Heterogeneous Computation Migration on LLVM

Tyng-Yeu Liang(✉) and Yu-Jie Lin

Department of Electrical Engineering, National Kaohsiung University of Applied Sciences, Kaohsiung, Taiwan
lty@mail.ee.kuas.edu.tw, jaredlin@hpds.ee.kuas.edu.tw

Abstract. In past decades, the development of mobile applications was limited due to lack of enough computational power. To resolve this problem, the framework of mobile cloud computing (MCC) was proposed for offloading the massive computation tasks of mobile applications onto cloud centers for execution. However, the computational power of mobile devices recently has received a great promotion, and the bandwidth and reliability of wireless networks has been significantly improved. These development advances make it practical for mobile devices to share the computational tasks of cloud centers. In other words, the direction of resource supply chain can be from clouds to mobile devices but also from mobile devices to clouds. This is useful for integrating the computational power of mobile devices and cloud resources to serve mobile or cloud users. To achieve this goal, this paper is aimed at the development of an efficient scheme of computation migration based on LLVM for addressing the problem of resource heterogeneity and dynamicity in MCC. With the support of the proposed scheme, user programs can dynamically move between mobile devices and cloud servers for the load balance, QoS and reliability of MCC.

Keywords: Mobile cloud computing · Mobile devices · Computation migration · LLVM · Reliability

1 Introduction

Nowadays, mobile devices such as smart phones and tablets have replaced PCs and laptops to become the main equipment for users to handle their daily staffs including communication, information searching, shopping, working, learning and gaming under the support of an enormous number of devise APPs. However, because of lack of enough computational capability and electrical power, the past application development of mobile devices was not effectively extended to the area of high performance computing. To address this issue, the framework of mobile cloud computing was proposed to support mobile devices for performing high performance computing applications.

The basic idea of MCC is to offload the massive computation tasks of applications from mobile devices to cloud servers for execution. This solution indeed successfully releases the application development of mobile devices from the problem of lacking enough computational power. Many mobile applications such as M-learning [1] and M-healthcare [2] have been developed based on the MCC framework. In these applications, mobile devices usually are resource consumers while clouds play resource

X. Huang et al. (Eds.): GPC 2016, LNCS 9663, pp. 146–162, 2016.
DOI: 10.1007/978-3-319-39077-2_10

providers. Nonetheless, this relationship of resource demand and supply has become not always one directional because the computational capability and electrical power of mobile devices have been greatly promoted.

Generally speaking, most of modern mobile devices have 1–4 GB RAM, quad core ARM CPU, and many-core GPU such as Adreno, PowerVR, and NVIDIA Tegra. They become comparable with common PCs in computational capability, and can continuously work for more than a dozen hours by the support of long-termed battery and mobile power supply. On the other hand, the bandwidth of wireless network technology has reached to hundreds Mbps through LTE. Consequently, many mobile APPs has been proposed for resolving research problems by using the computational power of mobile devices. For example, BOINC [3] allows users to donate the spare time of their mobile devices for different science research projects. Addi [4] and Octave [5] allows users to perform mathematical computation and plotting by means of MATLAB instructions and scripts in mobile devices. CCTools [6] and C4droid [7] supports a C/C++ programming for users to develop and execute scientific-computing applications on mobile devices. This development trend shows that mobile devices have drawn high attention from researchers. It also implies that mobile devices can play not only resource consumers but also providers in the architecture of MCC. Since the number of mobile devices has reached to billions, the computation power hidden in mobile devices is amazing and waiting for exploration. For clouds, they can offload tasks to mobile devices for reducing load pressure in rush hours. For mobile devices, they can share their own resources with others, and thereby do not always rely on the assistance of clouds any more.

To achieve this goal, the first challenge is to overcome the problem of resource heterogeneity in task migration. For this issue, many past studies [8, 9] were focused on JVM since Java is frequently used for developing portable applications, and Java bytecodes are executable on heterogeneous resources through JVM. However, Java programs cannot directly access system information and hardware through JVM. Consequently, Java is not as good as C/C++ in the performance of I/O. Although Java Native Interface (JNI) is effective for resolving this problem, it increases the programming complexity and execution cost of user applications. On the other hand, most of scientific and high performance computing (HPC) applications are developed by C/C++ instead of Java because of execution performance consideration. Even they are developed by Java. They are not always executable on any mobile devices because of different Java or JVM versions.

By contrast, Low Level Virtual Machine (LLVM) [10] has not the above problems existing in JVM. LLVM is a lightweight virtual machine. It can be divided into front-end compiler and backend executor. The front-end compiler, i.e., Clang is responsible to translate user programs into LLVM IRs while the backend executor converts and LLVM IRs into optimized native codes at runtime according to the architecture of target processor first and then executes the native codes by the target processor by means of MCJIT. Because the native codes generated by LLVM are easily linked with external libraries and able to directly access hardware, LLVM does not degrade the performance of CUDA and OpenCL API while JVM does. This advantage is very important for cloud and mobile computing because more and more clouds and mobile devices has supported GPU for HPC and big data applications.

As previously discussed, this paper is aimed at design and implementation of an efficient heterogeneous computation migration (simply called HCM) scheme based on LLVM. With the support of this scheme, the programs of mobile and cloud users can roam around the MCC architecture according to network bandwidth, electrical power, or computational capability for obtaining a good execution performance and fault tolerance. We have evaluated the efficiency of proposed scheme in this paper. Our experimental results have shown that the cost of proposed scheme is acceptable or negligible for the performance of tested applications.

