2010 39th International Conference on Parallel Processing

A Stack-On-Demand Execution Model for Elastic Computing

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

***Abstract*—Cloud computing is all the rage these days; its confluence with mobile computing would bring an even more pervasive influence. Clouds per se are elastic computing infrastructure where mobile applications can offload or draw tasks in an on-demand push-pull manner. Lightweight and portable task migration support enabling better resource utilization and data access locality is the key for success of mobile cloud computing. Existing task migration mechanisms are however too coarse-grained and costly, offsetting the benefits from migration and hampering flexible task partitioning among the mobile and cloud resources. We propose a new computation migration technique called stack-on-demand (SOD) that exports partial execution states of a stack machine to achieve agile mobility, easing into small-capacity devices and flexible distributed execution in a multi-domain workflow style. Our design also couples SOD with a novel object faulting technique for efficient access to remote objects. We implement the SOD concept into a middleware system for transparent execution migration of Java programs. It is shown that SOD migration cost is pretty low, comparing to several existing migration mechanisms. We also conduct experiments with an iPhone handset to demonstrate the elasticity of SOD by which server-side heavyweight processes can run adaptively on the cell phone.**

Keywords-stack-on-demand; computation migration; mobile agents; strong mobility; cloud computing

I. INTRODUCTION

The Cloud heralds a new era whereby the provision of computing infrastructure is shifted to the Internet. The paradigm shift offers a scalable pool of resources, typically presented as pay-as-you-go utility services, for application deployment, and enhances IT delivery's efficiency and cost-effectiveness. The ultimate form of cloud computing would be all the more impressive when the concept is continuing to make its way into the vast mobile world. The notion "mobile cloud computing" is ramping up at active pace [1]. Cloud computing is radically changing the development trend of mobile applications. By tapping into the Cloud's enormous computing power and storage, we envision higher sophistication and a wider range of mobile applications. Business users, for example, will find a wide repertoire of mobile productivity kits for collaboration, document sharing, scheduling, and sales force management. Personal use cases include cloud-enabled smart homes, mobile widgets that connect to cloud-managed phone books and calendars, and social networking mashups for sharing photos and videos over cloud storage.

One would ask what would be the best execution model to support these applications in mobile clouds. The model governs resource, code and computational components (e.g. threads), such as where and how to place computations, and whether mobility is supported and which parts (code, execution state and data space) of an executable should be made mobile. We are accustomed to the traditional *client-server* architecture for the Web till date (e.g. RPC, Servlet); and its component-based variant, namely *distributed object*, which allows a system to run computation on a remote system (e.g. DCOM, CORBA, RMI). They are however not designed for mobile applications: code just gets pinned at a site and call flow is unidirectional; only clients can call servers.

Research on mobility support over the last decade would offer a better model to meet the challenge [2]. Mobile code paradigms like code shipping (remote evaluation) and code fetching (code-on-demand, e.g. Java applets) help reduce network traffic by moving the code towards the resource resident site for execution. Transporting merely the code alone without execution state, mainly the call stack and heap of the running program, are far from the best choice for most stateful applications. Distinctively, *mobile agents (MAs)* can capture state, either programmatically (*weak mobility*) or transparently (*strong mobility*) via the underlying runtime support, and migrate state along with the code autonomously at its chosen point of runtime. MA appears a more perfect mobility solution and gives applications several potential benefits: less susceptibility to network failures, reduced network load, decentralization, and easier deployment scenarios. But the conventional MA paradigm is still not as so perfect as we see for agile and flexible mobile-cloud interfacing since a mobile agent itself is a combined unit of code and data that can be costly to migrate. Existing migration mechanisms are either too coarse-grained or not portable. Classical process migration [3] and VM live migration [4, 5] emerged in recent years are too monolithic: moving the entire address space of a process or even the whole (guest) OS image is too expensive, and the transported data is too bulky to fit in low-end mobile devices. Thread migration [6, 7] is relatively lightweight but is difficult to implement in a portable way while not significantly penalizing the execution.

In this work, we propose a compact migration mechanism called *stack-on-demand (SOD)* for stack-based virtual machines to support an elastic execution model. SOD migrates the minimal portion of state to the destination site for continued execution. This is done by exploiting a stack ma-

0190-3918/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPP.2010.79

chine's characteristics, chopping the stack into segments of stack frames and exporting only the top segment. SOD differs from MA in that program code and state, stack and heap, and even frames within a stack can all be decoupled for fine-grained migration purposes. Object misses after migration are handled transparently by on-demand fetching or some prefetching schemes, while the needed classes (program code) can be sent together with the captured frames or dynamically downloaded from the network. SOD paves the way for mobile cloud computing by the features below.

1. ***Lightweight task migration***: no matter how big the process image is, SOD migrates only the required part of data to the destination site. This saves a lot of network bandwidth and imposes lighter resource requirements on the target site, allowing a big task to fit into a small-capacity device in a discretized manner. Most importantly, with an extremely short freeze time, SOD enables quick access to non-local idle computing resources and efficient bidirectional call flow between cloud and mobile entities.
2. ***Distributed workflow style***: SOD's fine-grained migration mechanism allows different parts of the stack migrate concurrently to different sites, forming a distributed workflow. Once the top segment finishes and pops, its return values can be sent to the next site for continued execution. Freeze time between multiple hops is fully or partially hidden. The major benefits are enhanced locality and streamlined elastic task scheduling on cloud servers.
3. ***Autonomous task roaming***: SOD migration mimics the strong mobility feature of mobile agents, and is capable of adaptation to a new environment. A search process, for example, 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.

By SOD, we contribute a lightweight and flexible mobility paradigm for mobile cloud computing. We show how SOD realizes elasticity with some motivating application scenarios in Section II. Another contribution of this paper is our co-design of a novel efficient object faulting mechanism using exception handling to bring remote objects to the local heap on demand. Our way of handling object misses avoids sending the entire heap along the migration and the need to inject per-object state checks as in usual object-based DSMs.

We describe our implementation of the SOD concept in Section III. We make use of the Java Virtual Machine Tool Interface (JVMTI) [8], a standard API, to capture Java stack frames. This approach is very portable, avoiding the need of internal JVM changes and massive bytecode instrumentation for state capturing that imposes severe penalty on execution speed. Besides Java, our methodology is indeed applicable to all stack-based managed codes such as .NET. We evaluate SOD and compare it with live migration in Xen [5], process migration in G-JavaMPI [9] and thread migration in JESSICA2 [6] respectively (the later two are our previous work) in Section IV. Experimental results show the migration overhead of SOD is among the lowest while our object faulting can save up to 2 times of cost than using instrumented object checks. Related work is discussed in Section V; Section VI concludes and highlights our future work.

