Lightweight Application-level Task Migration for Mobile Cloud Computing

Ricky K.K. Ma, Cho-Li Wang
Department of Computer Science
The University of Hong Kong
Hong Kong
{kkma, clwang}@cs.hku.hk

Abstract—**Mobile cloud computing allows mobile applications to use the enormous resources in the clouds. In order to seamlessly utilize the resources, it is common to migrate computation among mobile nodes and cloud nodes. Therefore, a highly portable and transparent migration approach is needed. In terms of portability, application-level migration with code instrumentation is the most portable approach. However, in the existing literature, this approach imposes significant runtime overhead, even when no migration takes place. Most of these works are for mobile agents, and migrations are to be invoked by the programs. Migration points are also restricted to certain locations where migration status is being polled. In this paper, we propose a Java bytecode transformation technique for realizing task migration without imposing significant overhead on normal execution. Asynchronous migration technique is used to allow migrations to take place virtually anywhere in the user codes, and the proposed Twin Method Hierarchy minimizes the overhead resulting from state-restoration codes in normal execution. We have implemented our approach in our middleware. The results show that our approach can allow lightweight computation migration at application level, achieve considerable speedups and utilize the cloud resources from mobile devices.**

Keywords-computation migration; migration technqiue; stack-on-demand

I. INTRODUCTION

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. It combines computing power and data storage into the web. Personal computers become thin clients to interact with clouds. Resource sharing and collaboration over the clouds helps existing computing resources to be consolidated to solve large-scale problems. As the resources can be highly diversified and their availabilities can change dynamically, a portable and lightweight task migration mechanism is needed to hide the heterogeneity and to move computation processes agilely between different locations.

By connecting mobile devices to a cloud, we form a mobile cloud for computing. Mobile applications and widgets connect to the clouds to support more complex and wider range of applications. Computation migration [2] can be used to allows cloud nodes and mobile devices to share the computation and hardware resources without restricted to the client-server model. However, due to the diversity of the devices, the migration mechanism and policy need to have several characteristics to adapt to different execution environments. It needs to allow migration in heterogeneous environment, as mobile devices and cloud nodes may have different instruction set architectures. Besides, in order to allow the migration to be used in mobile devices, the mechanism needs to be lightweight. As mobile devices have limited computing power, overhead of migration would have significant effects on the performance. Last but not the least, as mobile devices are often connected to the internet through mobile network, the bandwidth is small, and it cannot have large amount of data transmission. The migration mechanism should keep transferring as few data items as possible.

Java has been commonly used as the platform for developing mobile applications. There are various features of Java that favors such evolution. Java bytecode is machine-independent. It allows applications to be executed in different environments without modifications to the applications and the underlying environment. This makes applications to be executable on virtually all the devices. Besides, with the customizable class loading, Java bytecode can be transported and executed over the net easily, achieving code mobility. In addition, with the provided object serialization mechanism, Java objects can be migrated transparently in the internet, enabling data mobility without much programming burden. However, execution state of a Java program cannot be serialized or transferred. As a result, migrating computation from one node to another is not trivial.

Many researches have worked hard to allow Java processes to migrate, especially for agent-based systems [3,4,5,6]. Computation migration can be implemented at different levels: application level [7,8], middleware level [9,10], Java virtual machine level [11,12] and OS level [13]. The motivation for implementing migration feature at the application level is the portability. This approach avoids modifying the Java virtual machines, and the resultant software can be used on any JVM implementations among mobile devices and cloud nodes. However, this approach has two major drawbacks: i) it can incur considerable overhead, even when there is no migration; ii) migration can only be initiated by the application. Application-level migration is

not automatic to programmers. The migration points are constrained to the locations where migration request status is being. In order to allow fine-grained migrations, status-checking frequency needs to be increased, leading to even higher overhead. On the other hand, migration at Java virtual machine level [11] can minimize the overhead problem. The complete state of the migrating thread can be captured. However, the approach imposes portability problems since the JVM has to be extensively modified. The software can only run on the specialized JVM, leading to portability issues.