The rest of this paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 describe the framework and implementation of HCM, respectively. Section 5 evaluates and discusses the efficiency of HCM. Section 6 gives a number of conclusions for this paper and our future work.

2 Related Work

In 1980s, process migration [11–13] was a hot research topic for addressing the issues of load balance and reliability of distributed systems. The main advantage of process migration is transparent for user applications since it is implemented at the system level [14, 15]. However, this approach suffers from the problem of residual dependencies that is a migrated process relies on a host for data structures or functional supports such as opened file descriptors and shared memory segments after this process moves away from the host. Consequently, it is difficultly applied to real applications.

By contrast, virtual machine (VM) migration [16, 17] can avoid the problems of process migration because it moves an entire OS, and all of its applications as one unit from one physical machine to another. It is frequently applied for the resource and energy management of cloud computing since VM software such as Xen, KVM and VMware is the main solution used for resource virtualization and sharing in data centers. Nonetheless, the time cost of virtual migration is so long as to degrade the performance of applications especially when the number of memory pages copied to the new VM is large or network speed is extremely slow. For this problem, live VM migration [18] is an effective solution for reducing the negative impact of migration cost because it allows applications to continue running during migration process.

The process of live VM migration can be classified into three phases: push, stop-and-copy and pull [19]. In the push phase, the original virtual machine continues to run on source node while simultaneously copies the recently used pages to the new virtual machine on destination node. For data consistency, the pages modified during migration process must be sent to the new virtual machine. In the stop-and-copy phase, the original virtual machine stops to copy the remaining dirty pages to the new virtual machine, and then the new virtual machine starts. In the pull phase, the new virtual machine executes and fetches necessary pages from the old one when it accesses the pages and the pages are not present in local memory. Most of proposed VM migration algorithms consist of one or two of these phases to reduce down time and total migration time. For example, the Pre-copy algorithm [20] mixes the push phase and the stop-and-copy phase. It adds a threshold condition to limit the time cost of sending data pages and estimates the cost

of data consistency maintenance and the update rate of dirty pages to decide when to stop the original virtual machine for sending all the dirty pages to the new virtual machine. It resolves the problem of frequent page re-transmission in the push phase, and reduces down time and total migration time. In contrast, the Post-copy algorithm [21] mixes the pull phase and the stop-and-copy phase in order to delay the copy of memory pages until they are accessed. Compared to the Pre-copy algorithm, the advantage of the Post-copy algorithm is to avoid page re-transmission. Each page is transmitted at most once while the performance of user applications suffers from a large amount of page faults as the new virtual machine frequently is faulty on accessing the absent memory pages.

On the other hand, a stack-on-demand solution based on JVM [22] was proposed for offloading massive computation tasks from mobile devices to clouds to speed up the completion of these tasks. This solution only sends the top stack frame and the currently called object method to clouds for execution. Consequently, it can reduce the communication cost of task migration to the minimal. However, it requires the client keeps connection with the server for exchanging the parameter data and execution result of the migrated object method.

Apparently, VM migration is not suitable for task migration between mobile devices and clouds because the migration cost is too expensive to be compensated in wireless networks. The worse is the memory space of mobile devices is not big enough for storing the image of VM. Although the Pre-copy and Post-copy algorithms can effectively reduce downtime or/and total migration time, they are not realistic for task migration between mobile devices and clouds because the operating system of mobile devices does not allow users to define the page-fault handler. On the other hand, the requirement of the stack-on-demand approach is difficult to be satisfied because the connection between client and server is easily broken due to the movement of mobile devices in wireless networks. In addition, it produces a lot of data communication when user applications are implemented by iteratively calling object methods.

By contrast, the proposed scheme is a lightweight process migration while it is independent to hardware or operating system because it is implemented based on LLVM. Although pure stop-and-copy migration increases down time as well as the number of copied memory pages, it is easy to be implemented without data-consistency maintenance. In addition, the number of memory pages transmitted for task migration on LLVM is much less than that of VM migration. Therefore, the proposed migration scheme adopts the pure stop-and-copy algorithm. For reducing migration cost, the proposed scheme compresses transferred data to minimize the amount of data transferred over wireless networks.

3 Framework

The framework of HCM is as shown in Fig. 1. This framework is mainly composed of sensor, server, and broker. The HCM sensor aims at monitoring resources, cyclically reporting resource states such as load or remaining electrical-power capability to the HCM broker, and sending a migration semaphore to the HCM server when it finds the

local resource states has satisfied the migration condition set by resource owner or administrator. The HCM server is mainly used to execute user programs, and backup or restore the runtime contexts of user programs on LLVM for task migration. Since mobile devices and cloud servers are regarded as the same in the framework of HCM, both of these two different resources can play source or destination nodes in task migration. The registered information mainly consists of their network location, and condition of accepting migrated tasks. On the other hand, the HCM broker is responsible to allocate resources for task migration and play an agent for mobile devices or cloud servers to migrate their tasks to new resources for execution, and cache the execution results of migrated tasks.