II. ELASTIC EXECUTION MODEL

The salient features of dynamic scaling or on-demand resource provisioning let cloud computing more or less equate to elastic computing. Mobile clouds make up an even more elastic computing infrastructure where applications may leverage device resources, cloud resources and also some transient surrogate (or cloudlet [10]) resources in between. This calls for a flexible execution model that supports partitioning, delegating and executing tasks of an application efficiently in the distributed runtime environment according to the demand and availability of resources.

We propose a dynamic execution migration mechanism, namely *stack-on-demand (SOD)*, to build this model. SOD is built for Java, inheriting its niceties of high portability and being the most popular programming language [11]. This approach provides applications with seamless and transparent use of non-local resources. It also allows a task to be fulfilled using local resources before attempting a priced cloud service. SOD is a portable yet efficient computation migration mechanism. Such dynamic migration is more preferred than simple RPC or message passing for real-life application scenarios. Ultimately SOD could migrate computation to any device reachable on the Internet, so it is a potential enabling technique to extend the current cloud infrastructure.

A. Stack-On-Demand Execution

SOD is based on a stack machine, specifically JVM in our case. A traditional process is mainly composed of code, heap and stack areas. Each thread has its own *stack* made up of a pile of *stack frames* (a.k.a. *activation records*), each of which contains the state of a method invocation. Only one frame, the top frame for the executing method, is active at any point in a thread of control. When a thread invokes a method, the JVM pushes a new frame onto the thread's stack. On method return, the JVM pops and discards the top frame; its execution result, if any, is passed back to the previous frame which is reinstated as the top frame. So the position in a stack is closely related to the execution flow. SOD exploits this observation to support partial state migration that is yet strong enough for the destination site to seamlessly resume execution. This is done by chopping the stack frames into segments and pushing only the topmost segment to remote while keeping the residual state at the local (home) site. One segment should logically map to one agglomerated task.

Unlike traditional process migration which performs full-rigged state migration (including code, stack, heap and program counter), in SOD, only the top stack frame or segment of frames is needed to carry out the execution, while the required code and data could be brought in on demand subsequently. Upon receiving the migration message, the target site will spawn a worker JVM process, if not yet spawned, and restore into the JVM the captured partial stack. The worker process now starts to execute on behalf of the migrating node. By special bytecode instrumentation (Section III.C), we force the system to throw a null pointer exception when a non-local object is being accessed for the first time. The exception is caught, triggering communication with the home node for fetching a serialized copy of the object. When the migrated frame set finishes execution, i.e. the set's last

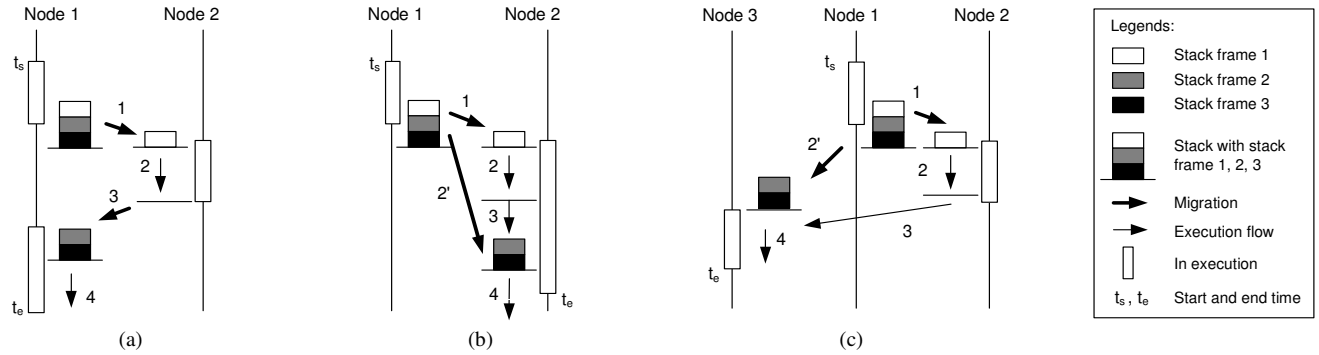

Figure 1. Elastic live migration with flexible execution paths based on SOD migration

frame is about to pop out, return values and updated data will be sent back to the home node, reflected in its heap (the worker JVM will exit upon certain timeout after that). Then execution will resume on the home node, going on with the residual stack. In this way, stack frames representing portion of execution state are exported on demand, hence the name "stack on demand" of this technique.

Indeed, the execution flow of SOD upon completion of a task at the destination site is not necessarily restricted to returning to the home. By pushing the residual stack frames off the home in time, the execution can continue on the current site, or transferred to another site, realizing task roaming. Fig. 1 illustrates three possible scenarios. There are initially three stack frames at Node 1. Fig 1a shows the simple case that the control returns to Node 1 after frame 1 is migrated to Node 2, executed and popped out there. In Fig 1b, after frame 1 is migrated and starts executing, frames 2 and 3 are concurrently pushed to Node 2 as well. The subsequent execution after frame 1 pops will become purely local, i.e. a total migration has happened. Fig. 1c corresponds to distribution of tasks in a multi-domain workflow style. Frame 1 is moved to Node 2 while the segment of frames 2 and 3 is moved to Node 3 in parallel. The control is transferred from Node 1 to Node 2, then to Node 3. With such flexible execution paths, SOD enables elastic exploitation of distributed resources.

B. Key Benefits and Application Scenarios

By SOD's phased and thin state migration, memory pressure on the target node is much reduced. SOD can hence migrate execution to a node with more restricted resources than the home node. SOD can also save the bandwidth consumed by transfers of unnecessary bottommost frames, especially when the task needs to traverse multiple sites.

The use of the SOD execution model is multifarious. We may describe some real-world application scenarios of it here. First, SOD-based task roaming can help speed up distributed information retrieval. Due to fine migration granularity, SOD leads to a more optimal task-to-node mapping. For instance, suppose a thread reaches a call depth of n frames ($n > 3$), and the top three frames correspond to methods intensively reading data from three different data sources on the Internet. To attain locality or affinity to all the three data sources, traditional migration techniques for MAs dictate the need for migrating the full stack for three rounds in a serial manner. Now with SOD (akin to Fig.1c), we can segment the stack and migrate only the top three frames concurrently to their respective data sources. Though the execution still needs to follow a sequential order, the migration latency of each of the later two rounds is effectively hidden by the on-site execution time of the last frame. Having state restored ahead of the passing of control and saving transfer time of (n-3) bottommost frames, task roaming goes more agile than ever.