We propose a migration mechanism at application level without significant overhead. The migration mechanism makes use of asynchronous exception and bytecode instrumentation [14] to perform state capturing. It differs from other capturing approaches in the ways that it avoids repeatedly polling migration state and checking migration state after every function return. In the state-restoring, a technique, namely Twin Method Hierarchy, is used to minimize the overhead induced by the restoration codes. These techniques are designed to use for SOD migration [10,15]. The rest of this paper is organized as follows. Section 2 describes the source of overhead in existing application-level computation migration mechanisms. Section 3 presents our proposed migration mechanism. We discuss our approach to capture state and restore state. Section 4 presents evaluation methodology and the experimental results on different migration scenarios. Section 5 presents the related work. Finally we conclude this paper and outline several future works in Section 6.

II. BACKGROUND

To achieve application-level migration, application codes (source codes or bytecodes) are transformed by a preprocessor to acquire migration capability. During preprocessing, some extra codes are inserted for obtaining and sending meta-data required for migration. When these preprocessed applications run, the added codes would be also executed, getting the process ready to be migrated. As those codes always execute during execution, even when no migration takes place, overhead is still imposed. These added codes are mainly used for status-polling, state-capturing, state-restoring, and facilitating the communication among migrating nodes during migration. Several systems [16,17,18,19] follow this preprocessing approach, either working with source code [16,17] or bytecode [18,19].

The essence of the migration mechanism is in the manner it captures and restores the Java stack of a process. A stack contains multiple stack frames (or activation records), keeping the data (e.g. local variables) for the executing method instances. In order to migrate a process, all the stack content has to be copied to a user-accessible buffer for network transfer. However, due to Java's pointer-less design, a method's frame is only accessible to that particular method.

Instrumentation 1: Use of status-polling for detecting requests
1. original statements of the function
2. call func2()
3. **if (isCapturing()) then**
4. store stackframe into context
5. store artificial PC as index value
6. return
7. **end if**
8. the remaining statements of the function

FIGURE 1. STATUS-POLLING FOR STATE CATPURING

Without modifying the JVM, the only way to capture the complete stack is to iterate the execution control to each of the method instances, "inviting" them to recite their stack frame contents. In some systems [18], the preprocessor inserts a status-polling block after each method invocation. The status-polling is to check whether the execution is in the capturing mode (i.e., the process has decided to migrate and it is now capturing the stack content.) If so, the extra block will be executed in which the state of the current stack frame and the artificial program counter (PC) are saved. On completion, a premature return statement is executed so that the execution is passed to the caller. As the caller would also discover the capture state is in effect, it would save the content similarly and further pass control to its caller. The capturing process repeats until all frames are captured. The transformed code example is illustrated in Fig. 1. Suppose in the original function, `func2()` is being executed. In the instrumented codes, when migration occurs inside `func2()`, state of `func2()` would be captured. And then a `return` is executed to return the execution to the previous frame. After the return of `func2()`, `isCapturing()` is called to check if state-capturing is being taken. When state-capturing is being taken, `isCapturing()`returns true. The if-block is then executed, and a return at the end of the if-block is executed to return the execution to the previous frame. The process repeats until all the frames have been captured.

In other systems [19], exception instead of status polling is used to parse all stack frames. When migration takes place, a special Java exception is thrown. An inserted exception handler catches the exception and takes control of the execution. It saves the current frame state and throws another exception to invoke caller's exception handler. The process repeats until all the stack frames are captured. This approach eliminates the status-polling blocks. However, before any method invocation, the current stack frame needs to be captured first. Otherwise, the stack frame would be cleared during the occurrence of exception. As the stack frame is always captured during normal execution even when there is no migration, the heavy overhead is not yet avoided.

Instrumentation 2: Status-polling for detecting restoration
1. **if (isRestoring()) then**
2. get artificial PC from context
3. switch (artificial PC)
4. case $invoke_1$:
5. load stackframe
6. goto $invoke_1$
7. case ...
8. ...
9. end switch
10. **end if**
11. original statements of the function

FIGURE 2. STATUS-POLLING FOR STATE RESTORATION

To perform state-restoring, a status-polling statement is executed at the beginning of each method. The situation is illustrated in Fig. 2. `isRestoring()` is always checked every time when the function is being executed. When a process is being restored, the function `isRestoring()` would return true. Then the state-restoring code inside the if-block would be executed, jumping to the appropriate location of the method. State are restored, and then execution is restored to the previously suspended location.