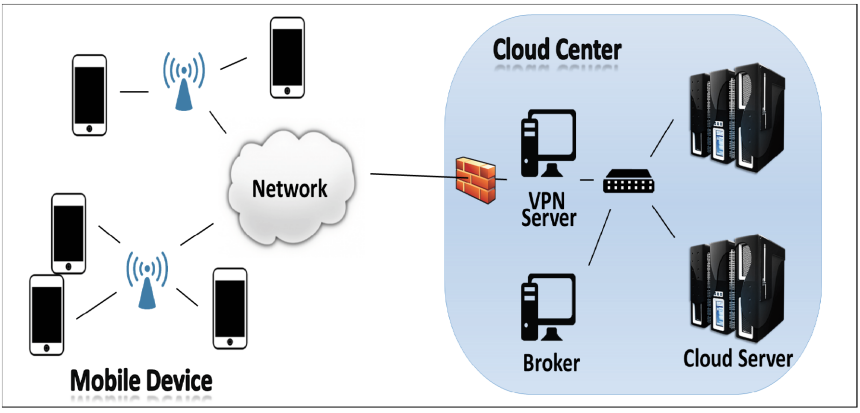

Fig. 1. Framework of HCM

In the HCM framework, any user program is compiled into an executable file of LLVM IRs. When the program is executed, the executable file of this program is loaded and executed by LLVM by means of MCJIT. All the HCM servers and the HCM broker are connected with a virtual private network for secure and communication across different network domains. The direction of task migration can be mobile device to cloud (M2C), cloud to mobile device (C2M) and mobile device to mobile device (M2M) no matter where source and destination node are located in networks. Because HCM currently supports only batch but interactive tasks, the I/O of user programs is redirected into file accesses. The user scenario and the process of task migration in the HCM framework are described as follows.

As shown in Fig. 2, when a HCM server is initialized at a mobile device or cloud server, it joins into the virtual private network of HCM first, and then collects and registers local resource information to the HCM broker. After the resource registration is finished, the HCM server waits for executing local tasks or remote tasks migrated from other cloud servers or mobile devices. On the other hand, it creates a background thread to perform the HCM sensor in order to cyclically detect local load state or electrical power capability. If the HCM sensor finds the resource state satisfies the migration

condition set by resource owner or administrator, it will set a migration flag as true; otherwise, it will set the flag as false. When the HCM server accepts a task of executing a program submitted by the local node, it automatically modifies the executable of the program by inserting a number of checkpoints into the entry and departure points of basic blocks or function calls in the program first, and then dispatches the user program onto LLVM for execution. When LLVM executes the user program, it checks the migration flag at the checkpoints inserted by the HCM server. If it finds the migration flag is set as true, it will immediately invoke the HCM server to retrieve and backup the context of the execution stack and memory segments including data and heap of the program from local memory into an image file. Then, it sends the image, executable and I/O files of the user program to the HCM broker for task migration. After receiving a migration request and the files of the migrated program, the HCM broker selects a new execution node from the pool of registered resources according to the task-acceptance conditions and the current states of resources. Next, it relays the files of the migrated program to the HCM server of the new execution node, and then the new HCM server rebuilds the context of execution stack and memory segments in the local memory and resumes the execution of the user program on LLVM. After the execution of the migrated program is finished, the new HCM server sends the output file of the user program to the HCM broker. Finally, the HCM broker caches the output file in the local file system until the old HCM server on the original execution node fetches the output file of the program.

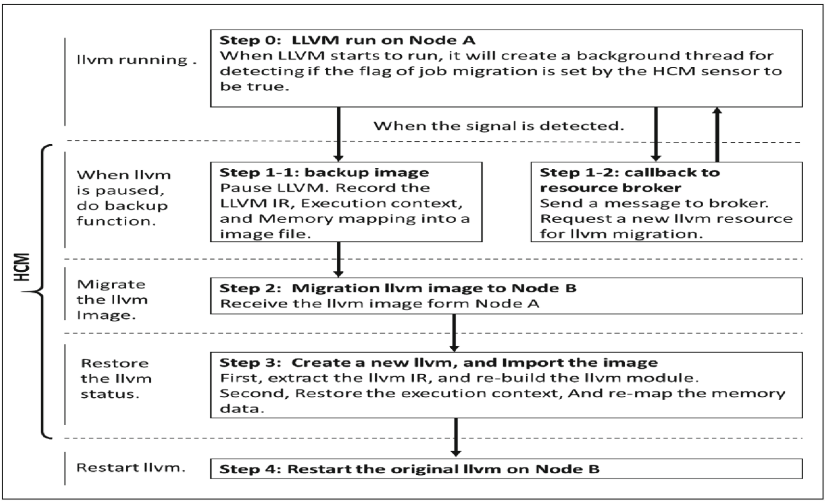

Fig. 2. Flow of task migration in the HCM framework

It is worthy to say that this study adds three command parameters into LLVM LLI for the HCM scheme as shown in Table 1. "–remote-mode" is used to set the local node as HCM server. The "–broker-ip" and "–broker-port" parameters is used to set up the network location of HCM broker for the HCM server. The HCM broker must have a

public IP for connection with the HCM servers and sensors on mobile devices and cloud servers.

Table 1. Commands of HCM

Command parameter	Description
–remote-mode=?	Maybe CLIENT or SERVER
–broker-ip=?	HCM broker public IP
–broker-port=?	HCM broker binding port

4 Implementation

The main jobs of implementing the proposed scheme consists of resource monitoring, program recompilation and standard I/O redirection. The details of these things are described as follows.