Second, besides offloading to clouds, execution can also be migrated SOD-style in the reverse direction from clouds to mobile devices. This flexibility opens up innovative and streamlined ways for building distributed applications. Along with the POJO-based development trend [12], application codes are decoupling from heavyweight infrastructure frameworks like EJBs and written as regular Java classes. With SOD's partial stack execution model addressing the resource constraints, applications can run more readily on JVMs for mobile platforms. Then it is no longer necessary to write separately client code and server code for building a distributed application. Instead, we only need to make one version of code and use some migration middleware and policies to control the application behavior. For example, people who want to share their photos with others have to upload the files manually to some websites like Flickr.com. Now, it can be done "elastically". When a user looks for a "beach" photo while the website does not have one, the web server may release some file search processes downstream by migrating a team of thread stack segments to all connected and trusted mobile clients to search for a beach photo. Finally if one is found, the migrant method returns to the server with the photo data which is then forwarded to the requesting client.

Similarly, many applications can delegate work appropriately among server and client sides using SOD. Some intensive processing like global search in database is performed on the Cloud; the search process can be migrated to a mobile device to let it perform localized search or sorting over the search results according to the interactive user options. The eventual SOD execution could work in a speculative manner. By wrapping all code in try-catch constructs via special bytecode preprocessing, if exceptions like `ClassNotFoundException` or `OutOfMemoryException` are thrown, the exception handler will capture the execution state and rocket it into the Cloud that has wider library base and memory capacity for retrying the execution. This exception-driven migration would evolve the general cloud computing philosophy for mobile platforms.

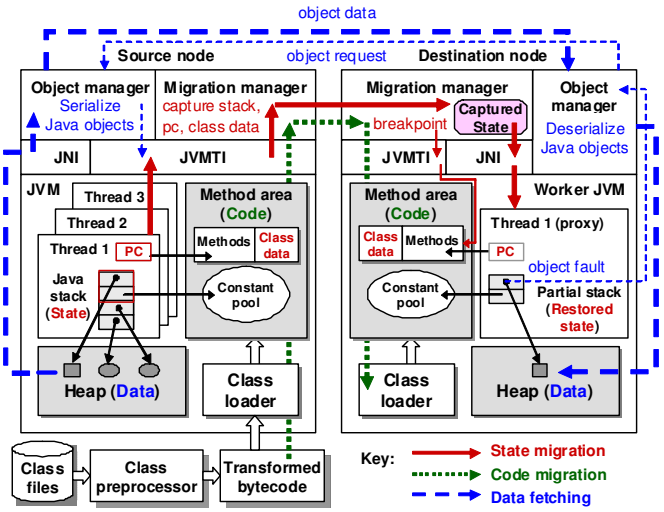

Figure 2. Java-based SOD migration middleware system

```
// save static fields of a class

// for a partial stack of nframe frames
while (depth < nframe){
  // get class name and method name
  jvmti->GetMethodDeclaringClass(method, &class);
  jvmti->GetMethodName(method, &method_name);
  // get program counter
  jvmti->GetFrameLocation(thread, i, &method, &location);
  // get local variable table
  jvmti->GetLocalVariableTable(method, &nlocal, &locals);
  for (int i=0; i<nlocal; i++) {
    // get signature, slot no. and value of a variable
    variable_signature = locals[i]->signature;
    variable_slotno = locals[i]->slot;
    jvmti->GetLocal<Type>(thread, depth, slot, &(value.i));
    // for object type, set variable value to null
  }
  // save them all into a message buffer
  depth++;
}

// send to destination node
```

Figure 3. State capturing code using JVMTI functions

III. System Design and Implementation

A. System Architecture

We implemented the SOD model into a Java distributed runtime named the *SOD Execution Engine (SODEE)*. Fig. 2 shows our system architecture with the key modules below.

1. *Class preprocessor*: for transforming the Java application bytecode before it is loaded into the JVM such that it is able to migrate and run seamlessly on a remote node.
2. *Migration manager*: for serving migration requests and communicating with other migration managers to carry out the state and code migration.
3. *Object manager*: for handling requests to fetch data from the heap of the home and writing execution results of migrated frames back to the home.

We adopted a highly portable design that all of them are working outside the JVM. This approach does not depend on a specific version of JVM, nor requires tricky hacking into the JVM kernel. So the middleware system will be directly compatible across future JVM releases without forklift upgrades. Codes for distributed execution semantics are incorporated into the Java executable ahead of class loading time through the class preprocessor that employs a bytecode engineering library, specifically BCEL [13] in our case, to do so. Class preprocessing is automatic, one-off and performed offline, needing no user intervention or source code modification. During preprocessing, the bytecode is rearranged to facilitate safe migration and augmented with helper functions for state restoration and remote object access.

Since execution state is totally inside the JVM, we use the JVM Tool Interface (JVMTI), the latest standard debugging interface, to expose them. Most modern JVMs support mixed-mode execution: program will run in interpreted mode, experiencing degraded execution performance, if some debugging functions are enabled; and while it is not the case, the program runs in Just-In-Time (JIT) mode. We disable all debugging functions before and after a migration event, so this approach is of reasonably slight overheads.

Migration manager is implemented as a JVMTI agent in C and injected into the JVM at startup time as a command line option. It interacts with the JVMTI layer to access the JVM internal runtime data for capturing the state, essentially part of the Java stack of the migrant thread. It manages code movement towards the destination as well. We assume a worker JVM is pre-started on the destination node for receiving the current class of the top frame. Subsequent classes are transferred and loaded on demand in an event-driven manner (`JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` is being captured and its callback would bring the class from the home). After class loading, the agent invokes the method through the Java Native Interface (JNI). Execution then resumes at the last execution point restored by a special technique using the restoration exception handler injected into the code.

Object manager is implemented in both C and Java. On the destination side, object manager refers to the Java methods that handle sending of object requests and flushing of execution results to the home. On the source side, it refers to the agent thread that listens to object requests, retrieves object references needed via JVMTI and invokes Java serialization via JNI to send the object to the requester. When the active frame encounters object misses, the execution will jump to the object fault handler for the frame (inserted during preprocessing), calling the object manager for fetching the missed object from the source node. Upon reaching the last frame, its return value and updated objects are handed to the local JVMTI client which in turn sends them back to the home. The home JVMTI client will pop the outdated frame off the stack using the `ForceEarlyReturn<type>` functions, supplied with the received return value. Execution will then resume seamlessly on the residual stack.

