Title	eXCloud: Transparent runtime support for scaling mobile applications in cloud
Author(s)	Ma, RKK; Lam, KT; Wang, CL
Citation	The 2011 International Conference on Cloud and Service Computing (CSC), Hong Kong, China, 12-14 December 2011. In Proceedings of CSC, 2011, p. 103-110
Issued Date	2011
URL	http://hdl.handle.net/10722/144630
Rights	Proceedings of the International Conference on Cloud and Service Computing. Copyright © IEEE.

eXCloud: Transparent Runtime Support for Scaling Mobile Applications in Cloud

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

Abstract—**Cloud computing augments applications with ease-of-access to the enormous resources on the Internet. Combined with mobile computing technologies, mobile applications can exploit the Cloud everywhere by statically distributing code segments or dynamically migrating running processes onto cloud services. Existing migration techniques are however too coarse-grained for mobile devices, so the overheads often offset the benefits of migration. To build a truly elastic mobile cloud computing infrastructure, we introduce eXCloud (eXtensible Cloud)–a middleware system with multi-level mobility support, ranging from as coarse as a VM instance to as fine as a run-time stack frame, and allows resources to be integrated and used dynamically. In eXCloud, a stack-on-demand (SOD) approach is used to support computation mobility throughout the mobile cloud environment. The approach is fully adaptive, goal-driven and transparent. By downward task migration, applications running on the cloud nodes can exploit or take control of special resources in mobile devices such as GPS and cameras. With a restorable MPI layer, task migrations of MPI parallel programs can happen between cloud nodes or be initiated from a mobile device. Our evaluation shows that SOD outperforms several existing migration mechanisms in terms of migration overhead and latency. All our techniques result in better resource utilization through task migrations among cloud nodes and mobile nodes.**

Keywords-stack-on-demand;mobile cloud;computation migration; cloud computing

I. INTRODUCTION

The essential characteristics of cloud computing, such as on-demand self-service, flexible network access, rapid elasticity and pay-per-use style of utility services, attract much attention from commercial users. However, despite the popularity of cloud computing, cloud computing is still an evolving paradigm [1]. There are some fundamental issues that hinder the widespread adoption of cloud computing in the future.

One of the issues is related to mobile cloud computing. With the desire of mobility of resources, mobile cloud computing has been evolved. It is basically the combination of mobile computing and cloud computing. Mobile devices connect to the cloud, and mobile applications make use of the resources of cloud. In mobile cloud computing, mobile applications can offload computation-intensive operations to clouds for speedup and use the resources for extend functionality, or simply draw services provided by clouds in client-server model. The importance of mobile cloud computing can be reflected by a recent study that more than 240 million businesses will use cloud service through mobile devices by 2015, and that would push mobile cloud computing revenues to an enormous amount [2]. However, in the current evolvement of mobile cloud computing, mobile applications can only use clouds in a restricted manner. Applications are executed in client-server models in order to use the cloud resources. This execution model is different from the one in pervasive computing. The basic problem that leads to this constrained execution model is the mobility of applications. In pervasive computing, computation components can move freely from one place to another. However, in mobile cloud computing, applications are not really "mobile". Data and application states cannot move freely between mobile devices and clouds. In the push-pull manner in the client-server model, data and state can be sent in a few ways only. When a client is requesting a service from the server, data and state are sent from client to the server. When a server is responding to a service, updated data and state are sent to the client. Apart from this request-reply occasion, data and state cannot move among clients and servers. In addition, only a small part of data can be sent between the mobile device and the clouds.

Some may argue that the above scenario is what cloud computing is: applications offload execution from mobile devices to the cloud, get the result back after execution, and cloud nodes never need to migrate application computation to the mobile devices. But we would argue that, although this constrained way may be enough for simple mobile applications, true mobility would allow powerful features which are essential to our *future* clouds. One of such features is the seamless and autonomous integration of mobile devices and clouds. With true mobility, it is no longer necessary to have mobile applications executed in a client-server model in order to use the cloud resources. Besides this, the unique resources of mobile devices, such as the bundled cameras and microphones, can be used freely by the computations that have been migrated to the clouds. In a broader view, with the capability to move computation across cloud nodes and mobile nodes, our future clouds can grow, in terms of resources, by utilizing resources whenever possible. Whether or not the resources are being used is another question relating to the policy.

Another issue is related to granularity of task mobility. Task mobility refers to the capability to allow users to continue their tasks on different nodes. The importance of task mobility in cloud computing can be reflected by the

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

work on VM migration. By realizing mobility through VM migration, resource utilization can be improved by load balancing. System serviceability and availability can also be improved by migrating applications from machines that have planned maintenance. However, just VM migration alone is far from enough for the need of mobility. As VM migration transfers the whole VM to another computer, it does not allow fine-grained load balancing among VMs. There would often be a dilemma of making migration decisions. For example, a migration would favor the execution of some tasks but penalize other tasks in the same VM. Therefore, we need better cloud computation migration techniques.