To summarize, the overhead of application-level migration mainly comes from the code to repeatedly check for migration requests and the code to detect a restoration request on methods' beginnings. In addition there are also other constraints, migration points are constrained to the locations where certain checking methods are executed such as that in Fig. 1. Migrations can only occur at the location where `isCapturing()` is executed. These locations are called migration points. In order to achieve finer-grained migration, the frequency of execution of request-checking needs to be increased. However, this would increase the overhead which in turn hampers the performance.

III. OUR PROPOSED MIGRATION MECHANISM

A. *Overview of State-On-Demand execution (SOD)*

The proposed migration mechanism is developed based on our previous work in [10]. Our system does not migrate the whole processes or threads among mobile devices and cloud nodes. Instead, it performs Stack-On-Demand (SOD) execution [10] to migrate tasks. SOD is an ultra-lightweight computation migration in which only the top portion of the runtime stack is being migrated. This design exploits the temporal locality of stack-based execution, in which the most recent execution state always sit on the top segment of a stack. By a partial stack migration, this speculative approach can reduce the migration cost of a bulky stack pointing to many objects. In addition, SOD also allows tasks to move around a heterogeneous platform to survive in highly dynamic and unpredictable environments. As a whole, SOD offers a very flexible style of mobile cloud computing. Fig. 3

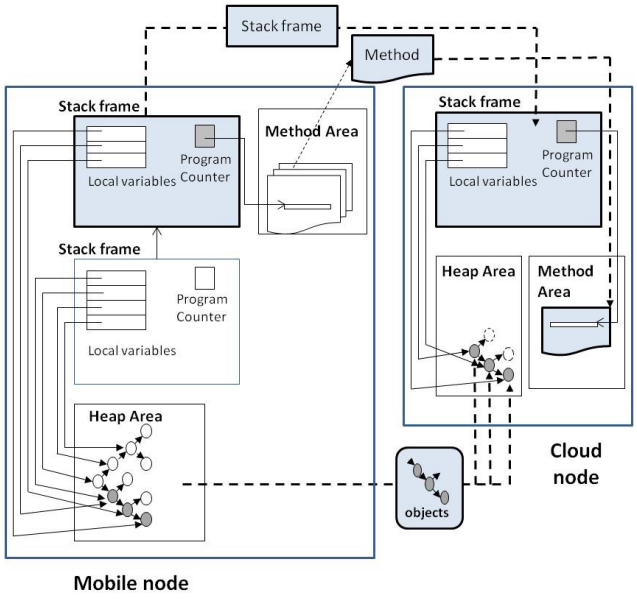

FIGURE 3. TASK MIGRATION FROM MOBILE NODE TO CLOUD NODE

shows the application scenario of SOD migration. As shown in the figure, migration is taken from a mobile device to a cloud node. Although the two computing machineries are in different capabilities and the network bandwidth between them is narrow, computation migration is possible with SOD. The mobile device transfers just enough portion of the stack to the cloud node, offloading the computation. Only the topmost stack frame, instead of the whole stack, is migrated to the cloud node. Besides, methods and objects are migrated on-demand. This minimizes the overhead in migrating tasks, and allows more flexible migration paths. More importantly, the migration is performed seamlessly without manual client-server programming.

SOD has several features that favor the execution in mobile cloud computing. One of them is its lightweight task migration. No matter how big the process image is, SOD migrates only the required part of the data to the destination site. This saves a lot of network bandwidth and takes less resource on the target sites. This feature allows SOD to access non-local idle computing resources and allow efficient bidirectional call flow between cloud and mobile devices. Mobile devices can make use of cloud resources seamlessly for performance scaling. Cloud nodes can also make use of the unique resources in mobile devices, such as photos taken and stored inside them. Another feature is SOD's fine-grained migration mechanism which allows different parts of the stack migrate concurrently to different sites, forming distributed workflow.

In our previous work, we proposed an approach which makes use of JVMTI to capture execution state in cloud nodes to perform SOD migration. With this tool interface, lightweight state capturing is allowed without the need of modifying JVM or extensive modifications of application

codes. However, JVMTI is available in some JVMs only. It is not available in resource-limited mobile devices. Besides, data captured by JVMTI are machine-dependent. Portability is an issue. In order to capture states in mobile devices in a portable manner, we propose an approach which performs task migration at application level to allow lightweight computation migration. In this approach, migrations are initiated in active and proactive ways, depending on the need of resources. State are captured with the use of asynchronous exception and are restored with the use of Twin Method Hierarchy approach in order to minimize the overhead.