B. State Migration

The nexus of strong mobility support is about transparent state capturing and restoration. The usual runtime data constituting execution state includes program counter (pc), stack, heap and static data. In Java, pc is managed per thread and refers to the bytecode index (bci) in the current method. Each frame has an array of local variables (including method ar-

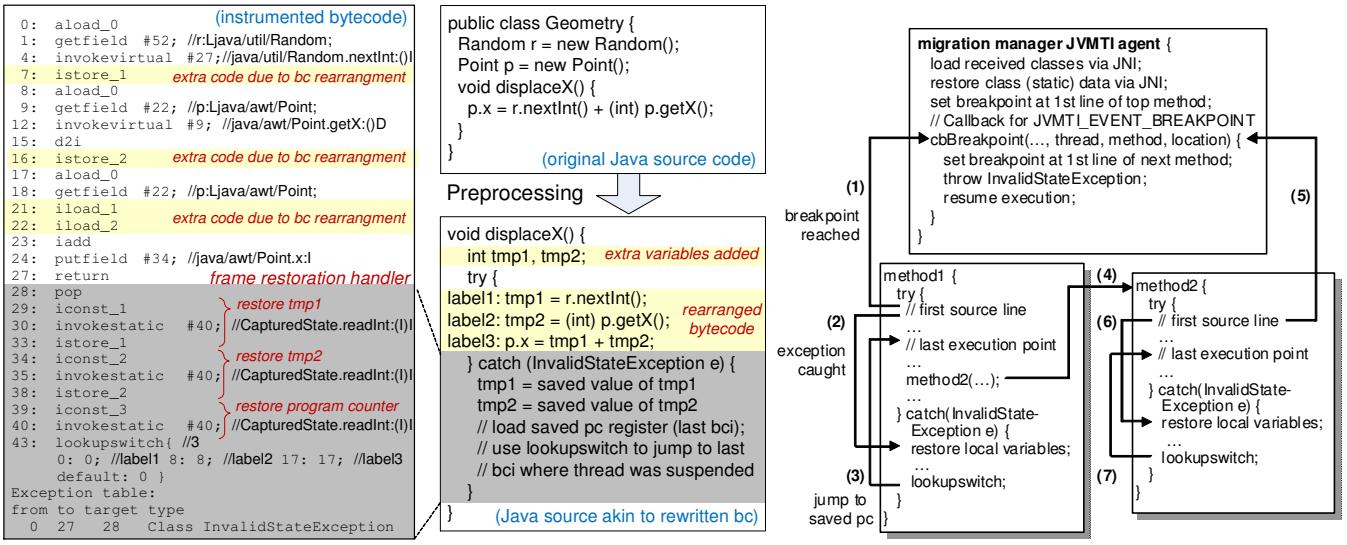

Figure 4. (a) Bytecode instrumentation for adding restoration handlers (b) Per-frame restoration flow

guments), an operand stack (storing intermediate computing results), a reference to the runtime constant pool of the current class, and the return location back to the caller, i.e. last bci or pc. Our design regards the heap not as part of the state, leaves it and the lower part of stack behind, and fetches them on demand. We will further explain how our system captures and restores the state as follows.

1) State Capturing via JVMTI

Upon receiving a migration request, the migration manager suspends the program execution and captures the topmost consecutive stack frames of the thread being migrated. Fig. 3 shows the core logics of the migration manager and the JVMTI functions it uses to capture each frame. First, the information and static fields of loaded classes are saved. For each frame, the current class name, method signature, pc and local variables (their signatures, slot ids, values) are serialized, as a `CapturedState` object, and sent to the destination node whereas referenced objects are left behind.

JVMTI does not provide interfaces to access the operand stack of each frame, nor does it ship with functions to reestablish execution contexts such as the pc. Moreover, when the execution point lies inside a native method, the local data in the frame are machine-dependent. These factors make it difficult to capture and restore the complete state and destroy the system's portability. Our solution is to restrict migrations to happen only at specific points where the operand stacks of all frames are empty and the execution is right outside a native method. These so-called *migration-safe points (MSPs)* are essentially located at the first bytecode instruction of a source code line where the operand stack is always empty. If the execution is suspended at locations other than a MSP, it will be resumed immediately until hitting an upcoming one. MSPs are defined for the current frame only. To make sure operand stacks of other frames are all empty upon migration, *bytecode rearrangement* is needed at the preprocessing stage. For example, in the original Java code snippet shown in Fig. 4a, the only MSP is at bci 0. While evaluating `p.getX()`, the return value of `r.nextInt()` is still on the operand stack of the frame for `displaceX()`. If migration takes place at this moment, we have no ways to capture this operand. To avoid this trouble, we add extra local variables `tmp1` and `tmp2` to store the intermediate values and rearrange the bytecode into three statements. After such preprocessing, now it is safe to migrate at the beginning of all the three statements (i.e. bci 0, 8, 17) and at the MSPs inside the body of `p.getX()` as well because all operand stacks are just empty at these points.

2) State Restoring by Restoration Handlers

For the state restoration, the migration manager loads all the received classes and restores their static field values by calling several JNI functions like `SetStatic<Type>Field()`. Restoring the captured stack frames relies on a concerted use of the breakpoint facility of the debugger and the *restoration handlers* inserted for each method at preprocessing time. Fig. 4a shows an example of restoration handler, the catch block in grey area. The handler code will be activated if a specific exception (`InvalidStateException`) is thrown. It will unwrap the `CapturedState` object through its `read<Type>` calls, reset the local variables (`tmp1` and `tmp2`) to their captured values (at bci 29-38), push the saved pc onto the operand stack (at bci 39-40), and use a `lookupswitch` instruction to jump to the bci where the execution was suspended.

Fig. 4b outlines the procedure of reestablishing the stack frames one by one. The migration manager begins with setting a breakpoint at the start of the top frame, i.e. bci 0 of method1. The breakpoint is reached immediately just after entering the method; the breakpoint event is captured, passing the control to the callback function, `cbBreakpoint` (1). The migration manager then sets another breakpoint at the start of the next invoked method (frame) if there is any. Next, it throws an `InvalidStateException` in the current method. The exception is caught immediately, jumping to the restoration handler (2). The local variables and pc of the current method are then restored. The control jumps to the suspension point (3); it then invokes the next method, i.e. method2, creating the second frame (4). The above steps of restoration are repeated until the last frame gets restored (5) (6) (7).