To support task mobility, the most classical method is process migration [3]. SPRITE, MOSIX, and the recent system OpenSSI [4] are typical examples of process migration systems. Process migration provides extra opportunities for dynamic load balancing and improving data locality or communication latency during the process's runtime. Despite these advantages, process migration has not been widely used. It is often too expensive to transfer the entire address space of a process. Some VM migration techniques improve on this by moving all dirty pages alone, yet the overheads involved are still much big for a mobile cloud environment. Moreover, most existing solutions require extensive modifications hacking into OS kernels [5], JVMs [6], system libraries [7] or applications [8]. These systems usually suffer from poor portability across constantly changing OS versions or heterogeneous hardware architectures.

To achieve more elastic cloud computing, we propose a middleware system, namely *Extensible Cloud* (eXCloud), with stack-on-demand (SOD) integrated atop VM systems to support multi-level mobility. SOD is based on a stack machine, specifically JVM in our case. Unlike traditional process migration which performs full-rigged state migration (including code, stack, heap and program counter), SOD migrates only the top stack frame, or top segment of frames, while the required code and heap data are brought in on demand subsequently. With SOD, we can dynamically move input processing to the location(s) where the greatest demand is being generated; we can also move the execution of a software component close to its associated data source for shorter access latency. We can offload a task onto a cloud platform if the desktop or mobile device has insufficient processing capacity or memory to carry out the computation. User tasks at a mobile device can be offloaded onto clouds to save energy consumption. It allows lightweight transparent migration among mobile devices and cloud nodes. Tasks can be migrated from mobile devices to cloud nodes, and from cloud nodes to utilize the resources.

The rest of this paper is organized as follows. Section II gives the introduction of eXCloud. Section III presents the system design and architecture of eXCloud. In Section IV, we evaluate and compare the performance characteristics of the various computation migration systems. Finally, Section V concludes our work.

II. EXCLOUD: MULTI-LEVEL MOBILITY SUPPORT FOR MOBILE CLOUD COMPUTING

A. Overview

In a mobile cloud, there are mainly two types of nodes—cloud nodes and mobile nodes. Cloud nodes refer to the computing nodes which are located statically. These nodes are powerful computing machines, equipped with powerful processors and plenty of memory to provide high performance computing. VM instances can be executed in these machines to provide elastic computing to clients. Mobile nodes refer to the mobile devices that are added dynamically to form the mobile cloud. These nodes are usually low-power and equipped with "just enough" resources, such as memory and storage. These devices are not designed to provide high-performance computation. As mobile applications become more and more complicated, much more computing power is needed to execute the applications. Resources from clouds can be used to execute the applications. However, among existing approaches, applications are often restricted to execute in a client-server manner, in which mobile devices act as a thin client and most computations are executed in the cloud nodes. This greatly reduces the application flexibility.

eXCloud is a multi-level mobile cloud infrastructure for providing transparent runtime support for scaling mobile applications. It allows different levels and different granularity of mobility in an adaptive and goal-driven manner. In this infrastructure, SOD is integrated atop VM. Live migration of VM instances [9] are allowed. SOD can also distribute and migrate the tasks across different nodes according to the following goals:

- *constraint-driven*: tasks are migrated when the executing node does not have sufficient resource or required resources. E.g. tasks are offloaded from mobile devices to the clouds when CPU or memory capacity is not enough, or certain required libraries are missing.
- *locality-driven*: migration is taken to move the computation closer to the data source for shorter access-latency.

B. State-On-Demand execution

SOD [10] 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–the most recent execution state always sits 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, mobile-agent solutions, which allow autonomous components to move around a heterogeneous network, seem best to survive in highly dynamic and unpredictable environments. So the SOD design also incorporates this notion, and as a whole, offers a very flexible style of mobile cloud computing in three features:

- *Lightweight task migration*: SOD copies only the required part of data to the destination. This saves a lot

