Lightweight Application-level Task Migration for Mobile Cloud Computing

Ricky K. K. Ma, **Cho-Li Wang**
28 Mar 2012

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

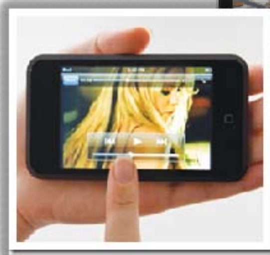
Outline

- Research background and motivation
- System design and implementation
- Performance evaluation

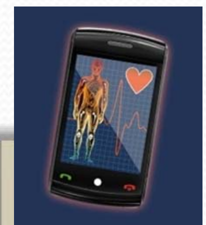
Background

- **Mobile cloud computing**:
  - Mobile apps or widgets connect to the Cloud
  - Support more complex and wider range of applications

More than 4.5 billion mobile-phone users all over the world.



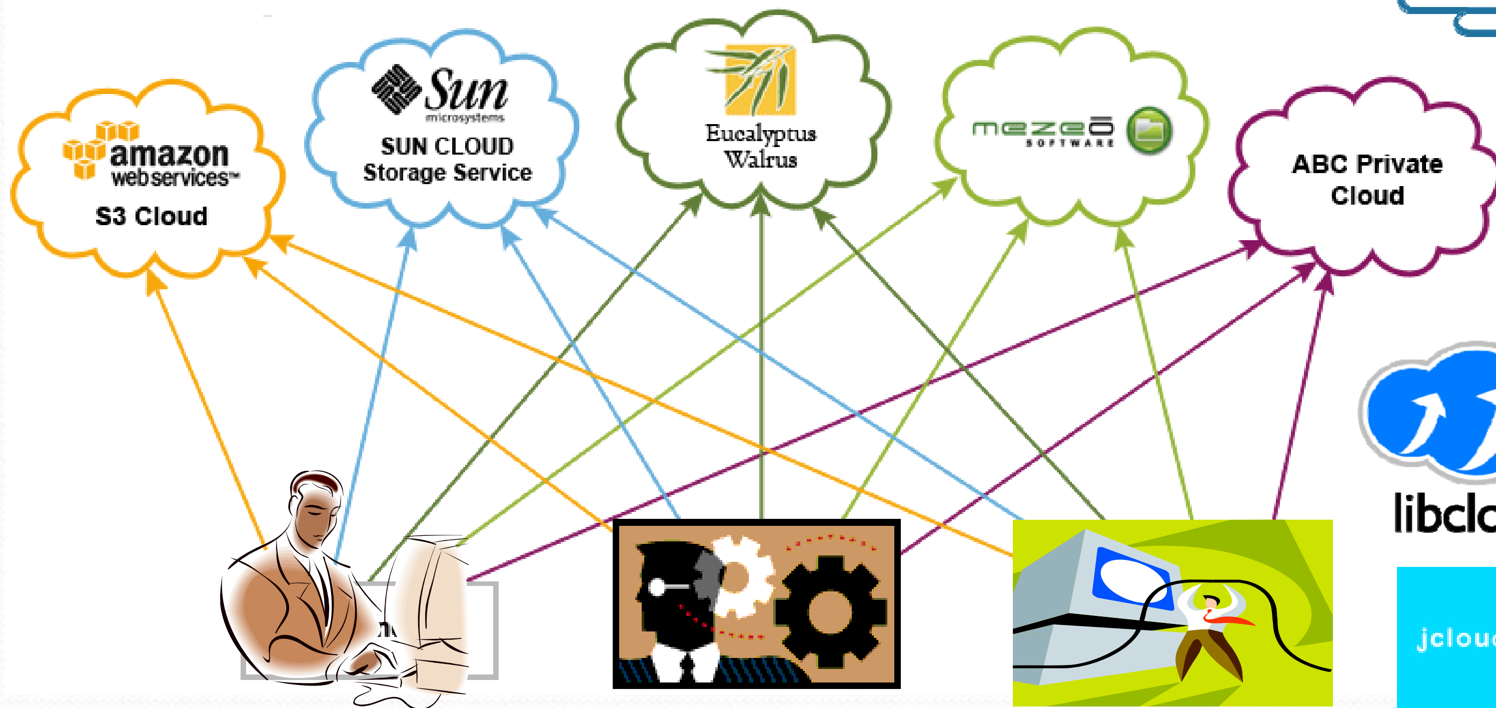
“Music Anywhere”



AI voice-recognition engines

Problems

- **API lock-in** → **Service Provider lock-in**
- **Client-server model**: **restricted form of computing**



Motivation:

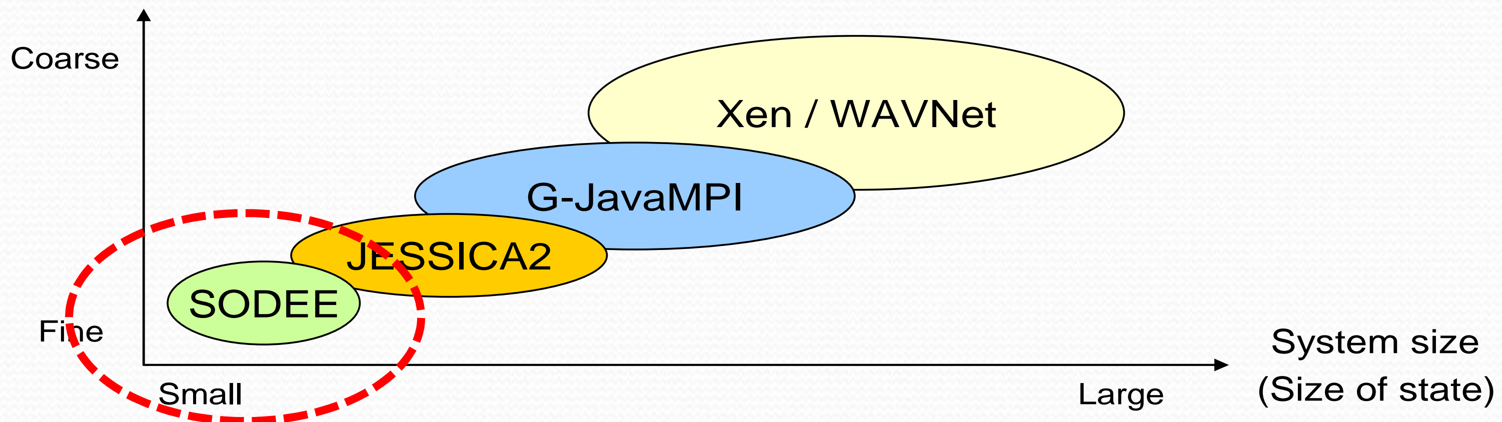
- **Migration techniques are required to dynamically move computation between mobile nodes and cloud nodes:**
  - **Low overhead:**
    - Especially when using in mobile nodes where processing power and resources are very limited
  - **Portable:**
    - Heterogeneous mobile nodes + Heterogeneous cloud nodes
    - Task migration among mobile nodes and cloud nodes
    - (Language-level virtualization for Cloud Computing)

HKU eXCloud Project Overview

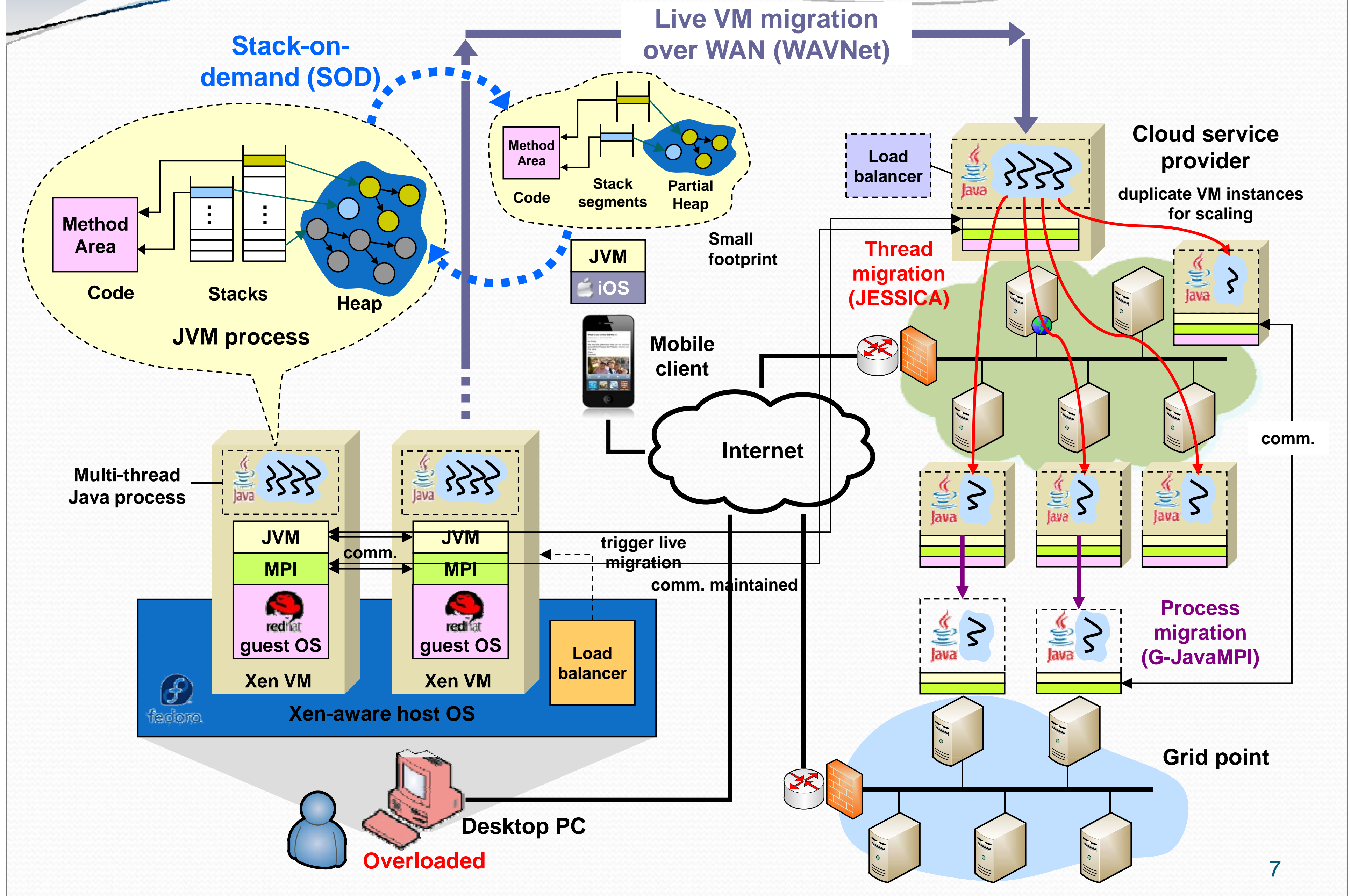
(part of China National Grid (CNGrid))