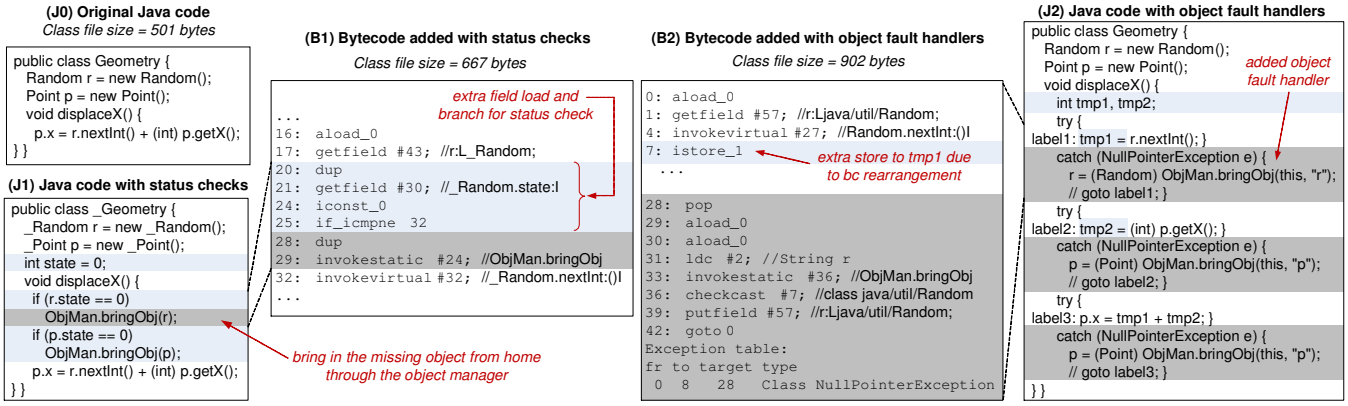

Figure 5. Comparison of object checking and object faulting approaches

C. *On-demand Data Fetching*

In SOD, as in many distributed runtime systems [6, 14], objects are brought to the destination only when they are being accessed so as to minimize transfer of unwanted data. As execution proceeds, the destination JVM will bring in objects and cache them as *local objects*, while the rest kept in the home JVM are seen as *remote objects*. Cached objects may turn stale too. So, prior to using any field of an object, a mechanism is needed to detect whether it is a remote or a local object that is valid for use. One common approach, as adopted in most object-based software DSMs, is to augment all objects with status flags and inject object status checking code at every read and write. This approach would however impose severe penalty on the execution speed since the overhead scales linearly with the access count, even if the objects have been cached locally, e.g. in the case of JavaSplit [14].

We exploit Java's inherent exception safety to deliver a new efficient solution. The key idea is to inject helper codes into the executable as try-catch blocks that detect and handle null pointer exceptions (`java.lang.NullPointerException`) at the preprocessing stage. During state capturing, we do not capture instance fields but set them to null during restoration. So every instance field access will in effect translate into a null pointer exception which will be caught by our injected catch blocks. The exception handling is to invoke the object manager for bringing the missed object to the local heap and rebuilding the object reference being accessed. Such exception handlers that we call *object fault handlers* are able to jump back to the last execution point after handling the exceptions thrown. Technically, the handler realizes this by a `goto` instruction jumping to where the null pointer exception just occurs. Execution then resumes as if no exception has occurred. Bytecode rearrangement with extra local variables introduced in Section III.A.1 has ensured the operand stacks are evacuated to avoid state inconsistency when performing the `goto` instruction. The target of the jump is always at the location corresponding to the start of a source code line. To help rebuilding the missed object reference, the preprocessor looks for `aload` instructions in the program to spot all access to local variables, and creates an object fault handler for each instance variable with its slot id (or field name) being hard-coded inside the code of the handler. The handler provides this id for the call of object manager so that the home object can be located via JVMTI. The home object is then serialized and shipped to the destination. The null reference is set accordingly, pointing to the deserialized object. To differentiate a regular null pointer exception due to application program bug, if a null reference is found at home by the object manager, the handler will throw another null pointer exception to indicate that this exception truly comes from the application level. During normal execution, only the try blocks would be executed. As Java per se has anyway to pay the overheads of exception safety checks, we take this free ride to realize an *object faulting* mechanism, analogous to page faults in OS. The execution speed won't be affected, except a slight side effect of increased code size.

Fig.5 depicts the comparison of the two object miss detection approaches. J0 shows the original Java code. Its bytecode is instrumented and becomes B1 (traditional object checking) or B2 (our object faulting). The equivalent Java codes (J1, J2) are shown for easier interpretation but do not exactly match the instrumented bytecode due to the regime of Java compiler. The shadowed region represents the extra code added (codes marked in pale shadow will be run in normal execution while those in dark shadow are being run only if the object is missed in the local heap).

In the traditional approach, each class needs to be augmented with an extra status field (we add an underscore prefix to the class name of all rewritten classes). We see from B1 that before the program can use the object *r* (calling *r*'s `nextInt` method at bci 32), four extra codes (bci 20-25) are needed to check the object status: `dup` and `getfield` for loading the status field followed by a branch-if operation. If the test fails, bci 28-29 will be executed, calling the object manager, `ObjMan.bringObj()`, to bring in the home object. The overhead of the four extra instructions is associated with every field access no matter if the object is local or remote.

In contrast, our object faulting method does not introduce checking overhead to the original flow of execution as we can see in B2 that the object *r* can be used (at bci 0-4) directly with no code added prior to it. The only overhead is an extra local store (`istore_1` at bci 7) due to bytecode rearrangement. In normal execution where most missed objects have been handled, the direct slowdown induced and indirect slowdown due to increased code size are both negligible. In

term of space overhead, the file sizes of the original class, instrumented classes with status checks and with object fault handlers are 501 bytes, 667 bytes and 902 bytes respectively. Our approach pays 35% more space overhead than the traditional approach to trade for best normal execution speed.

IV. PERFORMANCE EVALUATION

In this section, we evaluate SODEE with several applications. The evaluations were conducted on a cluster of nodes interconnected by a Gigabit Ethernet network. 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 except for those nodes running Xen with live migration for comparison purpose. As only certain modified OS versions support Xen, we installed CentOS 5.4 x86_64 on those nodes and configured each VM instance to have 2GB RAM. All nodes mounted the home directory on Network File System (NFS) to ease experiments with shared file access. The tested JVM version is Sun JDK 1.6 (64-bit).

A. Overhead Analysis

This part of evaluation aims to characterize the overhead of SODEE and see how it compares with other systems. We use the term *migration overhead* to denote the total cost of a migration that is measured by the difference of execution time with and without activating migration. For SODEE, we focus on several components of the migration overhead: (C1) cost of passing through the JVMTI layer, (C2) stack frame capturing, transfer and restoration, (C3) object faulting. We measured C1 by the difference in execution time taken with and without bringing up the JVMTI agent at JVM startup. C2 marks essentially how long the execution gets frozen and we will give a breakdown of this figure. By microbenchmarking, we evaluate C3 separately in Subsection B.