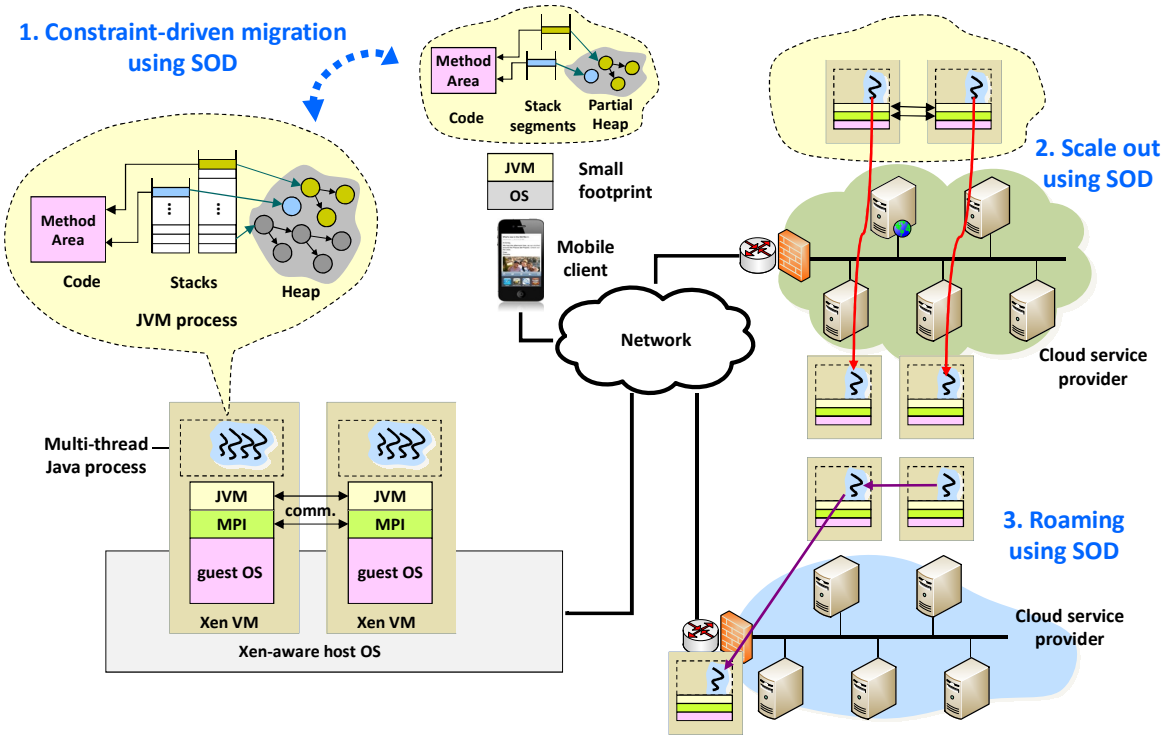

Figure 1. Task migration between cloud node and mobile node

of network bandwidth and reduces resource footprints on the target site. It allows a big task to fit into a small device in a discretized manner, with an extremely short freeze time. This allows quick access to non-local idle computing resources.

- *Distributed workflow style*: SOD allows different parts of the stack migrate concurrently to different sites, forming a distributed workflow. Freeze time between multiple hops is fully or partially hidden. Data locality can be enhanced, and streamlined elastic task scheduling on cloud servers is possible.
- *Autonomous task roaming*: SOD mimics strong-mobility mobile agents, and is capable of adapting to a new environment. Tasks can roam across a set of information bases for best locality. SOD-driven roaming can be more agile and flexible than traditional MAs because SOD is down to granularity of a method instead of the whole process.

C. Task Migration using SOD

Basically, different migration techniques focus on different computing applications and different execution paradigms. VM migration moves the whole VM, while SOD performs lightweight task migration to utilize resources. With SOD migration, only the topmost stack frame, or topmost segment of stack frames is transferred. This avoids the overhead and complexity of migrating the underlying stack frames, especially when the underlying stack frames are native frames or contain reference to native objects. Besides this, as SOD migration transfers heap data on demands, migrating large portion of heap data can be avoided. Fig. 1 shows the application scenario of SOD migration. In this multi-level migration system, apart from using SOD as a traditional migration mechanism to support load balancing and load sharing, SOD can be used in the following ways:

1. *Constraint-driven migration.* With SOD migration, tasks can be migrated among cloud nodes and mobile devices without significant overheads. 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. Migration among cloud nodes can allow dynamic load balancing, improved data access locality, and auto-provisioning of computing resources. Besides this, resources can be better utilized. As shown in Fig. 1, migration is taken from mobile devices to cloud nodes. This allows mobile applications to use the cloud resources seamlessly without following the client-server model. Tasks can also be migrated from cloud nodes to small-capacity devices to use the unique resources in the devices. E.g. Photos stored in a mobile phone can be used and found dynamically by a searching process of a web server which is originally executed in a cloud node.