Multi-level Mobility Support

Granularity	Migration Technique (System)	Target System Type (Area)
Frame level	Stack-on-demand (SODEE)	Cloud, cloudlet, mobile network (WAN/LAN)
Thread level	Thread migration (JESSICA2)	Cluster (LAN)
Process level	Process migration (G-JavaMPI)	Grid (WAN/LAN)
VM level	Live VM migration (Xen)	Cluster (LAN)
VM level	Wide-area live VM migration (WAVNet)	Cloud, p2p/desktop cloud (WAN)

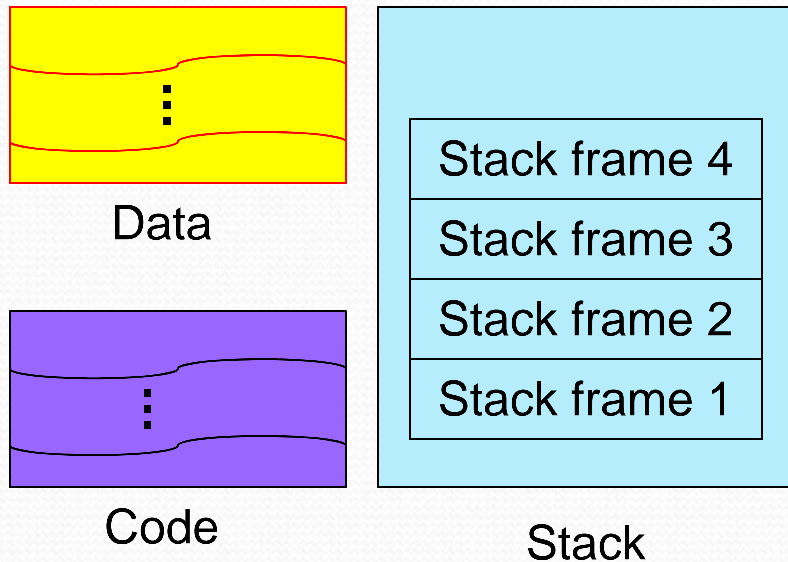


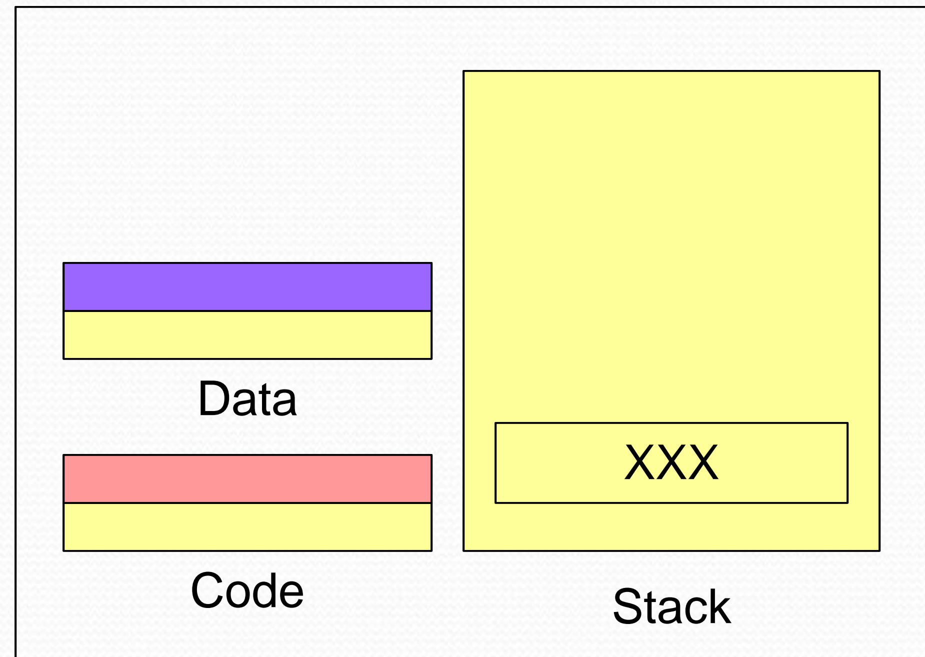
HKU eXCloud: Application Scenarios



Stack-On-Demand (SOD): Key Ideas

- Allow lightweight task migration

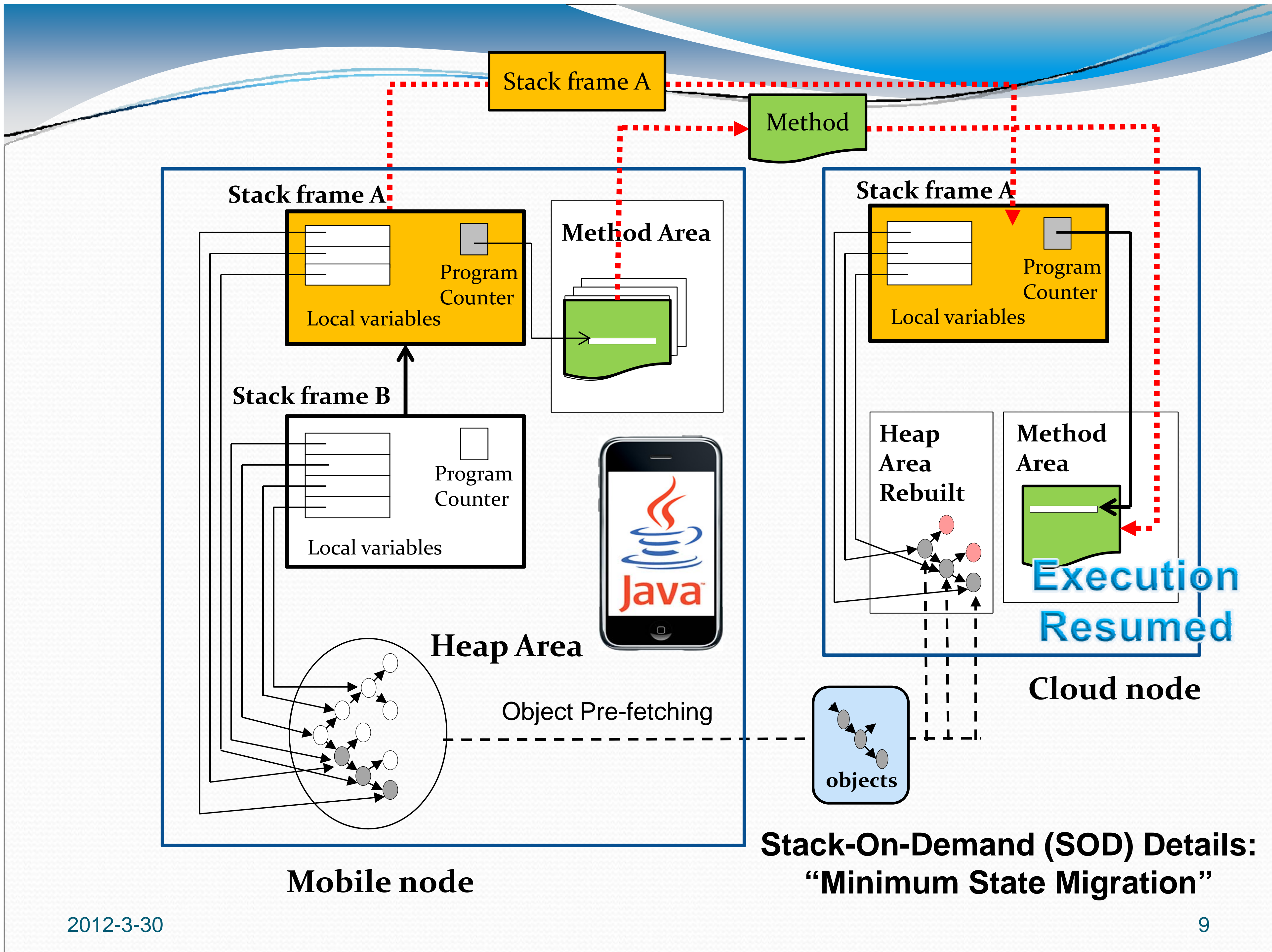




Migrating task on Source Node (a running thread)

Worker process on Destination Node

"A Stack-On-Demand Execution Model for Elastic Computing", IEEE ICPP2010.


