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

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

- Efficiency
- Programmability

- Bind to a memory consistency model
- Resemble ease of shared memory
- Less efficient
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)$^2$AS

- **Profile-Guided PGAS** (PG$^2$AS)
  - 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

- **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

Something new in this paper

Previous distributed JVM research (e.g. cJVM, JavaSplit, JESSICA, ...)
Techniques to improve locality

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

```
node 1

T1

node 2

T2

remote access

objects
```
Techniques to improve locality

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

![Diagram showing node 1 and node 2 with remote access and objects T1 and T2.](image)
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)
A cluster-wide JVM with
- Dynamic thread mobility in JIT mode
- Global Object Space (GOS)

Java Enabled Single System Image Computing Architecture
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
Problems for OO-Based Systems

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

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

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

Page size: 4KB
Page size: 128 byte
Challenge 2

- Thread migration cost is ill-modeled in past research.
  - Suppose thread \( T \) has \( n \) frames
    \[
    t_{\text{mig}}(T) = \sum_{i=1}^{n} [t_{\text{capture}}(i) + t_{\text{restore}}(i)] + \alpha + \frac{\sum_{i=1}^{n} L_{\text{frame}}(i)}{\beta}
    \]
    ... (1)

- Did not consider **indirect** cost of subsequent object misses after migration → inaccurate decisions

- How about including cost of shipping the thread’s working set?
  \[
  t_{\text{mig}}(T) = \sum_{i=1}^{n} [t_{\text{capture}}(i) + t_{\text{restore}}(i)] + \alpha + \frac{\sum_{i=1}^{n} L_{\text{frame}}(i) + W_T(t, \tau)}{\beta}
  \]
  ... (2)

- Yes! But not the best model for the migration cost
Suppose T1 accesses within the same interval:
- A (1 time), B (1 time), C (4 times)
- $W_{T1} = \{A, B, C\}$

(1) Without migration:
- Fetching roundtrips = 3

(2) With migration:
- Fetching roundtrips = 4
Challenge 2 (Cont’)

(3) With migration prefetching $W_{T_1}$:

$W_{T_1} = \{A, B, C\}$

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 $W_{T1}$:

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

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

Track access frequency

Fetching roundtrips = 3

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

```
gap=3    1  2  3  4  5  6  7  8  9  10  11  12  .....  
gap=5    1  2  3  4  5  6  7  8  9  10  11  12  .....  
gap=7    1  2  3  4  5  6  7  8  9  10  11  12  .....  
```

|=sampled  |means unsampled
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:

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

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

(Euclidean distance)  
(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

<table>
<thead>
<tr>
<th>Time:</th>
<th>Processor:</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>p0</td>
</tr>
<tr>
<td>t0</td>
<td>p1</td>
</tr>
<tr>
<td>t1</td>
<td>p0</td>
</tr>
<tr>
<td>t1</td>
<td>p1</td>
</tr>
</tbody>
</table>

```
01
o5
int
o4
float
int
o2
stack

05
o2
int
o1
double
float
int
o2
int
stack

01
o5
int
o2
do1
float
int
o2
o1
o100
int
o9
o10
stack
```
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.

```
  stack
  o1
  o2
  int
  double
  float
  int
  stack

  stack
  o1
  o5
  int
  int
  int
  int
  stack

  stack
  o10
  o9
  o100
  int
  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

```
stack
o2
int
float
stack
o4
int
o2
stack
o5
int
o1
float
int
t0
p0
Time: t0
Processor: p0

stack
o2
int
float
int
stack
o1
t0
p1
Time: t0
Processor: p1

stack
o1
int
o5
t1
p0
Time: t1
Processor: p0

stack
o1
int
o5
t1
p1
Time: t1
Processor: 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’)

Invariant references

Sticky set

Stack

Size estimated via object sampling

Key:
- Sampled objects
- Objects referenced invariantly by stack
- Unsampled objects
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)

- stack state 1
- stack state 2
- stack state 3
- stack state 4
- stack state 5

- C
- D
- E
- F
- G
- H

- B
- A

= extracted frame
= raw frame
= unvisited frame
= stack invariant
= non-invariant
= comparison
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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem Size</th>
<th>Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data set</td>
<td>Rounds</td>
</tr>
<tr>
<td>SOR</td>
<td>2K × 2K</td>
<td>10</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K bodies</td>
<td>5</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>512 molecules</td>
<td>5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>Original Time(ms)</th>
<th>Sampling Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X</td>
</tr>
<tr>
<td>SOR</td>
<td>2K*2K</td>
<td>24250</td>
<td>N/A</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K</td>
<td>53250</td>
<td>52636(-1.15%)</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>512</td>
<td>29461</td>
<td>29507(0.15%)</td>
</tr>
</tbody>
</table>

### Overhead of Sending OALs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>Original Time(ms)</th>
<th>Sampling Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X</td>
</tr>
<tr>
<td>SOR</td>
<td>2K*2K</td>
<td>3954</td>
<td>N/A</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K</td>
<td>19557</td>
<td>19426(-0.67%)</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>512</td>
<td>7942</td>
<td>8186(3.07%)</td>
</tr>
</tbody>
</table>

(a) Overhead of Total Execution Time

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>GOS Volume(KB)</th>
<th>Sampling Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1X</td>
</tr>
<tr>
<td>SOR</td>
<td>2K*2K</td>
<td>4491</td>
<td>N/A</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K</td>
<td>60130</td>
<td>140 (0.23%)</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>512</td>
<td>31240</td>
<td>828 (2.65%)</td>
</tr>
</tbody>
</table>

(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 ...

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Size</th>
<th>Original Time(ms)</th>
<th>1X</th>
<th>4X</th>
<th>16X</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR</td>
<td>2K*2K</td>
<td>3954</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>870(22.00%)</td>
</tr>
<tr>
<td>Barnes-Hut</td>
<td>4K</td>
<td>19557</td>
<td>1568(8.02%)</td>
<td>1683(8.61%)</td>
<td>2327(11.90%)</td>
<td>4609(23.57%)</td>
</tr>
<tr>
<td>Water-Spatial</td>
<td>512</td>
<td>7942</td>
<td>323(4.07%)</td>
<td>347(4.37%)</td>
<td>N/A</td>
<td>749(9.43%)</td>
</tr>
</tbody>
</table>
Stack Profiling Overhead

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Set Size</th>
<th>Baseline Exe Time</th>
<th>+ Stack Sampling Overhead</th>
<th>+ Sticky-set Footprinting Overhead</th>
<th>+ Sticky-set Resolution Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Immediate Extraction</td>
<td>Lazy Extraction</td>
<td>Nonstop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4ms</td>
<td>16ms</td>
<td>4ms</td>
</tr>
<tr>
<td>SOR 1K × 1K</td>
<td>6201</td>
<td></td>
<td>6216 (0.24%)</td>
<td>6207 (0.10%)</td>
<td>6211 (0.17%)</td>
</tr>
<tr>
<td>Barnes-Hut 4K</td>
<td>93857</td>
<td></td>
<td>94947 (1.16%)</td>
<td>94657 (0.85%)</td>
<td>94697 (0.89%)</td>
</tr>
<tr>
<td>Water-Spatial 512</td>
<td>59105</td>
<td></td>
<td>59232 (0.21%)</td>
<td>59161 (0.09%)</td>
<td>59209 (0.17%)</td>
</tr>
</tbody>
</table>
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/