Migration can be triggered either actively or proactively. In the active way, migration is triggered by migration manager. When certain conditions of the system, such as the threshold of loading, are reached and detected by the migration manager, the migration manager would issue the migration request and carry out migration. In the proactive way,

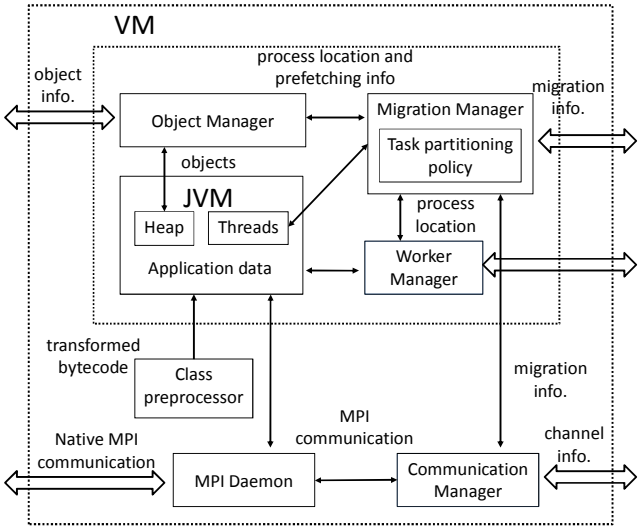

Figure 2. Main building modules of a eXCloud cloud node

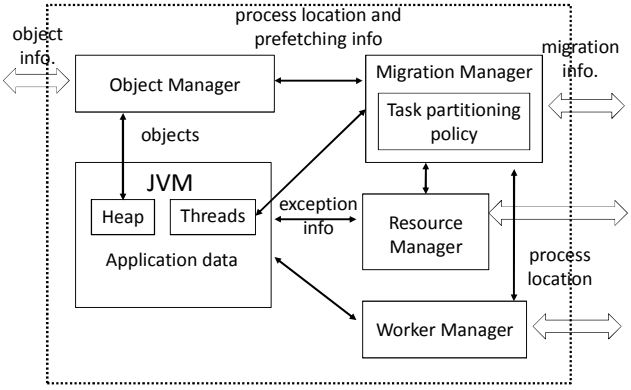

Figure 3. Main building modules of a eXCloud mobile node

migration is triggered by the program itself indirectly. This would happen when the executing program has reached certain special states, such as exceptions like `ClassNotFoundException` and `OutOfMemoryException`. The exceptions are handled by the exception handlers which are instrumented into the program 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. The task is resumed execution. Upon completion of the task, execution is returned to the original program in the mobile device, and the execution of the program continues.

2. *Scale-out.* This is mainly for parallel programs to allow dynamic load distribution, scale-up and scale-down auto-provisioning. With the use of restorable MPI layer, tasks in parallel programs can be scale-out to different machines to use the computing resources at there. Tasks can still communicate properly among each other after scale-out through the use of the restorable communication support.

3. *Task Roaming.* This is to roam tasks from a node to another node to improve data locality. Suppose a very large data file storing dynamic data is stored distributedly among a number of machines. When a searching process needs to perform searching among these files, in order to exploit the data locality, the searching process can be roamed among these machines to perform local search of the data file.

D. Adaptive migration mechanisms for task mobility

Cloud nodes and mobile nodes have different computing power. Besides this, their hardware and software architecture have different characteristics. Cloud nodes are powerful with plenty of computing related resources, such as memory and storage. VM instances are created in these nodes and computation tasks are performed insides these VM instances. Among the VM instances, similar hardware and software working environments are provided. For cloud nodes, computation performance is considered as one of the most important criteria. For mobile nodes, the nodes are often in different hardware and software configurations. As there are no VM instances between user applications and the underlying systems, portability becomes the most important criteria for execution. Due to the different criteria, different state-capturing and state-restoring mechanisms are used when migrations are taken places in different types of nodes. When state-capturing or state-restoring is taken in cloud node, states are captured using JVMTI functions. As JVMTI is a low-level layer that can access the internal of JVM, state can be captured more efficiently. However, when state-capturing or state-restoring is taken in mobile nodes, state are captured and restored at application level. This allows portable state migration.

III. System Design and Architecture

A. Design Goals

The SOD system has the following design goals:

1. *Low overhead.* The main objective of SOD is to deliver lightweight task migration. The total amount of overhead induced by the system must be low. The system must not induce delay or slowdown on the execution of the applications, especially during normal execution when there is no migration in place.

2. *Transparency.* The whole SOD mechanism needs to be transparent to users. There is no need for users to modify their programs, or to use specific libraries. Normal user applications are passed to the system for execution.

3. *Portability.* There is no need to use a specific JVM. Standard JVM and Java Standard Library should be used.

4. *Adaptation to the new environment.* When tasks are migrated to a new environment, they should be able to use the resources in the new location to utilize the resources and to improve the locality of resources.