Stack frame A
Method
Stack frame A
Program Counter
Local variables
Method Area
Stack frame B
Program Counter
Local variables
Java
Heap Area
Object Pre-fetching
Mobile node
Stack frame A
Program Counter
Local variables
Heap Area Rebuilt
Method Area
Execution Resumed
Cloud node
objects
Stack-On-Demand (SOD) Details:
"Minimum State Migration"

Existing Approaches to Migrate Tasks

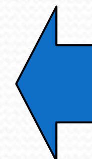
- **At JVM level**
  - Modifying JVM
    - *Sumatra, ITS, CTS, **JESSICA2***
- **As middleware**
  - **through JVMTI interface**
    - *CIA project, **G-JavaMPI***
  - JVM extension
    - *Mobile JikesRVM*
- **Application-level task migration**
  - **Rewriting bytecode**
    - *JavaGoX, Brakes, JavaSplit*
  - **Rewriting source code**
    - *WASP, JavaGo*

Requires intensive modification of JVM

Not portable for mobile nodes

JVMTI not available on mobile nodes

Requires certain extension of JVM



We focus on application-level task migration

Project	Level	Category	Granularity	Capturing techniques	Restoring techniques
Merpati	JVM	Interpreter	thread	Keep state in portable format	Reconstruct based on the state
JavaThread	JVM	Interpreter	thread	Keep state in portable format	Reconstruct based on the state
ITS	JVM	Interpreter	thread	Keep state in portable format	Reconstruct based on the state
CTS	JVM	JIT-compliant	thread	State in portable data structure	Reconstruct based on the state
JESSICA2	JVM	JIT-compliant	thread	JIT recompilation	Reconstruct based on the state
CIA	Middle-ware	extension of JVM	thread	JVMDI + bytecode instrumentation	JVMDI + bytecode instrumentation
Mobile JikesRVM	Middle-ware	extension of JVM	thread	Use extensions of JikesRVM	Use extensions of JikesRVM
Wasp	Application	Source-code preprocess	thread	1. Java Language extended 2. Exception (not asynchronous), but need to add migration points explicitly 3. Part of state always saved in each migration point	State-polling codes
JavaGo	Application	Source-code preprocess	thread	1. Java Language extended 2. Exception (not asynchronous), but need to add migration points explicitly	State-polling codes
Our approach	Application	Bytecode preprocess	**stack frame**	**Asynchronous exception (no need to add migration point and state-polling codes)**	**Twin Method Hierarchy (State-restoring codes executed only during restoration)**
JavaGoX	Application	Bytecode preprocess	thread	Exception (not asynchronous), but need to add migration points explicitly.	State-polling codes
MAG/ Brakes	Application	Bytecode preprocess	thread	State-polling codes.	State-polling codes

System Design

- **Design goal**
  - **Low overhead**
    - Allow lightweight task migration. Induce low overhead.
  - **Transparency**
    - No need for users to modify their programs
  - **Portability**
    - No need to use a specific JVM.
  - **Flexibility: Adaptation to new environment**
    - allow to use resources in new location to utilize resources (or better resource utilization)

Common approach in application-level migration

- Use of status-polling for detecting requests
- The status-polling codes are executed even when there are no migration

Instrumentation 1: Use of status-polling for detecting requests

```
-----------------------------------------------------------
original statements of the function
call func2()
if (isCapturing()) then
    store stackframe into context
    store artificial PC as index value
    return
end if
remaining statements of the function
```

1. Status-polling codes are added for each migration point

2. Status-polling codes are added after each function call

- The location of inserted codes determine the migration points
- Finer granularity of migration => more insertion of status-polling codes => large overhead

- Use of status-polling for detecting restoration
- Status-polling codes are added at the beginning of each function call
- The status-polling codes are executed even when there are no migration

Instrumentation 2: Status-polling for detecting restoration

```
----------------------------------------------------------------
if (isRestoring()) then
  get artificial PC from context
  switch (artificial PC)
    case invoke1:
      load stackframe
      goto invoke1
    case ...
      ...
  end switch
end if
original statements of the function
```