4.1 Resource Monitoring

Currently, the proposed scheme allows resource owners to set the conditions of performing task migration according to remaining power capability, CPU/Memory usages. When the percentage of remaining power capability or CPU/Memory usages is below or above a threshold, the HCM sensor will set the migration flag as TRUE, and then the process of task migration will be performed. To simplify our work, we made use of the Linux system primitives and files such as mpstat and /proc/meminfo to obtain the usages of current CPU and memory of user programs. On the other hand, we exploited the acpi function to get the remaining power capability of mobile devices. Since the information of these resource states is represented by means of strings, it is easy to implement the HCM sensor by Shell script. However, Android does not support bash, mpstat, acpi and the string tools such as head, tail, awk and wc. To overcome this problem, we ported these functions and tools with Android SDK for the implementation of resource monitoring.

4.2 Program Recompilation

In this paper, we developed a LLVM-IR re-compiler for HCM to automatically rewrite the LLVM executable files of user programs by inserting checkpoints, backup and restore functions. This re-compiler makes use of the toolkit of LLVM to operate the syntax trees of user programs. The process of executing a user program on LLVM is divided into five phases as shown in Fig. 3. The first phase is to load the executable of the user program, which is compiled by Clang into LLVM IRs. The second phase is to create a module, and then pass the LLVM IRs of the user program to the module. The third is to translate the LLVM IRs of the user programs into native codes based on the architecture of execution processor. The last two phases are to build a MCJIT execution engine and execute the native code with the MCJIT engine. Accordingly, the proposed

LLVM-IR re-compiler is used to rewrite the executable of the user program between the second phase and the third one.

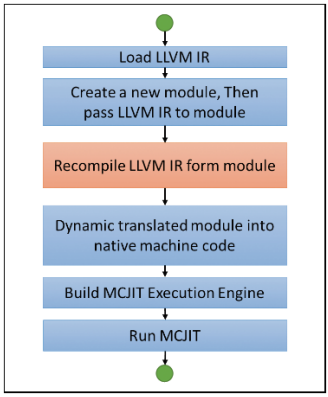

Fig. 3. Modified execution flow of user programs on LLVM

The recompilation flow of a user program is as shown in Fig. 4. First, the re-compiler register the functions and symbols of HCM to the LLVM parser in order for making these functions and symbols linkable with user programs. Second, it partitions the user program into a number of basic blocks, which are insert by checkpoints. Third, it collects the all of local variables including static or dynamic in each function, and inserts a log function for storing these local variable in a stack. Forth, it scans and stores all the function pointers and global variables of the user program into a symbol table. Finally, it inserts a backup and store function for basic blocks to back up and restore the context of the program in task migration.

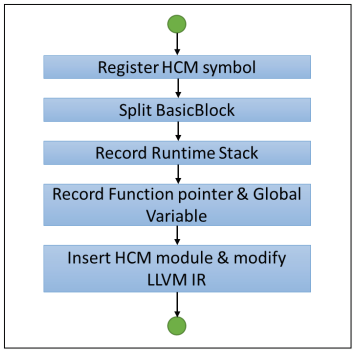

Fig. 4. Flow of program recompilation

Although the proposed re-compiler is to rewrite the LLVM IRs of user programs, here we use an example program presented by means of C but LLVM IRs in order to make it easy to understand the process of program recompilation, as shown in Fig. 5.

The first step of program recompilation is to insert the functions of storing the information of the memory image of the user program executed on LLVM. For live migration on LLVM, it is necessary to back up the content of heap, global variables, function points and the runtime stacks of user programs. For the backup of heap, the HCM server replaces the dynamic allocation functions such as malloc(), calloc() and realloc() with the HCM functions such as HCM_malloc(), HCM_calloc() and HCM_realloc() to keep track of the sizes and start addresses of memory spaces dynamically allocated. For the backup of global variables and function pointers, the HCM server makes use of the function such as RegGlobalVar() or RegFuncPoint() to track the mapping between the symbol name of global variables or functions and their memory addresses. Because the runtime stack of a user program is inside LLVM, the HCM server cannot directly control the location of local variables or parameters in the runtime stack. To overcome this problem, the HCM server maintain a shadow stack to track the names and memory of local variables in each user function by inserting PushCurStack() and PopCurStack(). It is worth noting that LLVM declares a local variable for each parameter of a user function, and replaces the parameter in any statement with the declared local variable, as shown in the ALLOC label. In general cases, LLVM cannot keep the memory addresses of global and local variables in the destination node as same as in the source node of task migration. Therefore, it is necessary to copy out the data of global and local variables from the memory of the source node before migration and copy the data values into the memory of the destination node after migration based on the mapping of names and memory addresses of variables.

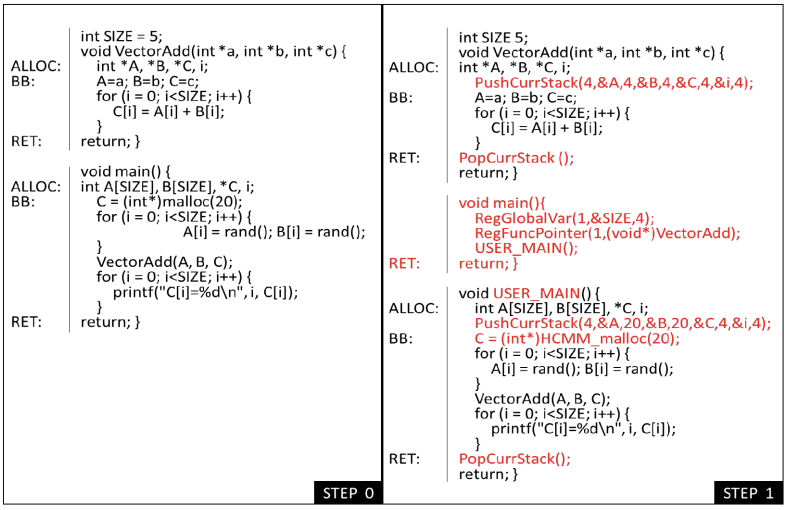

Fig. 5. Step 1 of program recompilation

The second step of program recompilation is to insert checkpoints in user programs for checking if the migration flag, i.e., MigrationFlag, is set by the HCM sensor to be “BAC”.

