An Enhanced DSM Model for Computation Offloading

Jaemin Lee*, Yuhun Jun† and Euiseong Seo*
* College of ICE, Sungkyunkwan University, Republic of Korea
† Memory Business Unit, Samsung Electronics Co., Ltd., Repulic of Korea

***Abstract*—The distributed shared memory (DSM)-based computation offloading scheme allows collaborative multiple threads to dynamically migrate and execute across a mobile device and computing nodes. Despite this strong advantage, it misses a significant portion of the potential performance gain because the traditional DSM model is suboptimal for computation offloading. This paper proposes an enhanced DSM model that aims to enable multiple computing nodes to efficiently and reliably offload concurrent multiple threads from a mobile device. To achieve this design goal, we propose the following novel schemes: a) *selective object tracking* minimizes the set of objects to be monitored by the DSM layer; b) *lock-thread repartitioning* dynamically relocates threads and locks in order to reduce remote lock acquisitions and inter-node synchronizations; and c) *thread-state checkpointing* protects the data and context upon unexpected system failures. We implemented METEOR, which is a prototype based on the proposed schemes, and evaluated it with diverse applications. The evaluation showed that METEOR, with four computing nodes, improved the performance by up to 109% and reduced energy consumption by up to 52% in comparison with the previous DSM-based offloading scheme.**

I. Introduction

Computation offloading is a computing model that offloads parts of computing-intensive tasks to the cloud or cloudlet to overcome the resource constraints of mobile devices [1], [2]. In comparison to the traditional client/server model, the computation offloading model can save software development and server management costs because it does not require dedicated server systems. In addition, since computing nodes can be flexibly located in cloudlets or network edges [3], which are in close proximity to mobile devices, computation offloading can significantly reduce wide-area network traffic and provide short response time.

The emerging mobile applications and services, including augmented reality, intelligent assistant, and 3D image processing, demand high performance, which also imposes a heavy burden in terms of battery life. In addition, the sensor nodes or intelligent things should be able to collect and process a large amount of data and provide diverse services under very limited battery capacity and computing capability [4]. Because it can remarkably increase the computing performance and battery lifetime of mobile devices, computation offloading is expected to become an essential technology for the forthcoming mobile applications and Internet-of-Things (IoT) era.

The distributed shared memory (DSM)-based offloading scheme [5] allows an existing thread to dynamically migrate to a computing node simply by transferring the thread state to the computing node. Because the live migration of threads can occur at any point during execution, existing applications can obtain a significant performance boost without modification. In addition, because the threads running at one end share address space with the threads at the other end, threads can execute simultaneously and collaboratively across the mobile device and computing nodes, and thus fully utilize the computing resources of both ends.

However, despite this strong advantage, the DSM-based offloading scheme lacks a significant portion of the potential performance gain because the original DSM model was designed for high-performance clusters and is not yet well-fitted to the computation offloading service. The performance drawback resulting from the inefficiency of the DSM model is expected to significantly impair the scalability and efficiency of computation offloading.

This paper proposes the enhanced DSM model for computation offloading. The proposed DSM model can efficiently and reliably offload collaborative threads over multiple computing nodes without the modification of existing applications. To achieve this design objective, we propose three novel schemes for the DSM model: *selective object tracking*, *lock-thread repartitioning* and *thread-state checkpointing*.

A DSM consistency protocol must keep track of changes to shared objects. The existing distributed runtime environments consider all objects as shared ones because it is challenging to predict without developers' annotations whether an object will be shared. Selective object tracking minimizes the set of monitored objects to reduce the monitoring overhead by providing the means to resolve object faults, which occur when a thread tries to access a formerly non-shared non-local object.

Remote lock accesses cause notable networking and performance overhead. Although a thread can be freely migrated to the home node of the frequently accessed lock, the blind placement of locks and threads will result in a disastrously uneven load balance because a popular lock will shepherd a large number of threads along with their relevant locks to its home node. The lock-thread repartitioning scheme dynamically relocates threads and locks considering the lock access patterns and load distribution to reduce remote lock accesses.

The increased scale makes the offloading system more prone to network or computing node failures than before. If the offloaded process is dealing with important data or conducting mission-critical tasks, the failure will result in catastrophic data loss or abrupt service termination. To prevent

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

this, METEOR provides a failure recovery scheme that relies on thread-state checkpointing so that applications can resume upon unexpected failures.