B. Initiating a Migration

Based on the types of migration nodes, there can be three types of migrations: i) migration among cloud nodes; ii) migration from mobile devices to cloud nodes; iii) migration from cloud nodes to mobiles devices. Migrations among cloud nodes allow dynamic load balancing, improve data access locality, and achieve auto-provisioning of computing resources. Migrations from mobile devices to cloud nodes allow mobile applications to use the cloud resources seamlessly without following the client-server model. Migrations from cloud nodes to mobile devices migrate tasks to specific locations for unique resources. For instance, photos stored in a mobile phone can be used and found dynamically by a web server searching process which is originally executed in a cloud node.

Migrations can be classified as either active or proactive. The active ones are triggered by the migration manager when certain conditions of the system, such as the threshold of loading, are reached and detected. Proactive migrations are the ones triggered by the program itself indirectly. This would happen when the executing program has reached certain special state, such as `ClassNotFoundException` and `OutofMemoryException` exceptions. The exceptions are handled by the additional exception handlers which are instrumented during bytecode preprocessing. For example, in a mobile device, when the program is trying to load a certain library which is not available in the device, a `ClassNotFoundException` would be thrown. The exception handler would then capture the execution state and issue SOD migration request. The task would be migrated to a cloud node where the required library is available and resume execution. Upon task completion, execution is returned to the original program in the mobile device, and the execution of the program continues.

C. State Capturing with Asynchronous Exception

We propose a state-capturing mechanism that does not use polling to check for migration request. Besides, it does not need to add extra checking statement after the return of a function. In our proposed mechanism, in order to minimize the overhead during normal execution, migration codes are added as exception handlers. As exception handlers are not executed in the normal execution, negligible overhead would be added in the normal execution. Asynchronous exception is used to notify the application of migration request.

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**

FIGURE 4. DETECTION OF MIGRATION REQUEST USING ASYNCHRONOUS. EXCEPTION

Bytecode instrumentation is used to insert the state-capturing codes into the applications. Fig. 4 illustrates how applications are instrumented. Try-catch blocks are added to the applications to catch the asynchronous exception `MigrationException`. When there is any migration request, the migration manager would throw an exception to the target thread. The exception is asynchronous as it can be thrown at any time. It would be trapped by the try-catch block in the target thread. The corresponding exception handler would be invoked, in which state of the current threads are captured. In order to iterate through all the stack frames, the handler throws another `MigrationException` to notify the caller recursively. The current stack frame would pop out, and the lower stack frame would receive the exception. The capturing activity repeats in each stack frame until the last required stack frame is reached and captured.

In some situations, the use of asynchronous exception can lead to data inconsistency or deadlocks. Therefore, we augment the asynchronous exception approach with techniques that overcome these issues:

i. Avoidance of data inconsistency

Asynchronous exception can happen at any locations. There are locations that would cause data inconsistency if migration takes place. Intermediate results are often stored in operand stack. When exception is received, operand stack is cleared. So, if the operand stack is not empty and exception is received at that time, the operand stack would be cleared and values in the operand stack would be lost. This problem can be solved by using extra local variables to save intermediate results. Bytecode arrangement is taken to rearrange the codes so that all intermediate results are saved in extra local variables. So, even if the operand stack is cleared when exception is received, no intermediate results would be lost.

There are situations where bytecode instrumentation and bytecode rearrangement cannot be performed, e.g. in native methods. However, exception can still be received when these functions are being executed. This would lead to data inconsistency. In order to avoid the problem, a flag, namely NO_MIGRATE is used to inhibit the exception triggering. Before the execution of native methods, NO_MIGRATE is set to be true. When migration manager is about to trigger migration, it always checks whether NO_MIGRATE is set or not. If it has not been set, it would trigger migration. If it has been set, then it would schedule to do another checking at a later time.

ii. Avoidance of Deadlock and Related Data Problems

For multithreaded applications, asynchronous exception is inherently not safe [20]. It can lead to deadlock if asynchronous exception is used in an uncontrolled manner. If exceptions occur concurrently with synchronized methods, the latters' execution would be interrupted. The lock held by the interrupted thread would not be released. Deadlock would happen if there are threads trying to get into the function. Besides, due to the interruption, the remaining statements in the critical regions would not be executed. Due to the hazard, the API functions that are related to asynchronous exception have been deprecated. In our approach, deadlocks are avoided. We use bytecode instrumentation, exception handlers, and lock-checking by migration manager before triggering migration. In order to avoid the deadlock problem, during bytecode instrumentation, a try-catch block is added. When migration request is received, the migration manager would suspend the target thread. Then it would check whether the target thread is holding any locks or not. If so, the migration manager would instruct a dummy thread to take over the locks. Then the migration manager would issue an exception to the target thread to trigger state-capturing action. The catch block of the target thread would be executed. Locks would be released by the target inside the exception handlers, and state of the current stack frame would be captured. The state-capturing would then be proceeded to other stack frames as usual. The locks released by the target thread would be acquired by the dummy threads that were issued by the migration manager.