If it is true, LLVM jumps to invoke the backup routines added in user functions by looking back upon the calling sequence of user functions, as shown in Fig. 6. For reducing backup cost, the HCM server currently insert checkpoints only in front and rear of function calls, and the most outer layer of nest loops in any function. If a user program has no function call and loops, the HCM server add a checkpoint for each basic block in the user program. On the other hand, the backup routine is aimed at recording which pointer variable must be redirected to a new memory address after migration process, and storing the stack frame of each user function in the calling sequence into the image file. For example, assume the execution flow of the example program arrives at the CP1 label of VectorAdd(), and the migration flag is set as "BAC". LLVM jumps to the BACKUP label to call BackupCurr-Stack() for storing the stack frame of the VectorAdd() first. Next, it returns to the CP3 label of USER_MAIN(), and then jumps to the BACKUP label for calling BackupCurrStack() for storing the stack frame of USER_MAIN(). Finally, it returns to main() and then jumps to the BACKUP label for invoking BackCurrStack() to store the content of heap and global variables into the image file.

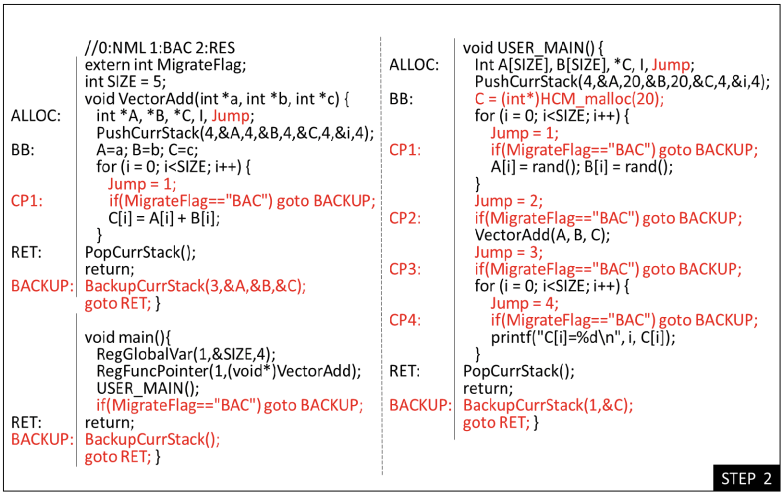

Fig. 6. Step 2 of program recompilation

It is worth noting there is a local variable called Jump in VectorAdd() and USER_MAIN(). This variable is used to control the flow of recovery process later. When LLVM performs BackupCurrStack(), it also stores the value of the Jump variable in the image file. For our example, the values of the Jump variables in VectorAdd() and USER_MAIN() are 1 and 2, respectively.

The third step of program recompilation is to insert restore routines for recovering the context of the migrated program on destination node, and resuming the execution of the program from the checkpoint starting the migrated process. As shown in Fig. 7, the HCM server adds a RestoreCurrStack() in each user function for rebuilding the stack frame of the user function. The Jump parameter of RestoreCurrStack() is used to guide LLVM jump to which statement in the user function after finishing RestoreCurrStack().

After receiving the image file of the migrated program, the HCM server on destination node sets MigrateFlag as "RES", and then recovers the content of heap, global variable and stack in the local memory of the destination node from the image file by invoking each RestoreCurrStack() based on the calling sequence of the migrated program.