We implemented *METEOR*, which is the prototype computation offloading system built upon the proposed DSM model, in the Dalvik process virtual machine (VM) of Android OS, and evaluated it in terms of performance, energy consumption and network traffic using diverse applications.

The remainder of this paper is organized as follows. In Section II, we summarize existing computation offloading schemes and briefly analyze the existing DSM-based computation offloading approach. Section III proposes the enhanced DSM model. The implementation details of METEOR are introduced in Section IV, and the evaluation of the prototype implementation is presented in Section V. Finally, Section VI concludes our research.

II. Background and Related Work

A. Approaches to computation offloading

The existing computation offloading approaches can be categorized into three by their offloading units [2], [6]: the method-level approaches [7]–[13], the component-level approaches [14]–[17], and the thread-level approaches [5], [18].

In method-level offloading schemes, when an offloadable method is called in the mobile device, the method in the computing node is executed instead. This approach requires minor modification of the application code, which relates to the annotation or declaration of offloadable methods [9]–[11].

In such schemes, an offloaded method instance is executed within the conventional process boundary, which must stay in a single node. ThinkAir [12] achieves scalability for the offloaded workload by automatically parallelizing and distributing it over multiple VMs. However, this approach works only when the workload is parallelizable.

In order to fully utilize the abundant cloud resource, component-level offloading approaches extend the traditional distributed application model to the combination of the front-end object located in the mobile device and the back-end objects that can run in the computing nodes as well as in the mobile device.

μCloud [14] can compose rich mobile applications by combining a local front-end object with multiple remote back-end objects. In the Weblet [15] framework, an application is a combination of a light-weight application and weblets, which are computing components similar to small Web servers. Yang et al. [16] also proposed a component-level offloading scheme for stream data processing.

These component-level offloading schemes are superior to the method-level offloading schemes in terms of scalability. However, the fact that existing applications must be considerably restructured is a significant obstacle to their popularization. In addition, a component instance can still be executed only within a single node, and scalability can be achieved only when the component is designed to scale out. Finally, the complex relationships among objects make fault tolerance to node failure difficult to achieve.

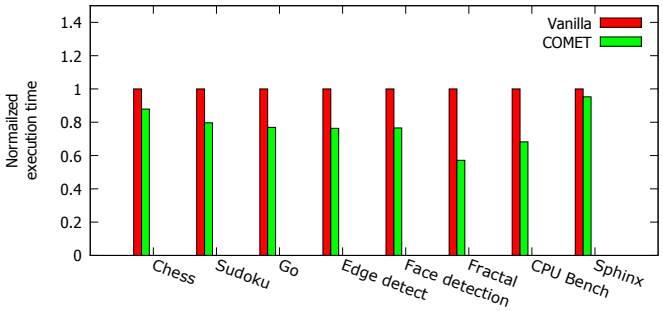


Fig. 1: Execution time of applications using COMET normalized to unmodified Android.

A computing node operates VMs that have the same software stack as the mobile device in the CloneCloud framework [18], which is a thread-level offloading scheme. A thread running in a mobile device can migrate to a VM in the computing node at any point of execution by transferring the memory structure and thread state. However, because CloneCloud does not support DSM, collaborative multithreading and data sharing across the mobile and the computing node are not possible.

B. COMET DSM offloading model

COMET [5] is a distributed runtime environment that enables the existing threads in the Dalvik VM of a mobile device to be migrated to the Dalvik VM in a computing node. Because the Dalvik VMs in the mobile device and computing nodes are tied together in the DSM layer, all threads can access the same address space regardless of their location, and therefore can collaborate with each other. However, the current design of COMET allows only one computing node.

In COMET, the τ-scheduler determines whether to offload a thread at every code block entry. The thread migration decision is made based on the round-trip time of the network communication and duration of the thread lifetime. The migration of a thread can be easily accomplished by transferring the thread state and stack because the Dalvik instance in the computing node sees the same address space as the mobile device through the DSM layer. However, in order to maintain the consistent shared address space, COMET requires timely synchronization of the shared data between the mobile device and the computing node.