Status-polling codes are added at the beginning of each function call

Our approach

- **Fine-grained Task Migration**
  - Among cloud nodes + Between a mobile node and a cloud node
  - Granularity : **Java Stack Frame**
- **Two types of migration**
  - **Active:** Triggered by migration manager
    - E.g. over loading
  - **Pro-active:** Triggered by the program itself
    - Eg. `ClassNotFoundException`, `OutofMemoryException`
  - Migration manager would then receive the requests, choose the appropriate destination and perform the migration

- Task Migration
  - **State-capturing with Asynchronous Exception**
    - Avoid status-polling (less time overheads)
    - During normal execution, as no extra codes are executed, no overhead are introduced.
    - Allow finer granularity of migration

Instrumentation with use of asynchronous exception
1. try
2. **original statements of function**
3. catch MigrationException
4. **capture state**
5. throw MigrationException
6. **end try**

Capturing codes are inserted as exception handler

No significant overhead introduced during normal execution

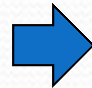
- **Issues working with asynchronous exception**
  - **Data inconsistency**
    - intermediate results are stored in **operand stacks**, or in **native methods**
    - **Solution: bytecode rearrangement**
      - **Extra local variables** are used to save intermediate results
      - **Extra flags** are used to inhibit migration at certain points
  - **Deadlock**
    - Can lead to deadlock if asynchronous exception is used in uncontrolled manner

State Restoring with Twin Method Hierarchy

- A bytecode instrumentation technique, minimize the overhead in normal execution
- **Twin Method Hierarchy**
  - Keep both **instrumented** and **original** methods
  - Normal execution: original methods
  - Restoration: the instrumented methods with restoration statements are executed.
    - **Checking statements are added at the beginning of the duplicated functions**
  - When restoration is completed, the original method will be executed

- Example

```
void func1(){
  func2();
  return;
}
```



1. During normal execution, original method func1() and func2() are executed => no overhead introducted

2. After restoration, method func1() and func2() are executed => no overhead introduced

```
void func1(){
  func2();
  return;
}
void SOD_func1(){
  if (isRestoring()) {
     restore_state();
     if (need_restore_other_frame)
        goto Label1
     else
        goto previously_suspended_location
  }
  func2();
Label2:
  return;
label1:
  SOD_func2();
  goto Label2
}
```

Original method

Method used during restoration

original method is executed after restoration has been done

Instrumented method is executed during restoration only

Performance Evaluation

- **Cloud server nodes**
  - Each node: 2 x Intel E5540 4-Core Xeon 2.53 GHz CPU, 32GB DDR3 RAM,
  - OS: Fedora 11 x86_64
  - JVM: Sun JDK 1.6 (64 bit)
  - Network: Gigabit Ethernet

- **Mobile nodes**
  - **iPhone 4 handset**: 800MHz CPU, 512 MB RAM
  - **JVM: JamVM** 1.5.1b2-3, slightly modified to expose the asynchronous exception API
  - Java class library: GNU Classpath 0.96.1-3
  - Connected to Cluster through Wi-Fi (bandwidth controlled by a router)

Performance Evaluation

- Focus on performance of task migration with SOD migration

Evaluations	Description
A	Overhead analysis in cloud nodes (No Migration)
B	Overhead analysis in mobile nodes (No Migration)
C	Migration from mobile node to cloud node (by **active migration**)
D	Migration from mobile node to cloud node (by **pro-active migration**)

Evaluation A & B: Overhead Analysis

- Testing programs

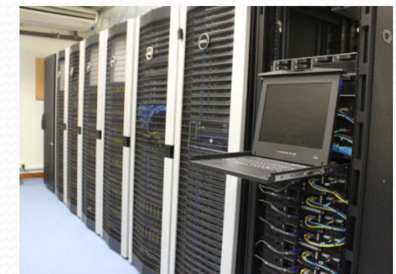
App	Description	Max. stack height	Total field size (byte)
Fib	Calculate 46th Fib. No.	46	< 10
NQ	Solve N-Queens problem with board size 14	16	< 10
FFT	Calculate 256-point 2D FFT	4	> 64M

- **Evaluation of Three Migration Techniques:**
  - **SOD migration using JVMTI (SOD_JVMTI)**
    - implemented as JVMTI agents
    - A middleware approach (Only available on Cloud nodes)
  - **SOD migration using status-checking (SOD_P)**
    - implemented at application level
  - **SOD migration using asynchronous exception (SOD_AE)**
    - implemented at application level

Evaluation A: Execution time on cloud nodes (overhead when NO migration)

	Orig	SOD_JVMTI		SOD_AE		SOD_P	
	time (s)	time (s)	*overhead (%)*	time (s)	*overhead (%)*	time (s)	*overhead (%)*
Fib	12.11	12.13	***0.17***	12.14	***0.25***	18.4	*51.78*
NQ	6.35	6.4	***0.79***	6.7	***5.51***	7.24	*14.02*
FFT	10.53	10.63	***0.95***	10.82	***2.75***	10.6	*0.47*