Table I lists our benchmark programs used in this part with the problem size (n), maximum height (h) of Java stacks and accumulated size (F) of all local and static fields. These programs belong to a rather compute-intensive kind. Besides SODEE, we also ran each program atop the following runtimes with migration support: G-JavaMPI, JESSICA2 and Xen. SODEE, G-JavaMPI and Xen all need an underlying JVM; they used the same version, Sun JDK 1.6, as specified. G-JavaMPI uses an earlier generation of JVM debugger interface to perform eager-copy process migration. JESSICA2 performs Java thread migration in JIT mode; its mobility support is implemented at the JVM level by modifying the Kaffe JVM [15]. Xen performs OS live migration; the implementation is at VM level (embedded in Xen).

We recorded the execution time of each benchmark on all systems with and without migration, as shown in Table II. The 2nd column ("JDK") refers to the original execution time taken on Sun JDK. The columns headed "mig" and "no mig" refer to the execution time taken on the migration systems with and without migration triggered. The "no mig" readings of SODEE and G-JavaMPI are about the same because they ride on a similar debugger interface. The raw execution time of JESSICA2 is longest for this system was developed based on a rather old version of Kaffe JVM whose JIT compiler is not as optimized as Sun JDK. Execution in Xen also sees a time penalty. However, in order to use hypervisors, Xen was executed on a modified host OS. So it is not appropriate to conclude that Xen caused a two times slowdown by comparing the "JDK" and "Xen (no mig)" columns.

TABLE I. PROGRAM CHARACTERISTICS

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

TABLE II. EXECUTION TIME TAKEN ON DIFFERENT SYSTEMS

	Execution Time (sec)								
		SODEE		*G-JavaMPI*		*JESSICA2*		*Xen*	
App	*JDK*	*no mig*	*mig*	*no mig*	*mig*	*no mig*	*mig*	*no mig*	*mig*
Fib	12.10	12.13	12.19	12.03	12.19	49.57	49.69	26.65	30.35
NQ	6.26	6.38	6.41	6.27	6.58	38.20	38.40	13.85	18.76
FFT	12.39	12.60	12.71	12.48	15.02	255.3	257.8	16.52	23.68
TSP	2.92	3.04	3.22	3.09	3.23	20.93	21.85	7.01	13.46

TABLE III. MIGRATION OVERHEAD OF DIFFERENT SYSTEMS

	Migration Overhead (ms) [a]			
	SODEE	*G-JavaMPI*	*JESSICA2*	*Xen*
Fib	52 (0.43%)	156 (1.30%)	123 (0.25%)	3695 (13.86%)
NQ	32 (0.51%)	307 (4.89%)	195 (0.51%	4906 (35.42%)
FFT	105 (0.83%)	2544 (20.39%)	2494 (0.98%)	7160 (43.34%)
TSP	178 (5.86%)	142 (4.59%)	922 (4.41%)	6450 (91.99%)

a. Inside bracket is the % overhead w.r.t. execution time w/o migration

Note that even with no migration, there is a slight penalty on SODEE and G-JavaMPI execution due to debugger interface and side effect of code instrumentation. The portion due to the side effect (C0) can be obtained by comparing the execution time of the original and preprocessed classes. C0 was found to be a negligible cost of 0.1% to 1.45%. By subtracting the values of "JDK" and C0 from "SODEE (no mig)" column-wise, we obtain C1 which ranges from 0.1% to 3.2%. We can see C1 is also a minuscule overhead. Based on Table II, we derive Table III showing the migration overhead in each test case (note: slight discrepancy between the two tables exists due to round-off errors). From Table III, we see that SODEE induces the least migration overhead among most of the applications except TSP. In TSP, almost all object fields need be used frequently. There is no benefit for SODEE to reap by deferring heap data transfer through on-demand object fault-in. G-JavaMPI, on the contrary, induced the least migration overhead in this case for it is most efficient to use process migration to bring in all objects by a single transfer. In other applications, as not all of the objects need to be used by the migrated frame, SODEE sees the lowest migration overhead. We observe the largest overhead at Xen because live migration transfers the entire OS that needs at least several seconds to complete the migration.

For further analysis over C2, we refer to C2 as *migration latency* which means the time gap between receiving a migration request and getting the execution resumed at the destination. C2 could be further broken down into three parts. *Capture time* means the interval between a migration request being received and the state data being ready to transfer.

TABLE IV. MIGRATION LATENCY IN DIFFERENT SYSTEMS

App	SOD			G-JavaMPI			JESSICA2		
	Mig. latency (ms)			*Mig. latency (ms)*			*Mig. latency (ms)*		
	Capture	*Transfer*	*Restore*	*Capture*	*Transfer*	*Restore*	*Capture*	*Transfer*	*Restore*
Fib	14.66			132.15			11.37		
	0.35	7.49	6.82	60.17	8.74	63.24	0.39	2.62	8.36
NQ	12.42			91.44			9.06		
	0.50	4.73	7.19	38.44	8.11	44.89	0.18	2.14	6.74
FFT	12.33			2470.15			74.08		
	0.54	4.75	7.04	457.45	1053.57	959.13	0.11	2.26	71.71
TSP	15.23			95.98			9.90		
	0.42	4.50	10.31	36.23	8.32	51.43	0.06	2.30	7.54

Transfer time is the time needed for the state data, upon being ready for transfer, to reach the destination. *Restore time* counts from the moment of state data being available at the destination to the point of execution resumption. Besides the state, code needs to be shipped as well. The time needed to transfer application classes and to load them into the destination JVM would be counted under the restore time. The state data for different systems may be defined differently. For SODEE and JESSICA2, it covers mostly the stack areas; for G-JavaMPI, it includes also the heap; for Xen, it means the entire OS image. For SODEE, G-JavaMPI and JESSICA2, migration latency is equivalent to *freeze time* during which the current execution is frozen. But for Xen, they are unequal for it starts capturing and pre-copying dirty pages to the destination well ahead of execution stoppage. Though its freeze time can be short (usually in hundreds ms range), its migration latency is long, so it is not considered as lightweight migration and excluded from the comparison here.