The field-based DSM model, which was proposed for COMET, employs lazy release consistency [19] at the granularity of the field level. The Dalvik VM for the field-based DSM keeps track of every modification to the field of an object in the tracked object set, which is a set of objects being monitored for the consistency protocol. During synchronization, these modifications are transferred from the pusher to the puller and reflected to the address space of the puller. Synchronization occurs on every thread migration and remote lock acquisition. The source node becomes the pusher

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

Application	Description	Locks	Threads	Exec. Time
Chess	Chess game, conducts BFS to find best moves	10	31	238 s
Sudoku	Sudoku game, solves Sudoku for given values	8	38	156 s
Go	Go game, plays a go game with a CPU opponent	16	47	482 s
Edge detect	Image filter, detects edges of given images	5	24	101 s
Face detection	Face recognizer, recognizes faces of given images	9	42	109 s
Fractal	Math tool, renders and zooms Mandelbrot fractals	14	37	150 s
CPU Bench	Computation benchmark, executes CPU-intensive tasks	12	81	304 s
Sphinx	Speech recognition, recognizes speech input from microphone	23	100	438 s

TABLE I: Characteristics of applications used for evaluation.

	Mobile Device	Computing Node
OS	Android 4.4.2	Ubuntu 14.04
CPU	Krait 400	Xeon E5-2407
Core	4 × 2.2 GHz	Dual 4 × 2.2 GHz
LLC	2MB L2	10MB L3
Memory	2 GB	64 GB
Network	802.11n: 72 Mbps 0.97 ms RTT LTE: 40 Mbps 35.4 ms RTT	1 Gbps Ethernet

TABLE II: System configuration of a mobile device and computing nodes used for evaluation.

Operation	Mobile Dev.	Comp. Node
Object tracking	26.28%	33.54%
Synchronization	15.65%	15.28%
Scheduling	1.59%	0.99%
Other overheads	2.28%	1.89%
Application	54.20%	49.57%

TABLE III: Execution time composition of CPU Bench using COMET categorized by offloading components.

on thread migration, and the lock acquiring node becomes the puller on remote lock acquisition.

The field-based DSM partially adopts the adaptive object home migration scheme of JESSICA2 [20] for lock object management. The node that creates a lock object becomes the initial home of the lock. When a thread acquires a remote lock, in order to reduce possible future remote lock accesses and the accompanying synchronizations, the τ-scheduler chooses between two options: migrating the thread to the home node of the lock or changing the home node of the lock to the node of the thread.

However, both choices cause serious performance degradation. If τ-scheduler decides to migrate the lock-holding thread to the lock home, the home node will eventually be filled with all the threads that have previously acquired the lock. This will result in a catastrophic load imbalance. If τ-scheduler decides to migrate the lock holder back to its original node after it releases the lock, the previous migration decision, which transferred the thread to the lock home, becomes useless. The same situation as that of the lock home migration also occurs if the lock is shared by other threads spread over the computing node and mobile device. The performance impact from these phenomena further worsens as the number of concurrent computing nodes grows. Section V quantitatively identifies this property.

The underlying DSM model markedly affects the performance of a distributed system. In order to examine the performance impact of the field-based DSM model, we measured the execution time of the workloads described in Table I using the hardware setup shown in Table II. The results are compared with those from the unmodified Android OS, which are marked as *Vanilla*. Figure 1 shows the normalized average execution time. COMET attained a 12% to 45% performance gain depending on the workload characteristics.

We investigated the execution time composition with Callgrind [21], a function profiling tool, to explore the possibility of further performance improvement. When offloading, both object tracking and synchronization consumed nearly a half of the processing time as shown in Table III. As stated earlier, the field-based DSM regards all objects as shared ones to transparently support existing applications, and thus all object accesses must be intervened by the consistency protocol. Therefore, a large amount of field accesses must be monitored and handled by the DSM layer. Moreover, accesses to the tracked objects, which are initiated by both object tracking and synchronization, must be performed in a critical section because a race condition may occur. As a result, the object tracking and synchronization overhead will increase as the number of threads grows.

III. Design

A. Selective object tracking

As explained in Section II-B, tracking all object accesses results in significant performance loss and poor scalability. We propose the *selective object tracking* scheme to remedy this drawback.

