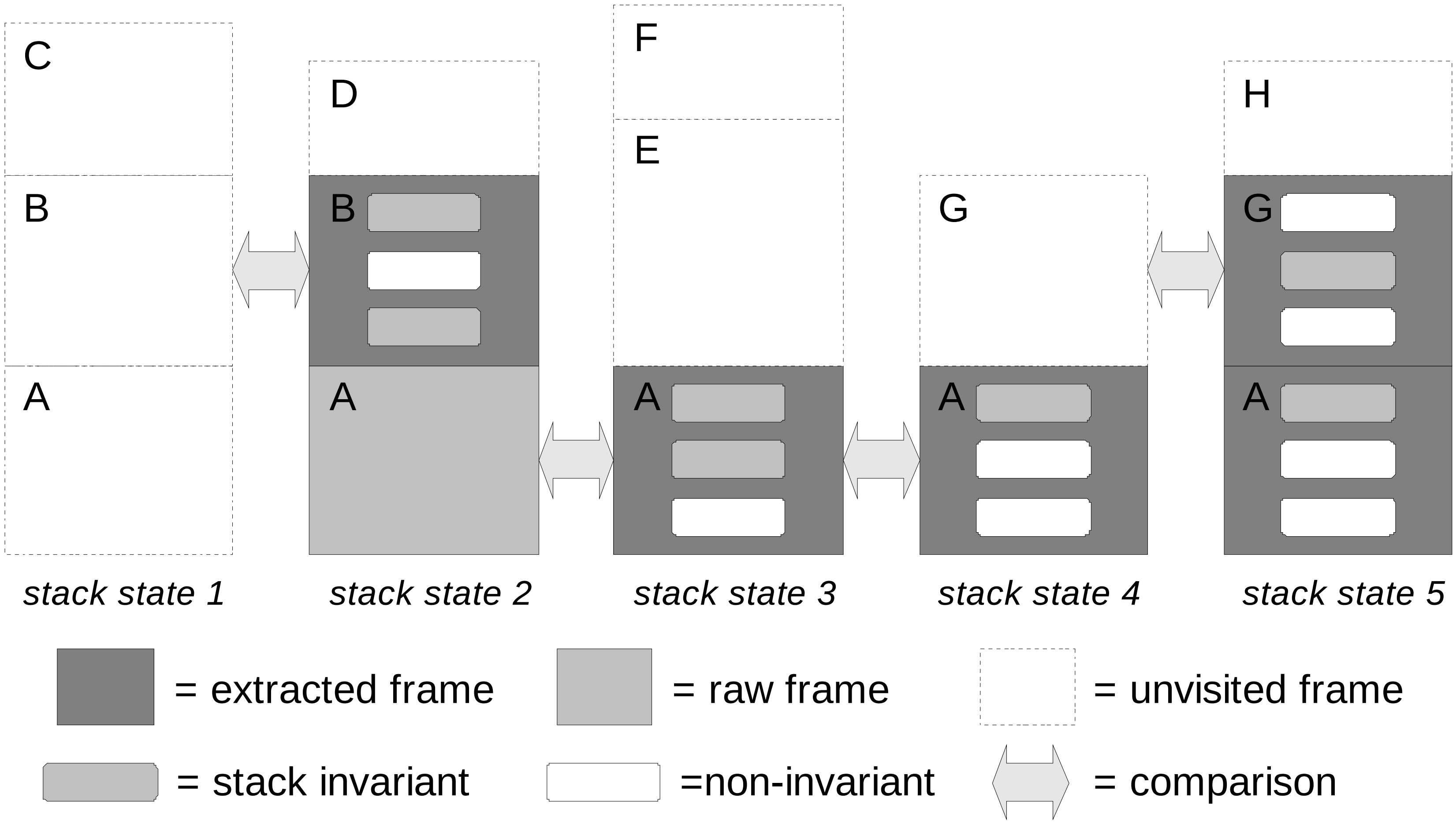
Stack Invariants (Cont')



Adaptive Stack Sampling

- Deduce invariants by comparing stack state snapshots frame by frame
- Adaptive optimization
  - Adjustable timer controlling which period of time to do stack sampling
  - Stack frame added with “visited” flag
    - If not touched across two sampling rounds, no need to sample it
  - Lazy Extraction: Capture frames in raw (native) form first
    - If a frame is not accessed again, no overhead
  - Compare two frames by “probing”
    - For each remaining invariance in old frame, check corresponding one in new frame.