The results are shown in Table IV. In this evaluation, JESSICA2 achieved shortest migration latency among most of the applications except FFT; SODEE was the runner-up. Capture time and restore time are minimal for NQ and TSP on JESSICA2 and Fib on SOD. Since thread migration in JESSICA2 is built into the JVM, state information can be retrieved directly from the JVM kernel. SODEE retrieves state data indirectly by calling JVMTI functions; this is not as efficient as JESSICA2. Most of the JVMTI functions, e.g. `GetFrameLocation()`, used in state capturing can be executed efficiently and finish within 1us. However, some functions take much longer time (e.g. `GetLocalInt()` take about 30us). As SODEE uses `GetLocal<type>()` functions to capture values of local variables in a stack frame, the capturing time per frame is larger than that of JESSICA2. However, the use of JVMTI allows a much more portable implementation than JESSICA2. State restoration in SODEE mainly relies on the exception handlers injected to each Java method. On the other hand, JESSICA2 sees a much longer migration latency in FFT in which restore time is the major component. This is related to the way that JESSICA2 handles static data. JESSICA2 always allocates space for static arrays at class loading rather than at access time. As a big static array of 64MB is used in FFT, a significant portion of time was spent on array allocation, leading to extensive restore time in JESSICA2. The large array also increased the capture, transfer and restore time of G-JavaMPI considerably. In SODEE, migration was placed at the method which did not need to operate on the array. As a result, its timings were not affected by the size of the array. If migrations were taken place at the methods that need to use the array, then the migration overhead would still be comparatively large. However, this demonstrates a special feature of SOD, which allows a process of large footprint to migrate partial computation that could fit into a resource-constrained device.

Migration latency of SODEE is much shorter than that of G-JavaMPI. By SOD, only the top stack frame was captured and restored. As heap data was not transferred during migration, the data size does not affect SOD migration latency. While in G-JavaMPI, the whole process data is captured with eager-copy, and worse still, all objects are exported using Java serialization. For Fib and NQ, G-JavaMPI needs to deal with around 46 and 16 stack frames (see Table I) respectively, due to recursive calls. For FFT, the total object size is larger than 64MB. As a result, G-JavaMPI sees much longer capture time and restore time than SODEE in all cases.

TABLE V. COMPARISON OF OBJECT FAULTING METHODS

Access Type	Access time (ns)			Slowdown	
	Original	*Object faulting*	*Object checking*	*Object faulting*	*Object checking*
Field Read	2.60	2.68	3.87	3.08%	48.85%
Field Write	5.67	5.79	7.13	2.12%	25.75%
Static Read	0.37	0.38	0.45	2.70%	21.62%
Static Write	0.13	0.14	0.46	7.69%	253.85%

B. *Detection of Remote Object Access*

In this part, we compare the slowdown due to object fault handlers with that of the traditional approach of injecting object status checking codes to detect remote object access. We measured the time needed to read and write an instance field and static field using a micro-benchmark program. As shown in Table V, our object faulting approach introduces 2.12% to 7.69% overhead, while status checking causes a slowdown of 21.62% to 253.85%. The exceptionally large slowdown happens to static write access because an additional static "status" field read imposes an extra latency up to 0.37ns on the relatively short time of a static field write. Since the status of the object needs to be checked before every field access, an additional field read (for the object status), a comparison and a branching instruction are required. These actions penalize the execution time even if the objects are available in the local heap. In contrast, the overhead induced by object fault handlers is nearly negligible.

C. *Effectiveness of Different Migration Techniques*

In this part, we compare the effectiveness of different migration techniques for improving data access locality. Due to different levels of implementation, it is not fair to compare their execution time directly, so we compare the performance gain due to migration. We used a full-text document search application for this evaluation. In this application, files are read locally if the files are located in the current node, or otherwise, through NFS, followed by searching for a string over the buffers read. We did the test with three large data files (600MB each). Locality of file access can be improved by migrating the execution to the nodes which host the files.

TABLE VI. PERFORMANCE GAIN ON MIGRATION SYSTEMS

System	Exe. time w/o mig. (sec)	Exe. time w/ mig. (sec)	Exe. time on NFS server (sec)	Performance gain
JESSICA2	358.10	348.08	343.31	2.88%
Xen	57.72	57.29	50.71	0.75%
SODEE	23.25	18.81	16.01	23.60%

The execution was repeated three times on each system. For the first run, the program was initiated and executed on the NFS client node till the end with no migration. For the second run, after the program was started on NFS client and before reading any files, we triggered migration to move the execution to the NFS server. For the third run, the program was run locally on NFS server. The OS buffer cache was cleared prior to each run to isolate the locality effect.

Table VI shows the results obtained. SODEE delivers the best performance gain, 23.60%, among the three systems. JESSICA2 obtains a slight gain of 2.88% only. We suspect some bottlenecks exist in the I/O library of the JVM implementation. So even if the file data are available locally, it does not help speed up the file reading. Xen attains a negligible gain of 0.75% only for most of the locality benefit has been offset by the relatively large migration overhead.

We further studied the autonomous task roaming feature of SOD. A roaming process only needs to ship the needed state from one node to another. We simulated a WAN-connected Grid environment where ten NFS servers were used to host ten data files of about 300MB. The program was initiated on a node. Then it retrieved the ten files, one from each server, and performed a text search for a string. If there was no migration, the whole file data would need to be transferred from the server to the executing node over NFS. But with migrations, access locality could be exploited by bringing the task to the server, and doing the search locally there. The maximum performance gain was attained by activating ten rounds of SOD migrations, pushing the task to each NFS server. The execution time dropped from 124.3 sec to 36.71 sec, giving a speedup of 3.39 over the case of no migration.

D. SOD for Devices in Bandwidth-limited Environment

To testify that SOD can help a resource-demanding job execute on a resource-poor device, we performed a separate experiment on a bandwidth-limited network using an iPhone 3G handset with a 412MHz ARM CPU, 128MB RAM, and 16GB storage. The iPhone was installed with JamVM 1.5.1 b2-3 (JVM) and GNU Classpath 0.96.1-3 (Java class library). It was connected through Wi-Fi connection to the cluster network whose bandwidth was deliberately controlled by the bandwidth control service of a router. We developed an application mimicking the scenario mentioned in Section II.B. The application is a web server program for photo sharing. When the program receives requests from client browsers, it searches for images in specific directories, generates HTML pages with links to the photos found, and sends them back to the clients. Clients can then view the photos by following the links. SOD migration is used to allow the photos stored on the iPhone to be shared with the server and hence with other clients without the need of installing any web or FTP servers on the iPhone beforehand. This is done as follows:

TABLE VII. MIGRATION LATENCY VS. AVAILABLE BANDWIDTH

Bandwidth (kbps)	Capture time (ms) t1	Transfer time (ms)		Restore time (ms) t4	Migration latency (ms) (t1+t2+t3+t4)
		State data t2	Class file t3		
50	14.05	766.00	908.33	40.33	1728.72
128	13.16	796.67	398.67	50.00	1040.33
384	14.37	321.67	407.33	28.67	772.04
764[a]	13.50	280.00	392.50	30.50	716.50

a. Case of unlimited bandwidth (764 kpbs measured)

1. The server calls the method to search over some device-specific directory (e.g. "`/User/Media/DCIM/100APPLE`").
2. Just before the search is carried out, the task (the method frame) is migrated to iPhone using SOD migration.
3. The resumed task then searches for photos available in the specified directory on the device.
4. After searching finished, the task returns to the original web server with the list of photos found and published as web pages for clients to view the list of links to photos.
5. When a client clicks on a specific link, the server pushes a new task to iPhone by SOD again. The task then returns to the server node with the requested photo data. The server will send back the photo to the client for viewing.