D. State Restoring with Twin Method Hierarchy

In order to allow methods to be restored, it is a common practice to perform status checking at the beginning of the related functions. However, the extra conditional branchings would induce significant overhead in the execution of applications. The overhead depends on the execution frequency of the functions.

We minimize the overhead by using a proposed approach, namely Twin Method Hierarchy (TMH). The heart of TMH lies in the idea of duplicating the original methods to allow the instrumented and original methods to be used respectively at different stages. During normal execution, the original methods are executed. During restoration, the instrumented methods with restoration statements are executed.

Methods are instrumented as follows:

i. Methods are duplicated into another set M', while the original methods are in the set M.

ii. In the duplicated methods M', checking statements are added at the beginning of the duplicated functions.

iii. In the normal execution, only methods in set M are executed.

iv. During restoration, methods in set M' are executed. When restoration is finished, those newly executed functions would be from set M.

An example of simplified restoration instrumentation is illustrated in Fig. 5. Fig. 5A shows the program codes before instrumentation, and Fig. 5B shows the instrumented program codes with major statements highlighted. The codes are instrumented in such a way that during normal execution, `func1()` and `func2()` are executed, while during state restoring, `SOD_func1()` and `SOD_func2()` are executed. Inside `SOD_func1()` and `SOD_func2()`, they are instrumented in the way such that when state restoration has been finished, the execution is then switched back to the original, non-instrumented program codes. If there are any further execution for the functions `func1()` and `func2()`, `func1()` and `func2()` are executed, instead of `SOD_func1()` and `SOD_func2()`. So, during migration, though additional statements would be executed inside `SOD_func1()` and `SOD_func2` during state restoration, when state restoration has been finished, the original, non-instrumented program codes would be executed. As there are no additional statements added into these original functions, there are no overhead imposed. As a result, the execution performance is returned to normal when migration has been done.

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

FIGURE 5A. ORIGINAL CODES WITHOUT ANY INTRUMENTATION

```
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
}
```

FIGURE 5B. PROGRAM CODES INSTRUMENTED WITH RESTORATION CODES

IV. PERFORMANCE EVALUATION

In this section, we evaluate our approaches with several applications in cloud nodes and mobile devices. The evaluations were conducted on a cluster of nodes interconnected by a Gigabit Ethernet network. Each node consists of two Intel E5540 Quad-core Xeon 2.53GHz CPUs,

TABLE 1. EXECUTION TIME IN CLOUD NODES

	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*

TABLE 2. EXECUTION TIME IN MOBILE DEVICE

	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*

32GB 1066MHz DDR3 RAM and a pair of SAS/RAID-1 drives. The OS is Fedora 11 x86_64. All nodes mounted the home directory on Network File System (NFS) to ease experiments with shared file access. The tested JVM version is SunJDK 1.6 (64-bit).

For the mobile devices, iPhone 4 handsets were used. It contains an Apple A4 CPU (800MHz), 512MB RAM, and 16GB storage. JamVM 1.5.1b2-3 (VJM) and GNU Classpath 0.96.1-3 (Java class library) are installed on the iPhone. It is connected through Wi-Fi connection to the cluster network. For JamVM, JIT is not available. Besides, in the original API, asynchronous exception is not available for use in applications. We have slightly modified the codes of JVM to expose the asynchronous exception API. Just a few lines of codes are added or modified. Other JVM implementations, such as Sun JDK, currently have provided API to allow applications to issue asynchronous exception at application level. However, Sun JDK is not available for iPhone. In this section, we focus on the evaluations of single-threaded applications.