Adaptive Stack Sampling (2)



Outline

1. Background
2. Challenges and Problems
3. Adaptive Object Sampling
4. Adaptive Stack Sampling
5. **Performance Evaluation**

Experiments

- Tests
  - Measure accuracy (shown already)
  - Measure overheads
    - Sampling-based access tracking
    - Computation of TCM
    - Stack profiling
  - Evaluate benefit over cost
- Application benchmarks
  - Ported from SPLASH2 to Java version
  - Barnes-Hut: fine-grained
  - Water-Spatial: medium-grained
  - SOR: coarse-grained
- Experimental environment: a segment of 8 Intel P4 nodes over Fast Ethernet

Experiments

- Tests

Benchmark	Problem Size		Sharing	
	Data set	***Rounds***	***Granularity***	***Object size***
SOR	2K × 2K	10	Coarse	each row at least several KB
Barnes-Hut	4K bodies	5	Fine	each body less than 100 bytes
Water-Spatial	512 molecules	5	Medium	each molecule about 512 bytes

- Application benchmarks
  - Ported from SPLASH2 to Java version
  - Barnes-Hut: fine-grained
  - Water-Spatial: medium-grained
  - SOR: coarse-grained
- Experimental environment: a segment of 8 Intel P4 nodes over Fast Ethernet

Object Sampling Overheads

CPU Overhead of logging accesses into OALs

Benchmark	*Size*	*Original Time(ms)*	*Sampling Frequency*			
			1X	*4X*	*16X*	*Full*
SOR	2K*2K	24250	N/A	N/A	N/A	24360(0.45%)
Barnes-Hut	4K	53250	52636(-1.15%)	52742(-0.96%)	53354(0.20%)	53844(1.12%)
Water-Spatial	512	29461	29507(0.15%)	29545(0.28%)	N/A	29717(0.87%)

Overhead of Sending OALs

Benchmark	*Size*	*Original Time(ms)*	*Sampling Frequency*			
			1X	*4X*	*16X*	*Full*
SOR	2K*2K	3954	N/A	N/A	N/A	4035(2.04%)
Barnes-Hut	4K	19557	19426(-0.67%)	19712(0.79%)	19824(1.36%)	20805(6.38%)
Water-Spatial	512	7942	8186(3.07%)	8252(3.90%)	N/A	8340(5.01%)

(a) Overhead of Total Execution Time

Benchmark	*Size*	*GOS Volume(KB)*	*Sampling Frequency*			
			1X	*4X*	*16X*	*Full*
SOR	2K*2K	4491	N/A	N/A	N/A	990(22.05%)
Barnes-Hut	4K	60130	140(0.23%)	525(0.87%)	2310(3.84%)	8309(13.82%)
Water-Spatial	512	31240	828(2.65%)	879(2.81%)	N/A	2589(8.29%)

(b) Overhead of Network Bandwidth

Object Sampling Overheads

- CPU overhead of computing TCM is the greatest overhead in the profiling subsystem
  - When system scales, TCM becomes bottleneck soon!
  - So sampling must be done …

Benchmark	*Size*	*Original Time(ms)*	*Sampling Frequency*			
			1X	*4X*	*16X*	*Full*
SOR	2K*2K	3954	N/A	N/A	N/A	870(22.00%)
Barnes-Hut	4K	19557	1568(8.02%)	1683(8.61%)	2327(11.90%)	4609(23.57%)
Water-Spatial	512	7942	323(4.07%)	347(4.37%)	N/A	749(9.43%)

Stack Profiling Overhead

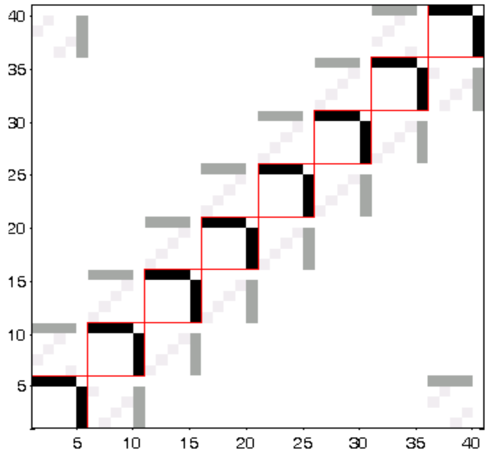
- Timer-based control of stack sampling phases saves over half of overheads
- Lazy extraction saves up to 1/3 overheads