Among the various migration techniques, only SOD can support this application. Since the web server maintains active connections to the outside world, process migration as in G-JavaMPI is not allowed. SOD addressed this issue by pinning down the frames that hold the socket connections. Table VII shows the migration latencies obtained under different bandwidth constraints. The transfer time of state accounts for the major part in the latency, i.e. the smaller the bandwidth, the longer the migration latency. However, the capture time and restore time are not affected by network bandwidth. Compared with migration to a cluster node, the capture time and restore time of migration to the iPhone are much longer. This is because JVMTI is not available in JamVM, so we used Java serialization to save captured data in a portable format so that it is restorable without JVMTI on the device side. We retrofitted the migration manager as a pure Java worker program that uses Java reflection to load classes of the application and call its methods for restoring stack frames. Carrying out restoration at Java code level with rather low processing power of the device makes the restore time much longer than that taken on a cluster node. While this was the case, the use of Java serialization and migration manager retrofitting will no longer be necessary when more mobile JVMs go for implementing JVMTI; as we recently see, phoneME [16] has done so as of this writing.

This application demonstrated that SOD can be used to increase mobility and flexibility of execution by allowing a big job to partially run on mobile devices, and hunt for on-device resources such as files. Originally, without installing specific server software that is clearly unfit to mobile devices, the file resources can hardly be shared with the external entities. Now, this can be achieved easily using SOD execution.

V. RELATED WORK

Zhang et al. propose an elastic application model [17] to support partitioning a single application into multiple com-

ponents called *weblets*, and dynamic adaptation of weblet execution configuration. A weblet is location transparent. It can be run on mobile devices or migrated to the Cloud. However, application developers need to follow their specific framework and programming model. Our SOD model however allows a general application to be partitioned transparently by the runtime. Cloudlet [10] 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.

Existing work on mobile agent systems (MASs), such as Aglets, Voyagers, etc, focuses on mobility alone and pays little attention to resource constraints and lightweight migration. All codes and execution states are carried throughout the whole migration itinerary. This wastes much bandwidth. Comparatively, SOD saves the transfer of unnecessary execution contexts (the lowermost stack frames). Our previous work [18] tackled the execution problem of mobile agents in resource-limited pervasive computing environments. Codes and states are segmented and retrieved from the source site on demand for execution. That work focused on adaptation of execution in pervasive devices while this work focuses on lightweight migrations in mobile cloud environments.

CIA project [19] relies on Java Platform Debugger Architecture (JPDA) and bytecode instrumentation to achieve transparent migration. Branching and other instructions are inserted to each method's beginning. This affects the normal execution speed even when there is no migration. Our approach adds restoring codes as exception handlers and imposes no penalty on normal execution. G-JavaMPI, our previous work, adopted a similar state capturing and restoring approach as in SOD. Their flexibility however differs a lot: G-JavaMPI can migrate the whole process to a single node only while SOD divides a process into segments being migrated concurrently to different cloud or mobile entities.

JavaSplit is a distributed shared memory (DSM) runtime. It employs bytecode instrumentation to insert status checking codes before each object access. These codes induced slowdown of multiple times to all object reads and writes. On the contrary, our object faulting approach saves this heavy penalty by trading more code space for the best execution time.

VI. Conclusion and Future Work

The paper proposes a *stack-on-demand (SOD)* execution model with a compact migration technique to realize elastic computing in mobile cloud environments. By segmenting the stack of a running thread, SOD enables lightweight partial state migration. The use of SOD is versatile: it can facilitate offloading tasks of a mobile application to the Cloud, or conversely, streaming a process of large footprint down to a resource-poor device in a discretized manner. SOD can also support advanced features like autonomous task roaming and distributed execution flow without carrying needless data throughout the migration itinerary. For the best SOD-based performance, we design an object faulting mechanism based on exception handling to detect remote object access. While dynamically bringing in remote objects is nothing new, our object faulting technique accomplishes "heap-on-demand" much more efficiently. Experiments show that SOD induces less overhead than other migration systems for most of the benchmarks. The SOD concept can be further explored in terms of migration, prefetching and task distribution policies.

Acknowledgment

This research is supported by Hong Kong RGC grant HKU 7179/09E and China 863 grant 2006AA01A111.

References

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

[2] A. Fuggetta , G. P. Picco and G. Vigna. "Understanding Code Mobility." *IEEE Transactions on Software Engineering*, v.24 n.5, p.342-361, May 1998

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

[4] "VMware VMotion for Live Migration of Virtual Machines." Internet: www.vmware.com/products/vmotion/

[5] 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 Symposium on Networked Systems Design and Implementation*, p.273-286, 2005

[6] 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 International Conference on Cluster Computing (CLUSTER 2002)*, p.381-388, Chicago, USA, Sep 2002

[7] 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

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

[9] 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

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

[11] "TIOBE Programming Community Index" Internet: www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[12] "The Rise of the POJO" Internet: www.oracle.com/technology/tech/java/newsletter/articles/rise_of_the_pojo.html

[13] "BCEL" Internet: jakarta.apache.org/bcel/

[14] 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

[15] "Kaffe JVM" Internet: www.kaffe.org

[16] "phoneME Java ME" Internet: https://phoneme.dev.java.net/

[17] X. Zhang, S. Jeong, S. Gibbs, and A. Kunjithapatham. "Towards an Elastic Application Model for Augmenting Computing Capabilities of Mobile Platforms," In *Proc. of the 3rd International ICST Conference on Mobile Wireless Middleware, Operating Systems, and Applications (MobileWare)*, Chicago, USA, Jul 2010

[18] Y. Chow, W. Zhu, C. L. Wang, and F. C. M. Lau. "The State-On-Demand Execution for Adaptive Component-based Mobile Agent Systems," *In proc. of the Tenth International Conference on Parallel and Distributed Systems (ICPADS 2004)*, p.46-53, Newport Beach, California, USA, Jul 2004

[19] 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 International Conference on Mobile Agents*, Atlanta, Georgia, USA, Dec 2001