A. Overhead of Normal Execution in Cloud Nodes

In this evaluation, we compare different mechanisms of state-capturing and restoring for SOD migration [10] in cloud nodes. These mechanisms are: use of JVMTI (SOD_JVMTI), the traditional use of status-checking (SOD_P), and our newly-proposed use of asynchronous exception (SOD_AE). SOD_JVMTI is a computation migration approach that we proposed in our previous work [10]. JVMTI [21] is used to capture execution state, and help to restore state for SOD migration. As JVMTI is close to the internals of JVM, it captures state more efficiently. However, SOD_JVMTI can only be used among cloud nodes as the tool interface is not available on mobile JVMs. On mobile devices, only our new SOD_AE and the traditional SOD_P can be used. Several computation-intensive applications were used. These applications are Fibonacci number calculations using recursion, solving N-Queens problem, 2D FFT calculations, and solving the Travelling Saleman Problem of *n* cities. Some micro-benchmarks are used to compare the performance of different state-capturing mechanisms. We would like to compare different comparison to capture different number of stack frames, and different size of objects.

We put the programs into execution on the cloud nodes and the results are shown in Table 1. Among different migration mechanisms, SOD_JVMTI has the lowest overhead. In SOD_JVMTI, implementations are made as agents instrumented into the JVM. As it is in the lower layer than the other two implementations, it imposes the smallest overhead. Both implementations of SOD_AE and SOD_P are made at application level. As SOD_AE avoids the execution of most of the extra statements and SOD_P cannot avoid the execution, SOD_AE imposes much less overhead than SOD_P during normal execution when there is no migration for most cases. In SOD_AE, the overhead can be reduced as much as from 51.78% to 0.25%.

B. Overhead of Normal Execution in Mobile Devices

We also evaluate the proposed approach in mobile device. Applications used are the same as section A. However, as the computing power of mobile device is much smaller than the cloud nodes used in Section A, some parameters used in this section are different from those in Section A to avoid very long execution time. As JVMTI is not available for JVM in the testing devices, SOD_JVMTI cannot be used. As a result, only SOD_AE and SOD_P are used and compared in this section. The results are shown in Table 2. For SOD_AE and SOD_P, the weight of overhead is relatively smaller than the weight of overhead in cloud nodes used in section A. SOD_AE has the small overhead in execution. This is also because SOD_AE has avoided execution of extra statements which cannot be avoided in SOD_P. In SOD_AE, the overhead can be reduced as much as 43% to 0.1%.

C. Migration for Performance Improvement

In this experiment, we would evaluate the performance gain of using the migration technique to migrate computation-intensive tasks from mobile devices to cluster nodes through Wi-Fi connection. In the experiment, we first executed the applications in a mobile device. When the computation-intensive task is just started, migration is taken to migrate the task from the mobile device to a cloud node where it is resumed to continue execution. When the task finishes, the results are sent back to the mobile device where the application continues the execution. The results are shown in Table 3. It is shown that the performance gain with migration can be more than 56 times. The large performance difference of cloud nodes and mobile nodes are mainly originated from the difference of computing power and JVM used. As the JVM used in the testing mobile devices does not provide JIT execution, the performance of execution at there is greatly affected.

TABLE 3. MIGRATION FROM MOBILE DEVICE TO CLOUD NODE

	exec. time w/o mig. (s)	exec. time w/ mig. (s)	gain (%)	capture time (ms)	transfer time (ms)	restore time (ms)	total migration latency (ms)
Fib	56.79	0.99	**5636**	140.33	94.33	11.67	**246.33**
NQ	32.67	1.04	**3041**	183.26	86.31	10.52	**280.09**
FFT	6.06	1.26	**381**	156.48	232.46	14.58	**403.52**

D. *Migration for Resource Utilization*

In this experiment, we simulate the scenario of migrating task from mobile device to cloud node in a proactive way to use the resources in the cloud nodes, as discussed in section III B. Two applications, namely DBRetrieve and FaceDetect, are used. The applications require some resources which are not available in iPhone. Without the use of migration, the program cannot be executed in mobile devices. DBRetrieve is a database client program which connects to a MySQL database server located in a cloud node to retrieve data. The program is executed in an iPhone which does not have the database driver and library to connect to the database. Besides, the database server is located behind firewall which it can be accessed by certain nodes only. During execution, when the program is trying to execute the statement which is trying to use the database driver, a `ClassNotFoundException` is thrown. SOD migration is then triggered in which the current stack frame, which is the topmost stack frame, is captured and transferred to that cloud node. Execution is resumed at its last suspended point in that cloud node, where connections to the database are made, and data are retrieved from the database and processed. After the completion of the execution of the migrated frames, the required results are returned to the calling program in the iPhone and the execution is resumed.