SOD model was implemented with Java runtime as a migration middleware system named SOD Execution Engine (SODEE). SODEE is a layer between user applications and the underlying supporting components. The middleware is transparent to user applications. There is no need to modify source codes of applications. There are no specific restrictions on the application programs. In the underlying components, standard JVM are used. There are no modifications of JVM.

B. Main buildling modules

Fig. 2 and 3 illustrate the high-level design and the main building modules of a cloud node and a mobile node in eXCloud respectively. The main differences between the two type of nodes are that class preprocessor and MPI communication components are found in the cloud nodes only, while resource manager is found in mobile nodes only. Besides this, all the components in a cloud node are atop VM while there is no VM used in mobile node. This means that multi-level migration is available among cloud nodes only. For migration involving mobile nodes, only SOD migration can be taken. Though the high-level views in both nodes are very similar, the implementations are very different. This is mainly due to the different design criteria. One of the main criteria for a cloud node is elasticity, while it is portability for a mobile node.

The main modules are described as follows:

1. *Class preprocessor* is responsible for transforming Java application bytecode before it is loaded into JVM. The instrumentation is executed offline once for each related bytecode file, adding code for state capturing and restoring.

2. *Migration manager* is responsible for serving migration requests, and communicating with other migration managers to carry out the state and code migration. *Task migration policies* are used to dynamically determine the distribution of tasks during execution. The decision criteria include the resources needed by the task, and the locality information of data and resources. The decision would be used by migration manager to schedule SOD migration, and by object manager to do prefetching.

3. *Object preprocessor* is responsible for the object synchronization among sites. It serves other sites their required objects, and also updates objects which are received from other sites. As objects are migrated, they need to send from home site to the target site. Besides this, objects can be updated in other sites, and it is necessary to have the updated copy of objects for execution.

4. *Worker manager* is responsible for the managing worker processes. In SOD, a worker process is created in the target site to execute the migrating task on behalf of the home site. Before a migration request is received by a site, an instance of worker process can be created to stand-by for any coming SOD migration. This would minimize the initialization time on creating worker process instance. As the stand-by working processes are in suspended state, and the number of stand-by working processes is controlled by the worker manager, the amount of resources consumed by these stand-by processes would be very small, well under control of the worker manager.

5. *Communication manager and MPI daemon* work together to provide restorable MPI communications. Application programs communicate with MPI daemon though the JavaMPI binding API. The MPI daemon performs native MPI communication to provide efficient MPI communications. Communication manager is responsible for managing the channels. It receives migration information from migration manager, and is responsible for communicating and coordinating MPI daemons to restore MPI channels for during migration.

6. *Resource manager* is in mobile nodes only. It is used to aid for handling proactive migrations which are triggered when applications are requesting resources which are not available in the current node. When an application is trying to use a resource which is not available in the current device, a resource-related exception would be thrown. Examples of resource-related exception are `ClasssNotFoundException` and `OutOfMemoryException`. Information of these exceptions are redirected to resource manager. Resource manager would check if the resource is available in other nodes. When the resource is found, resource manager would then select an appropriate node, and contact migration manager to carry out a SOD migration, in which the requesting task would be migrated to the selected node where its execution would then be resumed. After finishing the task, results are sent back to the original node, and execution is resumed.

IV. Performance Evaluation

In this section, we evaluate the eXCloud performance in various aspects. We evaluate SOD migration with other computation migration mechanisms on a Xen virtual cluster, emulating a multi-instance cloud infrastructure. A virtual cluster is created on top of physical clusters with sufficient hardware processors given to all VM instances. All hosts are connected through Gigabit networks. In this performance evaluation, although live VM migrations are allowed, we focus on the evaluation of fine-grained task migration by SOD in different scenarios.

A. Environments

Two platforms, namely Platform A and Platform B, were used. Platform A targets for high performance computing, which are used in Section B. Platform B targets for mobile cloud computing with mobile devices connected, which are used in Section C-E. Platform A is a cluster of nodes interconnected by a Gigabit Ethernet network. Each node consists of 2 × Intel 6-Core X5650 Xeon 2.66GHz CPUs, 48GB 1333MHz DDR3 RAM and 7.2K rpm SATA II drives. The OS is Scientific Linux 5.5 x86_64. All nodes mount the same home directory on Network File System (NFS) to facilitate shared file access. The virtual machine manager (VMM) used is Xen 3.0.3-105.el5_5.2. 5 VMs are started in each host. Each VM is allocated with 4 logical processors and 512MB memory, running RedHat Enterprise Linux AS 4.6 (32-bit). The Java VM used is Oracle JDK 1.6.0_20-b02 (32-bit), operating in server mode.