Benchmark	Data Set Size	Baseline Exe Time	+ Stack Sampling Overhead				+ Sticky-set Footprinting Overhead				+ Sticky-set Resolution Overhead
			Immediate Extraction		*Lazy Extraction*		*Nonstop*		*Timer-based (100ms)*		
			4ms	*16ms*	*4ms*	*16ms*	*4X*	*Full*	*4X*	*Full*	
SOR	1K×1K	6201	6216 (0.24%)	6207 (0.10%)	6211 (0.17%)	6206 (0.08%)	6714 (8.28%)	6707 (8.17%)	6519 (5.13%)	6480 (4.50%)	6639 (1.85%)
Barnes-Hut	4K	93857	94947 (1.16%)	94657 (0.85%)	94697 (0.89%)	95209 (1.44%)	98968 (5.45%)	102190 (8.88%)	93649 (-0.22%)	102334 (9.03%)	97585 (4.20%)
Water-Spatial	512	59105	59232 (0.21%)	59161 (0.09%)	59209 (0.17%)	59124 (0.03%)	59834 (1.23%)	61985 (4.87%)	59501 (0.67%)	60313 (2.04%)	60002 (0.84%)

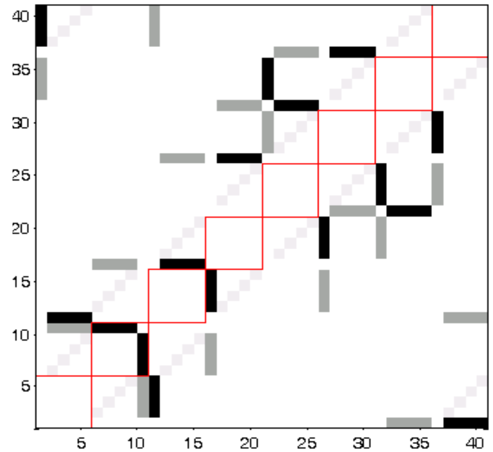
Effect of New Thread Migration Policy

- We assess this using an application “Customer Analytics“ with dynamic change in sharing patterns:

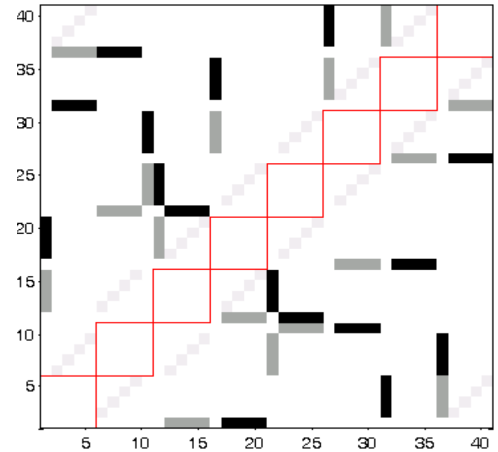
Epoch 1

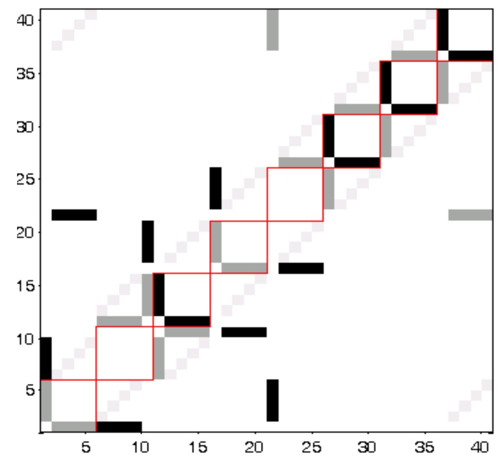


Epoch 2



Epoch 3





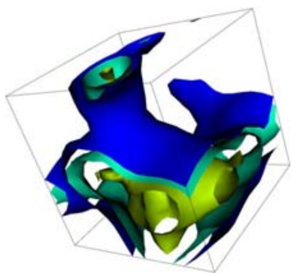
With thread migration enabled, the system strives for upkeep of most of the locality (see right fig).

Execution time shorten by over 60% compared to no migration.

Conclusion

- This work discusses a couple of advanced profiling strategies for optimizing locality
  - Adaptive object sampling
  - Online stack sampling
- Experimental results show
  - Low overhead
  - New thread migration policies based on
    - Profiled thread-thread correlation
    - Profiled per-thread sticky set
  - Can shorten much the execution on the distributed runtime system

Thank You !



Any Questions or Suggestions?

Contact Details

King Tin Lam
email: ktlam@cs.hku.hk

For more information, please visit

HKU Systems Research Group
http://www.srg.cs.hku.hk/

Dr. C.L. Wang's webpage:
http://www.cs.hku.hk/~clwang/