- ***SOD_JVMTI imposes the smallest overhead***
  - the lower layer implementations.
  - **Mobile devices do not support JVMTI**
- ***SOD_AE is slightly higher than SOD_JVMTI (< 5%)***



Evaluation B: Execution time on mobile nodes (overhead when NO migration)

	Orig	SOD_AE		SOD_P	
	time (s)	time (s)	*overhead (%)*	time (s)	*overhead (%)*
Fib	**10.85**	**10.86**	***0.09***	**15.58**	*43.59*
NQ	**32.13**	32.23	***0.31***	33	*2.71*
FFT	**5.39**	5.4	***0.19***	5.41	*0.37*

- SOD_JVMTI not reported (as JVMTI is not available for JVM in mobile devices)
- ***SOD_AE has the smallest overhead (<0.31%)***

Evaluation C: Migration from mobile device to cloud node

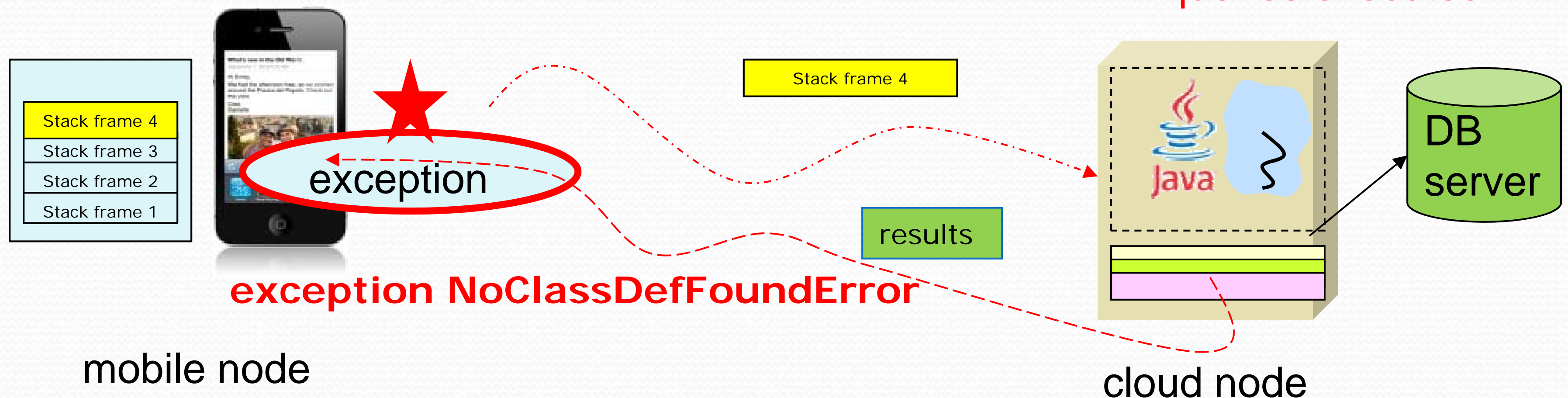
- **Active migration for performance improvement**
  - Migrate computation-intensive tasks from mobile devices to Cloud nodes. Upon finish of tasks, execution with data are migrated back to mobile devices.

- ***Performance gain : FFT: x3.8, NQ: x30, Fib: x57 times.***

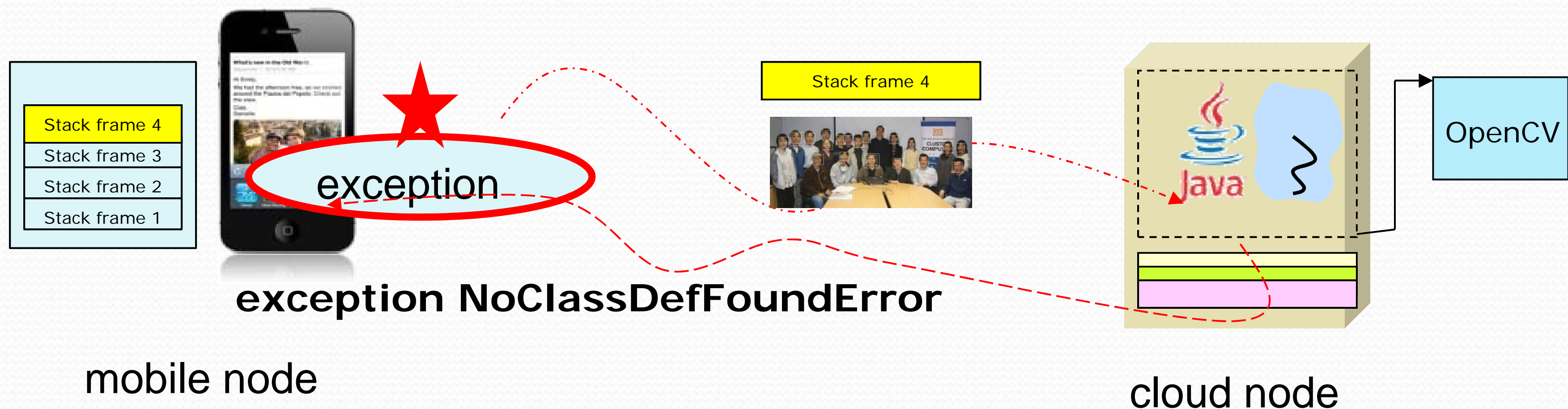
	exec. time w/o mig. (s)	exec. time w/ mig. (s)	Speed up	(A) capture time (ms)	(B) transfer time (ms)	(C) restore time (ms)	**Total migration latency (s)**
Fib	56.79	0.99	**57**	140.33	94.33	11.67	**0.246**
NQ	32.67	1.04	**30**	183.26	86.31	10.52	**0.280**
FFT	6.06	1.26	**3.8**	156.48	232.46	14.58	**0.403**