The inter-node synchronization only of the shared data, not of all data, will sufficiently guarantee the correct consistency. Naturally, if we can minimize the tracked object set by excluding all unshared objects, we can improve the field access performance and further reduce the network traffic for synchronization while guaranteeing correct consistency. The obstacle to this is that an unshared object can be accessed by a remote node at any time and that it is technically challenging to predict remote access to a local unshared object.

Selective object tracking defines unshared and shared objects as objects that have been accessed only by one thread and objects that have been accessed by multiple threads, respectively. An object is initially unshared, and it permanently

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

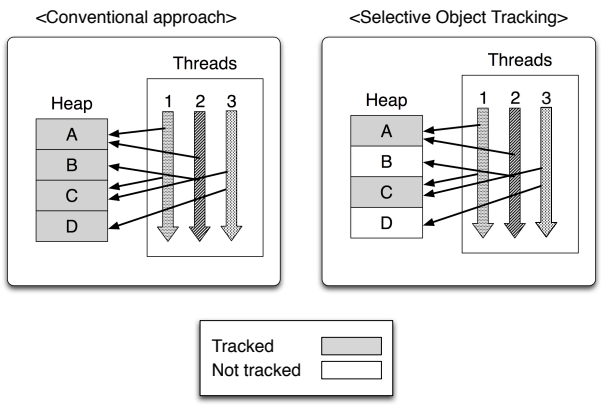


Fig. 2: Selective object tracking minimizes tracked object set.

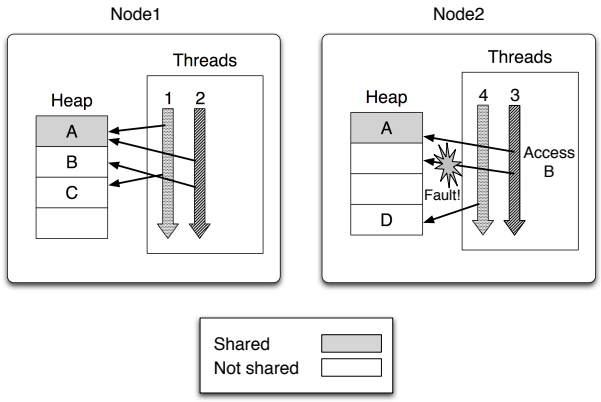


Fig. 3: An example of a shared object fault.

turns into a shared object once it is accessed by a thread other than the one that accessed it before. Selective object tracking does not track unshared objects, as shown in Figure 2.

Because an unshared object is not included in the synchronization process, when a node accesses an unshared object located in a remote node, the address for the object is not allocated. Therefore, this kind of access results in a *shared object fault* as shown in Figure 3. In the beginning, only *A* is a shared variable, and *B*, *C*, and *D* are unshared variables. Thus, *A* is included in synchronization and is present in both node 1 and node 2 while *B* does not exist in node 2 because it was created by thread 2 in node 1 and has never been accessed by other threads. The moment thread 3 accesses *B*, a shared object fault occurs.

In order to resolve the shared object fault, selective object tracking requires a new component, the object fault handler. The object fault handler is invoked when a shared object fault occurs. The object fault handler suspends the thread that generated a shared object fault and sends queries to the other nodes to check the existence of the object at the faulting address. If a node has an object at the given address, then the node changes the object to a shared object and synchronizes the object with the faulting node. The faulting node places the object at the faulting address and resumes the execution of the suspended thread. When a thread accesses an unshared object that the thread has never accessed before and exists in the local node, the object simply turns into a shared object and no shared object fault occurs.

Because the object fault handler suspends the thread execution and conducts all-node communication, frequent occurrences of shared object faults may significantly degrade performance. However, actual object faults rarely occur because a shared object fault for an object occurs at most once in the entire node pool.

B. Lock-thread repartitioning

As explained in Section II-B, using a remote lock may cause a hotspot filled with migrated threads when a lock acquirer is forced to migrate to the lock home. It may also produce a large number of synchronization events when the lock moves back and forth following the acquirer. We propose the lock-thread repartitioning scheme that periodically relocates locks with their relevant threads in the same node to reduce both thread migrations and synchronization events due to remote lock acquisition.