Platform B is a cluster of nodes interconnected by a Gigabit Ethernet network and Wi-Fi (802.11g). Each node consists of 2 × Intel E5540 Quad-core Xeon 2.53GHz CPUs, 32GB 1066MHz DDR3 RAM and SAS/RAID-1 drives. The OS is Fedora 11 x86_64. All nodes mount the same home directory on Network File System (NFS) to facilitate 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) were installed on the iPhone. It was connected through Wi-Fi connection to the cluster network.

B. Overhead Analysis

Overhead analysis of SOD migration in a single level migration system had been taken in our previous work [10]. In this section, we evaluate SOD migration in a multi-level migration system. Several computation-intensive applications are used. Table 1 lists the applications, their problem sizes (n), maximum Java stack heights (h) and data size (D) of all local/static fields for reference. Fib and NQ have small data sizes but many stack frame operations. TSP and FFT has relatively fewer stack frame operations, but have much larger data size. This part of evaluation aims to characterize and compare the overheads of different migration mechanisms on top of VM environment. We ran each program listed in table 1 atop SODEE [10], JESSICA2 [6], and G-JavaMPI [11] in order to measure the overhead of stack segment migration, thread migration, process migration respectively. JESSICA2 performs Java thread migration in JIT mode; its mobility support is implemented at the JVM level by modifying the Kaffe JVM [12]. G-JavaMPI uses an earlier generation of JVM debugger interface to perform eager-copy process migration. SODEE and G-JavaMPI need an underlying JVM (JDK 1.6) to execute. Their executions were encapsulated in VM instances. Xen is used to support the guest OSes.

Table 2 shows the migration overhead of different approaches. The following metrics were measured:

i. *Exec. time w/o mig*–the total execution time under the system with no migration taken.

ii. *Exec. time w/ mig*–the total execution time under the system with 1 migration taken.

iii. *Migration overhead (MO)*–the time difference between the total execution time with migration and the execution time without migration.

Exec. time w/o mig of SODEE and G-JavaMPI are about the same. Both of them use the debugger interface. SODEE uses JVM TI while G-JavaMPI uses the older version JVMDI. The time for JESSICA2 is significantly larger, as it uses a rather old version Kaffe JVM in which JIT compiler is not as optimized as JDK. With migration, MO of SODEE is the smallest. It captures and restores the smallest amount of frames and data. MO of JESSICA2 is the second-smallest. State-capturing and restoring can be done efficiently as it is taken inside JVM. MO of G-JavaMPI is the longest, as it captures the whole process, including the whole heap, which is heavyweight.

TABLE 1. PROGRAM CHARACTERISTICS

App	Description	*n*	*h*	*D* (byte)
Fib	Calculate the *n*-th Fibonacci number recursively	46	46	< 10
NQ	Solve the *n*-queens problem recursively	14	16	< 10
TSP	Solve the traveling salesman problem of *n* cities	12	4	~ 2500
FFT	Compute an *n*-point 2D Fast Fourier Transform	256	4	> 64M

TABLE 2. MIGRATION OVERHEAD IN DIFFERENT SYSTEMS

App	SODEE on Xen (Stack seg. mig.)			JESSICA2 on Xen (Thread mig.)			G-JavaMPI on Xen (Process mig.)		
	Exec. time (sec)		MO (sec)	Exec. time (sec)		MO (sec)	Exec. time (sec)		MO (sec)
	w/ mig	w/o mig		w/ mig	w/o mig		w/ mig	w/o mig	
Fib	12.78	12.70	0.083	47.31	47.25	0.060	16.45	12.68	3.770
NQ	7.722	7.670	0.049	37.49	37.30	0.193	7.937	7.638	0.299
TSP	3.599	3.59	0.013	19.54	19.45	0.096	3.674	3.590	0.084
FFT	10.8	10.6	0.194	253.6	250.2	3.436	15.13	10.75	4.381

TABLE 3. MIGRATION LATENCY IN DIFFERENT SYSTEMS

App	SOD			G-JavaMPI			JESSICA2		
	Migration latency (ms)			Migration latency (ms)			Migration latency (ms)		
	Capture (ms)	*Transfer (ms)*	*Restore (ms)*	*Capture (ms)*	*Transfer (ms)*	*Restore (ms)*	*Capture (ms)*	*Transfer (ms)*	*Restore (ms)*
Fib	**6.31**			894.73			12.75		
	0.25	2.71	3.4	42.5	2.44	45	0.2	10.3	2.26
NQ	**6.8**			69.25			8.06		
	0.32	2.89	3.6	35.5	2.81	31	0.11	1.73	6.23
FFT	**19.39**			3659.56			59.08		
	0.35	14.9	4.1	742	2440	477	0.08	2.4	56.6
TSP	**8.08**			78.84			19.4		
	0.3	2.8	5	32	4.46	42	0.05	10.6	8.74