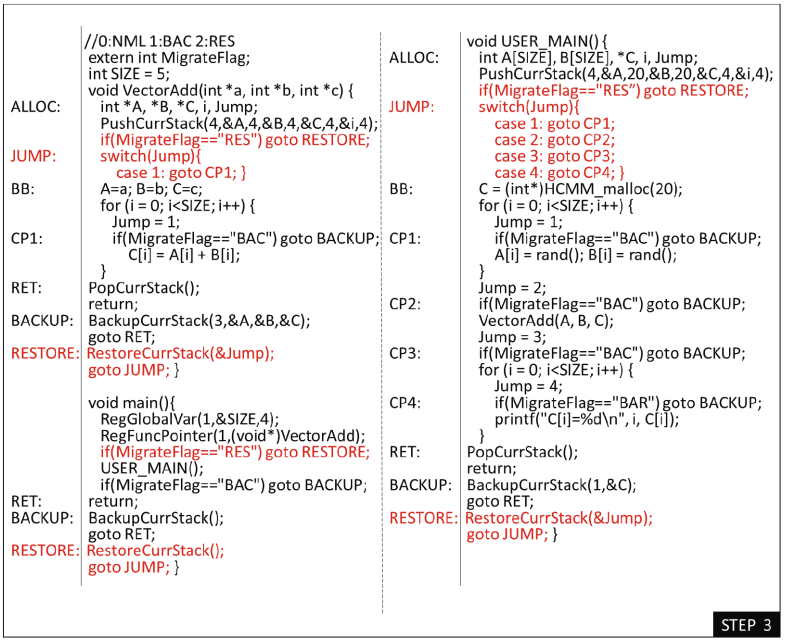

Fig. 7. Step 3 of program recompilation

For the previous assumed case, the recovery process is described as follows. First, LLVM re-executes the program from main() while it only invokes RegGlobalVar(), RegFunctionPointer() and RestoreCurrStack() to obtain the new memory addresses of global variables and VectorAdd(), and to restore the content of heap and global variables from the image file to the memory of the destination node, respectively. Second, LLVM jumps to USER_MAIN() while it only calls PushCurrStack() and RestoreCurrStack() to get the new memory locations of local variables in this function, and to copy the values of the local variables from the image file to the new memory locations of these variables, respectively.

Finally, LLVM jumps to the CP2 label of invoking VectorAdd(). After it jumps to VectorAdd(), it calls PushCurrStack() and RestoreCurremStack() to do the same things as done in the USER_MAIN(), and then jumps to the CP1 label to continue the execution of the for loop.

4.3 Redirection of Standard I/O

In the HCM framework, the standard I/O of user programs is redirected to files. Because of security consideration, we developed an adapter library for HCM to direct any file functions called by user programs into a dedicated file directory in order for preventing the programs from accessing other file directories. Whenever user programs access files out of this dedicated file directory, HCM will catch the file accesses and terminate the execution of the programs right away. In addition to user files, HCM automatically open a file in the same directory for the standard output such of each user program before it starts to run. However, users must direct the standard input of their programs to the files specified by themselves. Before a user program is moved from one node to another, the descriptors of the files opened by the user program will be logged, and the opened files will be automatically closed. Then, the HCM server will send the files with the image file of the user program to the destination node. After receiving the files, the new HCM server will write these files into disks with the same file path, and will automatically open these files, and seek the access headers of the files to the logged positions according to the file descriptors.

4.4 Optimization

Although the cost of task migration on LLVM is much less than that of VM migration, in order to further reduce migration cost, the proposed scheme compress any files transmitted from source nodes to destination nodes for task migration, and then decompress the files in destination nodes for rebuilding the context of user programs. Since the computational power of mobile devices recently has been greatly improved, the benefit from saving communication cost usually is larger than the loss from compressing and decompressing file data because the bandwidth of wireless networks is not stable especially in outdoors.

5 Performance Evaluation

We have primarily evaluated the performance of proposed scheme in this paper. We implemented an application called Matrix Multiplication for this performance evaluation. For building our test bed, we used two Xiaomi MiPAD to play mobile client and server, and two PCs to play broker and cloud server, respectively. The specification of these mobile devices and PCs is depicted in Table 2. On the other hand, the broker and the cloud server are connected with a 100 Mbps Ethernet. The gateway between PCs and mobile devices was ZyXEL USG20W, i.e., an access point supporting 802.11n. The mobile devices communicated with the PCs through Wi-Fi or 4G.

We have finished three experiments in the test bed. The first experiment is to estimate the cost of checkpoints in HCM. The second and the third ones are to measure the costs of task migration under three different paths including mobile device to cloud server (M2C), cloud server to mobile device (C2M) and mobile device to mobile device (M2M) through Wi-Fi and 4G, respectively. In fact, the task migration between mobile device

and cloud server is delegated to the HCM broker (denoted as B). Therefore, the three different paths of task migration physically are M-B-C, C-B-M and M-B-M.

Table 2. Device specification

Device	CPU	RAM	Network	OS	Amount
Xiaomi Mipad	NVIDIA Tergra K1	2G-LPDDR3	Wi-Fi 802.11n	Android 4.4.2	2
X86 Server	AMD A10-5800 k	8G-DDR3	Ethernet 100 Mbps	Ubuntu 14.04	1
Broker	Intel i5-760	4G-DDR3	Ethernet 100 Mbps	Ubuntu 14.04	1

5.1 Checkpoint Cost of HCM

In this experiment, we aimed at evaluating the impact of checkpoint cost on the execution performance of the test application executed on the mobile client. To achieve this goal, we recompiled the LLVM executable of the test application, and evaluated the execution performance of the test applications with and without program recompilation. In addition, we also recompiled the test program by inserting a checkpoint in each LLVM IR to compare this exhaustive checkpoint way with the lazy checkpoint way of HCM in terms of time cost. Our experimental result is depicted in Fig. 8.

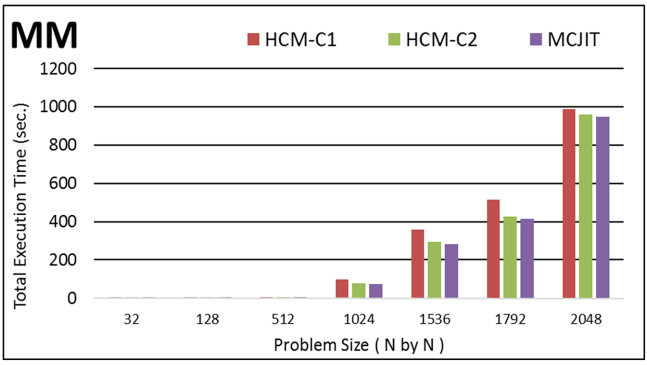

Fig. 8. Checkpoint cost of the test applications

In this figure, HCM-C1 and HCM-C2 represent the exhaustive checkpoint way and the lazy checkpoint way of HCM, respectively. In contrast, MCJIT denotes no checkpoint in the test application. The performance comparison between the HCM-C2 case and the MCJIT case shows that the checkpoint cost of HCM is negligible for the execution performance of the test application. Conversely, the cost of exhaustive checkpoint way is significant for the execution performance of the test application. The applications must spent extra 20~33 % execution time for checkpoints. Although performing checkpoint for each instruction is useful to reduce the delay of activating the mechanism of task migration, the overhead apparently is too expensive to be tolerated.