Another application, FaceDetect is a face detection program. It finds regions of faces in photos that are stored in iPhone. The searching is based on a profile that determines what frontal face features need to be detected in order to recognize as a face. The application makes use of a well-known library called OpenCV. OpenCV is an open-source library for real-time computer vision. However, the library is not available for iPhone. In our evaluation, we execute the application in iPhone. When the application is trying to use the library, an exception `NoClassDefFoundError` is thrown. This triggers a proactive migration, and the current stack frame is migrated to a cloud node where the library is available. The task is resumed. During execution of the task, photo data are fetched seamlessly from iPhone to the cloud node. Upon finishing the task, the resulting photo data are returned back to iPhone, and the execution is resumed. An example of the resulting photos is shown in Fig. 6. Faces detected are surrounded by white rectangles.

FIGURE 6. FACE DETECTION

TABLE 4. MIGRATION FROM MOBILE DEVICE TO CLOUD NODE IN PROACTIVE MIGRATION

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

The results are shown in Table 4. The migration latency for DBRetrieve and FaceDetect are 167ms and 265ms respectively. Most of the migration latency is originated from the capture time. Between DBRetrieve and FaceDetect, FaceDetect has larger capture time and transfer time. In FaceDetect, data of photo are sent from the mobile device to the cloud node, and data of resulting photo are sent from cloud node to mobile device. As more data are captured and transferred in FaceDetect than DBRetrieve, the capture time and transfer time of FaceDetect are larger accordingly.

V. RELATED WORK

MAG [3], Brakes [18] and JavaGoX [19] implement Java thread migration at application level. These systems are designed as mobile agent systems. They use preprocessor to instrument applications' bytecode. MAG and Brakes inserts state-capturing codes after each method invocation. The inserted capturing codes lead to the large overhead during migration. JavaGoX uses exception handlers to navigate the stacks. Though our approach also uses exception handlers to navigate the stack, our approach doesn't require the need of saving state before invoking functions, which are required in other approaches. Besides, in our approach, migration request is notified through using asynchronous exception, while in other approaches, migration request is triggered by the program itself. During normal execution, those approaches always execute the state-checking statements. In our approach, the state-checking statements are executed only during state-restoring. The statements would not be executed during normal execution.

Mobile JikesRVM [9] is a thread migration framework implemented as middleware on top of JikesRVM. However, it requires certain extensions of the underlying JVM, and it is not transparent to programmers. Cloudlet [22] and Clo-

neCloud [23] use VM migration to migrate computation from mobile devices in a portable manner. Cloudlet is a customized computing infrastructure which allows mobile devices to leverage resources of nearly cloudlets by VM migration. CloneCloud is a system that seamlessly offloads part of the execution of mobile applications from mobile devices to a computation cloud. Both systems require the use of VM in mobile devices. Migration in the systems are rather coarse-grained. Significant overheads are im-posed even when there are no migrations.

VI. Conclusion and Future Work

The paper proposes a Java bytecode transformation technique for realizing transparent task migration in a portable and efficient manner. The technique allows migration to take place at application level to allow high portability. Migration can take place among mobile devices and cloud nodes. It differs from other approaches that it does not impose significant overhead on execution when there is no migration occurred. Experiments show that the techniques allow lightweight migration at application level among mobile devices and cloud nodes. The techniques can be further explored and evaluated with policies, such as task distribution policy.

Acknowledgement

This research is supported by Hong Kong RGC Grant HKU7179/09E and Hong Kong UGC Special Equipment Grant SEG HKU09.

References

[1] P. Mell and T. Grance. “The NIST definition of cloud computing,” Technical report, National Institute of Standards and Technology, Information Technology Laboratory, 2011

[2] C. L. Wang, K. T. Lam, and K. K. Ma, "A Computation Migration Approach to Elasticity of Cloud Computing,” Internet and Distributed Computing Advancements: Theoretical Frameworks and Practical Applications, IGI Global

[3] R. F. Lopes, and F. J. D. S. E. Silva, “Migration Transparency in a Mobile Agent Based Computational Grid,” In Proc. of the 5th WSEAS Intl. Conf. on Simulation, Modeling and Optmization, pp. 31-36, Greece, August 17-19, 2005

[4] T. Illmann, T. Krueger, F. Kargl, and M. Weber, "Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture," In Proc. of the 5th Intl. Conf. on Mobile Agents, Atlanta, Georgia, USA, Dec 2001

[5] A. J. Chakravarti, X. Wang, J. O. Hallstrom, and G. Baumgartner, “Implementation of strong mobility for multi-threaded agents in Java,” In Proc. of 2003 Intl. Conf. on Parallel Processing, IEEE Computer Society, Taiwan, Oct. 2003.