Table 3 shows the breakdown of migration latencies. The following metrics were measured:

i. *Capture time*–the interval between a migration request being received and the state data being ready to transfer.

ii. *Transfer time*–the time needed for the state data, upon being ready for transfer, to reach the destination.

iii. *Restore time*–the time when state data being available at the destination to the point of execution resumption.

iv. *Migration latency*–the time between receiving a migration request and getting the execution resumed at the destination. The value is equal to the sum of capture time, transfer time and restore time. The migration latency of SOD is the smallest. The migration latencies among different applications are about very close. Among the applications, in SOD, only the top stack frame is captured and restored. As heap data is not transferred during migration, the migration

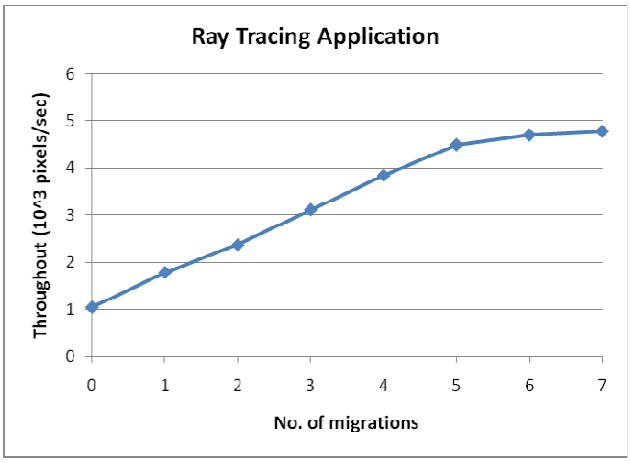

FIGURE 4. SCALING OUT FOR PARALLEL PROGRAM

latency is the smallest. The migration latencies of G-JavaMPI are the longest. During migration, eager-copy is used and the whole process data is captured and restored.

C. *Scaling out by SOD Migration*

In this experiment, we demonstrate how SOD with restorable MPI layer can be used to achieve task offloading among cloud nodes to have adaptive computing power according to the need. The program used is a parallel Java ray-tracing program using MPI to render 3D images. These images all have 360x275 pixels, and 182 objects. The rendering process uses anti-aliasing, tracing level 20, and 7 worker processes. The program renders the 3D images repeatedly and the average throughput is recorded. Initially, the program is executed with all worker processes executed in a single node. This simulates the situation that low rendering power is required at the beginning. Usually cloud resources are used adaptively when necessary. And then scale-out is taken by migrating rendering worker processes to other cloud nodes. This simulates the situation that a higher rendering power is required as time goes by, which can be obtained adaptively by scaling-out tasks to other nodes. In each migration, tasks are migrated to other idle cloud nodes by using SOD migration to allow rapid scale-up of the use of cloud computing resources. Fig. 4 shows the results of the experiment. Initially, as one node is used, the throughput is not high. With the increasing demands of rendering power, rending tasks are migrated to other available nodes by SOD. This experiment demonstrates the use of task mobility for on-demand scaling.

D. *Migration from mobile device to cloud nodes*

In this experiment, we evaluate the performance gain of using the migration techniques to migrate computation-intensive tasks from mobile devices to cluster nodes. In the experiment, instead of evaluating specific migration strategies, we focus on evaluating the achievable performance gain. We first execute the applications in a mobile device.

TABLE 4. MIGRATION FROM MOBILE DEVICE TO CLOUD'S NODE IN ACTIVE MIGRATION

	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**

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 returned back to the mobile devices where the application continues the execution. The result is shown in Table 4. It is shown that the performance gain with migration can be 3 to 56 times. The migration latency ranges from around 250ms to 400ms. The capture time and transfer time are much larger than the time in Section B. As the programs are originally executed in the mobile device, state-capturing refers to the capturing of state of the program in the mobile device. As the device's speed is much slower than the cloud node, the capturing time is larger. Besides this, in mobile device, state-capturing is taken at application level, and Java object serialization is used. The transfer time is also much larger as Wi-Fi instead of gigabit network is used.

E. *Migration from cloud node to mobile devices*

In this experiment, we demonstrate how SOD can be used to use the resources in mobile devices in a feasible manner without significant memory overhead. In the experiment, a web server program is executed in a cloud node, which returns file information and files to the clients. In the setting, there are 5 directories, each holding 100 image files. There is another empty directory named "ip4". When the server program tries to read from this directory, a migration request is triggered. The task is migrated to an iPhone, where the directory information is actually read. After the task completes, the execution returns to the server program with the information of the image files found. The server program then returns the aggregated results in HTML format to the clients. Here are testing results: The memory footprint of the process in the cloud node is 31,907,096 bytes (about 30MB) while the memory footprint of the process in the iPhone is 852,544 bytes. That means, when compared with process migration, SOD avoids a significant amount of memory consumption (up to 97%). Besides this, in the whole execution, there are active network connections between the server program and the clients. With the use of SOD, the need of migrating these native states can be avoided.