5.2 Cost of Task Migration Through Wi-Fi

This experiment is to evaluate the efficiency of task migration under Wi-Fi. We performed task migration for the MM application in three different migration paths (i.e., M2C, C2M and M2M) with or without compressing transferred data, and then evaluated and the cost of task migration. In addition, we estimated the transmission rate of these three paths in advance for different data sizes as depicted in Fig. 9. It shows that the transmission rate increases as well as the size of each transmitted data. The C-B-M path has the largest transmission rate while the M-B-M path has the smallest one no matter what size of transmitted data.

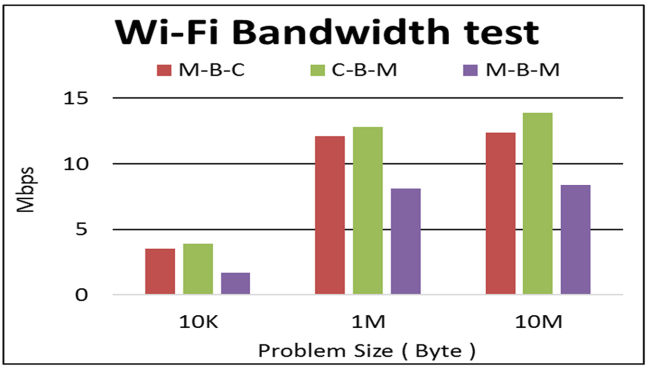

Fig. 9. Transmission rates of different migration paths.

Figure 10 is the cost of task migration for the MM application under three different migration paths. The breakdown of task migration cost includes backup, transmit and restore. The backup and restore costs are the time used to save and restore memory data, program codes and execution context to/from the image file of the test application, respectively. In contrast, the transmit cost is the time spent on transmitting the image,

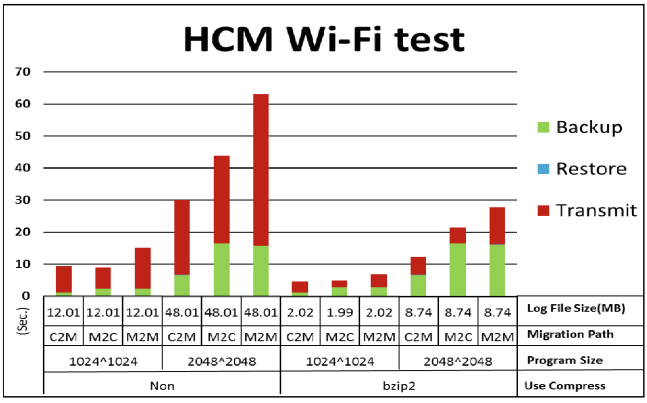

Fig. 10. Migration cost of MM through Wi-Fi

executable, I/O files from the source node of task migration to the destination node. The total cost of task migration is increased with the size of the files no matter what migration path is. Both of the transmit cost and the backup cost are obvious while the restore cost is negligible. In addition, the transmit cost is effectively reduced after applying data compression. This result makes it confirmed that compressing the image files of migrated programs is useful and necessary for reducing the cost of live migration especially when the migration path is M2M because the transmission rate of the M2M path is much smaller than those of the other two paths.

5.3 Cost of Task Migration Through 4G

In this experiment, we aimed at evaluating the cost of task migration through 4G. We also estimated the transmission rates of different migration paths through 4G in advance as shown in Fig. 11. Basically, the estimated result physically is only from 8~15.8 Mbps which is as same as shown in Fig. 10 although the maximal transmission rate of 4G is announced to be 1 Gbps. That is because the physical transmission rate of 4G is affected by many environmental factors such as signal strength, sender/receiver location, and outdoor/indoor.

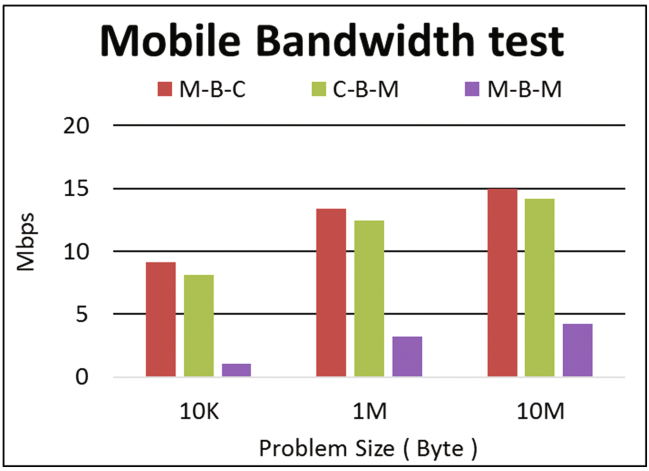

Fig. 11. Transmission rates of different migration paths.