[6] L. Bettini and R. D. Nicola. "Translating strong mobility into weak mobility, " In Proc. of the 5th Intl. Conference on Mobile Agents, pp. 182–197. Springer-Verlag, 2002.

[7] M. Factor, A. Schuster, and K. Shagin. “JavaSplit: a Runtime for Execution of Monolithic Java Programs on Heterogenous Collections of Commodity Workstations,” In Proc of the 5th IEEE Intl. Conf. on Cluster Computing (CLUSTER’03), p.110-117, HK, China, Dec 2003

[8] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. "Portable support for transparent thread migration in java," In ASM2000, pp. 29-43, 2000.

[9] R. Quitadamo, G. Cabri, and L. Leonardi. "Mobile JikesRVM: A framework to support transparent Java thread migration," Science of Computer Programming, 70(2-3):221-240, 2008

[10] R. K. K. Ma, K. T. Lam, C. L. Wang, and C. G. Zhang. "A Stack-On-Demand Execution Model for Elastic Computing," In Proc. of the 39th Intl. Conf. on Parallel Processing (ICPP2010), pp. 208-217, San Diego, California, USA, Sep 2010

[11] W. Zhu, C. L. Wang, and F. C. M. Lau. “JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support,” In Proc. of the IEEE 4th Int Conf. on Cluster Computing (CLUSTER 2002), pp. 381-388, Chicago, USA, Sep 2002

[12] S. Bouchenak and D. Hagimont. "Zero overhead java thread migration," Technical Report 0261, INRIA, 2002.

[13] S. Osman, D. Subhraveti, G. Su, and J. Nieh. “The Design and Implementation of Zap: A System for Migrating Computing Environments,” In Proc. of 5th Symposium on Operating Systems Design and Implementation, pp. 361–376, 2002

[14] “BCEL” Internet: jakarta.apache.org/bcel/

[15] R. K. K. Ma, K. T. Lam, and C. L. Wang. "eXCloud: Transparent Runtime Support for Scaling Mobile Appications in Cloud, " In Proc. of Intl. Conf. on Cloud and Service Computing (CSC2011), HK, Dec 2011

[16] S. Funfrocken, "Transparent Migration of Java-based Mobile Agents (Capturing and Reestablishing the State of Java Programs),” In Proc. of 2nd Intl. Workshop Mobile Agents 98 (MA'98), pp. 26-37, Stuttgart, Germany, September 9 - 11, 1998.

[17] T. Sekiguchi, H. Masuhara, A. Yonezawa, "A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation," In Proc. of 3rd Intl. Conf. in Coordination Langauges and Models (COORDINATION’99) Symposium, pp. 211-226, April 1999.

[18] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable Support for Transparent Thread Migration in Java," In Proc. of 2nd Intl. Symposium on Agent Systems and Applciations and 4th Intl. Symposium on Mobile Agents 2000 (ASA/MA2000), Zurich, Switzerland, September 13-15, 2000.

[19] T. Sakamoto, T. Sekiguchi, and A. Yonezawa, “Bytecode transformation for portable thread migration in Java,” In Proc. of 2nd Intl. Symposium on Agent Systems and Applciations and 4th Intl. Symposium on Mobile Agents (ASA/MA2000), Zurich, Switzerland, September 13-15, 2000.

[20] “Java Thread Primitive Deprecation” Internet: http://download.oracle.com/javase/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html

[21] “JVM Tool Interface (JVMTI) Version 1.1.” Internet: java.sun.com/javase/6/docs/platform/jvmti/jvmti.html

[22] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. “The Case for VM-based Cloudlets in Mobile Computing,” IEEE Pervasive Computing, 8(4), 2009

[23] B. G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. "CloneCloud: Elastic execution between mobile device and cloud," In Proc. of EuroSys 2011