V. Related Work

CloneCloud [13] is a system that seamlessly offloads part of the execution of mobile applications from mobile devices to a computational cloud. During migration, in the cloud nodes, VM is used to simulate an execution environment identical to the mobile devices. The partition of migration is determined by offline static and dynamic profiling. Possible migration points are determined, and the actual migration point is set based on profiling before execution. In our approach, there is no need to clone an identical environment. It is more adaptive to the new execution environment. Besides this, migration in our system is determined dynamically at runtime.

Cloudlet [14] is a transiently customized computing infrastructure where mobile devices leverage resources of a nearby cloudlet by VM migration. Their migration approach is rather coarse-grained while ours can migrate tasks at much finer granularity within very short time.

MAUI [15] is platform which minimizes power consumption of mobile devices by offloading tasks to cloud nodes. Method shipping with related heap objects is used. Application codes are analyzed, and potential migration points are annotated to allow remote execution. The migration decisions are based on the amount of runtime resources. In our system, granularity of migration is much finer as it allows migration to take place in the middle of a method execution. Besides this, apart from resources, migration is goal-driven in various aspects, such as constraints and locality.

VI. Conclusion

In this paper, we have introduced eXCloud as a middleware system to provide seamless, multi-level task mobility support to allow migration at different granularity, ranging from coarse to fine. While virtual machines and live migrations already provide resource isolation and execution mobility at the infrastructure level, our stack-on-demand (SOD) execution model advances the state-of-the-art by allowing lightweight partial state migration to facilitate fine-grained task migration among cloud nodes and mobile nodes. Computation can now migrate quickly among nodes to allow better resource utilization .

Acknowledgment

This work is supported by Hong Kong RGC grant HKU 7179/09E 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

[2] "Report: Mobile Cloud Computing A $5 Billion Opportunity" Internet: http://www.crn.com/mobile/222300633

[3] D. S. Milojicic, F. Doublis, Y. Paindaveine, R. Wheller, and S. Zhou. “Process Migraiton,” ACM Computing, 2000

[4] "OpenSSI project", Internet: http://openssi.org/cgi-bin/view?page=openssi.html

[5] 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

[6] W. Zhu, W. Fang, C. L. Wang, and F. C.M. Lau. “A New Transparent Java Thread Migration System Using Just-in-Time Recompilation,” *In Proc. of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, pp. 766-771, MIT Cambridge, MA, USA, November 9-11, 2004.

[7] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[8] 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)*, pages 110–117, Hong Kong, China.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. “Live migration of virtual machines,” *In Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation* - Volume 2, NSDI 2005, Berkeley, CA, USA, pp. 273-286. USENIX Association.

[10] R. Ma, K. T. Lam, C. L. Wang, and C. Zhang. “A stack-on-demand execution model for elastic computing,” *In Proc. of the 39th International Conference on Parallel Processing (ICPP 2010)*, pp. 208-217.

[11] L. Chen, T. C. Ma, C. L. Wang, F. C. M. Lau, and S. P. Li. “G-JavaMPI: A Grid Middleware for Transparent MPI Task Migration,” Chapter 20, *Engineering the Grid: Status and Perspective*, Nova Science Publisher, Jan 2006

[12] "Kaffe JVM" Internet: http://www.kaffe.org

[13] 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*

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

[15] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Stefan, R. Chandra, and P. Bahl. "MAUI: making smartphones last longer with code offload," *In Proc. of MobiSys 2010*

[16] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schioberg. “Live wide-area migration of virtual machines including local persistent state,” *In Proc. of the 3rd International Conference on Virtual Execution Environments, VEE 2007*, New York, NY, USA, pp. 169-179. ACM.

[17] M. Nelson, B.-H. Lim, and G. Hutchins. “Fast transparent migration for virtual machines,” *In Proc. of the USENIX Annual Technical Conference (ATEC 2005)*, Berkeley, CA, USA, pp. 25-25. USENIX Association.

[18] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y.Wang. “Seamless live migration of virtual machines over the MAN/WAN,” Future Gener. Comput. Syst. 22, 901-907.

[19] R. Ho, C. L. Wang, and F. Lau. “Lightweight Process Migration and Memory Prefetching on openMosix,” *In Proc. of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS2008)*, Miami, Florida USA, April 14-18, 2008

[20] 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