The result of this experiment is shown in Fig. 12. Most of the situations observed from the previous experiment also occurs in this experiment. A different situation is that the transmission rate of the C2M path is less than that of the M2C path under 4G. Consequently, the transmit cost the C2M path is more than that of the M2C path. On the other hand, the impact of compressing transfer data under 4G becomes more obvious than that under Wi-Fi.

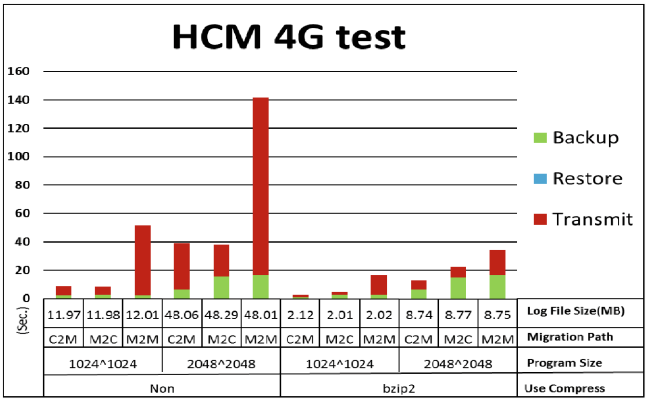

Fig. 12. Migration cost of MM through 4G

6 Conclusions and Future Work

In this paper, we have successfully developed an efficient heterogeneous computation migration scheme for mobile cloud computing. With the proposed scheme, user programs can dynamically move among mobile devices and cloud servers while they can obtains an execution performance as good as native code on any execution node. Consequently, the proposed scheme is effective for integrating the computational power of mobile and cloud resources to serve mobile and cloud users as well as possible. In this paper, we only focused on the development of live migration mechanism on LLVM. We will develop an advance scheduling algorithm based on the proposed migration scheme for mobile cloud computing.

Acknowledgment. This work is supported by Ministry of Science and Technology of the Republic of China under the project number: MOST 103-2221-E-151-044.

References

1. Seppälä, P., Alamäki, H.: Mobile learning in teacher training. J. Comput. Assist. Learn. **19**(3), 330–335 (2003)
2. Varshney, U.: Pervasive healthcare and wireless health monitoring. Mobile Netw. Appl. **12**(2–3), 113–127 (2007)
3. Anderson, D.P.: BOINC: a system for public-resource computing and storage. In: Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10 (2004)
4. Champion, C.: Addi (2015). https://play.google.com/store/apps/details?id=com.addi
5. Quarteroni, A., Saleri, F., Gervasio, P.: Scientific Computing with MATLAB and Octave. Springer, Heidelberg (2010)
6. Chukov, A.: CCTools (2015). https://play.google.com/store/apps/details?id=com.pdaxrom.cctools
7. n0n3m4: C4droid - C/C++ compiler & IDE (2015). https://play.google.com/store/apps/details?id=com.n0n3m4.droidc

8. Zhu, W., Wang, C.-L., Lau, F.: JESSICA2: a distributed Java virtual machine with transparent thread migration support. In: Cluster Computing, pp. 381–388 (2002)
9. Bouchenak, S., Hagimont, D.: Zero overhead Java thread migration. Technical report 0261 (2002)
10. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: Code Generation and Optimization, pp. 45–86 (2004)
11. Milojičić, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Comput. Surv. **32**(3), 241–299 (2000)
12. Smith, J.M.: A survey of process migration mechanisms. ACM SIGOPS Oper. Syst. Rev. **22**(3), 28–40 (1988)
13. Zayas, E.: Attacking the process migration bottleneck. ACM SIGOPS Oper. Syst. Rev. **21**(5), 13–24 (1987)
14. Walker, B., Popek, G., English, R., Kline, C., Thiel, G.: The LOCUS distributed operating system. In: Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pp. 49–70 (1983)
15. Miller, B.P., Powell, M.L., Presotto, D.L.: DEMOS/MP: the development of a distributed operating system. Softw. Pract. Exp. **17**(4), 277–290 (1987)
16. Piao, J.T., Yan, J.: A network-aware virtual machine placement and migration approach in cloud computing. In: Grid and Cooperative Computing, pp. 87–92 (2010)
17. Stage, A., Setzer, T.: Network-aware migration control and scheduling of differentiated virtual machine workloads. In: Proceedings of the ICSE Workshop on Software Engineering Challenges of Cloud Computing, pp. 9–14 (2009)
18. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: USENIX Symposium on Networked Systems Design and Implementation, pp. 273–286 (2005)
19. Perez-Botero, D.: A brief tutorial on live virtual machine migration from a security perspective. University of Princeton (2011). http://www.cs.princeton.edu/~diegop/data/580_midterm_project.pdf
20. Theimer, M.M., Lantz, K.A., Cheriton, D.R.: Preemptable remote execution facilities for the V-system. ACM SIGOPS Oper. Syst. Rev. **19**, 2–12 (1985)
21. Hines, M.R., Deshpande, U., Gopalan, K.: Post-copy live migration of virtual machines. ACM SIGOPS Oper. Syst. Rev. **43**(3), 14–26 (2009)
22. Ma, R.K.K., Wang, C.-L.: Lightweight application-level task migration for mobile cloud computing. In: Advanced Information Networking and Applications, pp. 550–557 (2012)