Total migration latency (s) = A+B+C

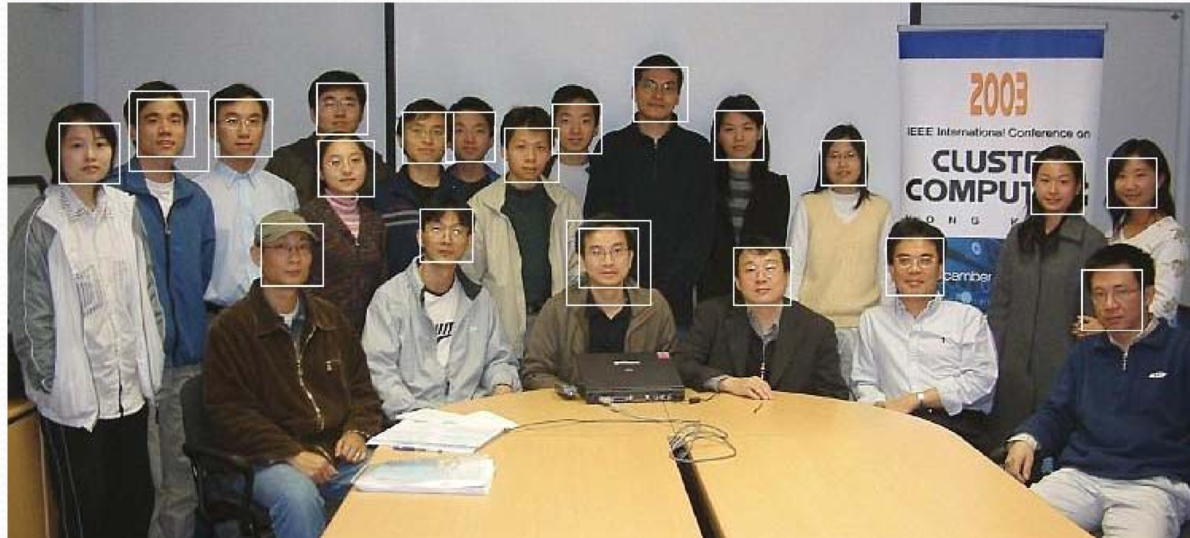
- **Evaluation D: Migration from mobile nodes to cloud node (Pro-Active)**
  - Two applications executed in mobile nodes
    - **DBRetrieve** and **FaceDetect**
    - Both require special resources not available in mobile nodes
  - **DBRetrieve** : During execution



- **FaceDetect:** Finds regions of faces in photos that are stored in iPhone
- **Requires OpenCV library**
  - open-source library for real-time computer vision and image processing
  - platform-dependent, not available in iPhone



Performance Evaluation



apps	capture time (ms)	transfer time (ms)	restore time (ms)	total migration latency (ms)
DBRetrieve	85	76	6	167
FaceDetect	103	155	7	265

- *If no SOD migration, the applications cannot be executed in mobile devices at all due to the missing resources*

Conclusion and Future Work

- **Java bytecode transformation technique**
  - Transparent task migration in a portable and efficient manner
- **Application level → Higher portability**
  - Migration can take place among mobile nodes and cloud nodes
  - Does not impose significant overhead on mobile devices
- **SOD Support More Flexible Computing in Cloud**
  - RMI-style, process roaming, workflow model
  - Improved resource utilization
  - Avoid (API & provider) lock-in
- **Future Work:**
  - Killer applications of SOD?
  - Migration policies? (Resource-driven, Cost-driven, Energy-aware,..)
  - Object pre-fetching, frame-based task scheduling, ..

Q & A