We apply the approach of AIDE [22], which transforms a resource partitioning problem into a graph partitioning problem, and solve the transformed graph partitioning problem with a heuristic algorithm, as follows.

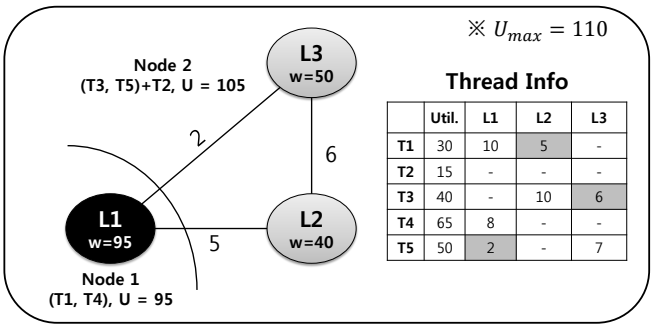
Suppose that there is an undirected weighted graph that has locks as vertices. An edge is formed between two locks when there exists at least one thread that accesses both locks. The primary lock of a thread is defined as the lock that the thread most frequently acquires. The weight of an edge represents the number of remote lock accesses caused by both locks on the assumption that the locks are located in two different nodes and threads are collocated with their primary locks. The weight of an edge, $e(x, y)$, that connects vertices x and y is defined by Equation 1.

$$e_{(x,y)} = \sum_{i=0}^{m} min(Counter(t_i, l_x), Counter(t_i, l_y)) \quad (1)$$

$Counter(t_i, l_x)$ is the acquisition count of lock l_x by thread t_i. t_i is the i-th thread that uses both locks, x and y, and m denotes the number of such threads.

Figure 4 shows an example of the lock-thread partitioning problem. There are three locks, L1, L2, and L3, and five threads, from T1 to T5 in the system. Each thread demands CPU utilization, as shown in the Thread Info table, and acquires each lock, as shown in the same table. L1 and L2 are connected via T1, and the weight of the edge is 5. The edges between L1 and L3 and between L2 and L3 are formed by T5 and T3, and their weights are 2 and 6, respectively. Because remote lock accesses are minimized when all threads are collocated with their primary locks, the weight of a vertex, which means the computation demand induced by the lock, is chosen to be the utilization sum of the threads that have the lock as their primary lock. Therefore, the weight of L1 is 95, that of L2 is 40, and that of L3 is 50.

978-1-5090-4327-9/17/$31.00 ©2017 IEEE



	Util.	L1	L2	L3
T1	30	10	5	-
T2	15	-	-	-
T3	40	-	10	6
T4	65	8	-	-
T5	50	2	-	7

Fig. 4: An example of lock-thread partitioning.

When the number of computing nodes is K, if we can partition the locks into K non-empty groups so that the sum of the edges connecting the groups with each other is minimized, we will get the optimal lock placement that minimizes remote lock accesses by simply locating the locks of each group in each node. Partitioning vertices into K groups while minimizing the weight sum of the edges that connect the groups with each other is the well-known minimum K-cut problem [23]. However, because threads are located with their favorite locks, the computation capability of computing nodes and the performance demand of threads must be considered together with the lock partitioning problem.

All locks and threads in a partition are located in the same node. Therefore, the sum of the thread utilizations in a partition cannot exceed the computing capability of a node. The minimum K-cut problem with the constraint that the sum of the vertex weights in a partition cannot exceed a given upper limit is one of the constrained optimization problems classified as an NP-hard class problem [24]. Of the few heuristic solutions for the problem, we apply the Kernighan-Lin (KL) algorithm [24].

Any K partitions of a graph can be given to the KL algorithm as an input. The KL algorithm chooses and exchanges two vertices from two partitions, one from each, to earn the maximum gain, which is the sum of the partition-connecting edges. This choice and exchange operation is carried out for all pairs of K partitions, defined as a stage in the KL algorithm. The KL algorithm repeats the stage until no gain is obtained from the last stage. The time complexity of the algorithm is $O((kn)^2 \log n)$, where k is the number of partitions and n is the number of vertices. Both the number of shared locks and the number of nodes are generally small. However, we can suppress the computing overhead by applying an optimization method [25] in case that the time complexity adversely affects the overall performance.

The actual partitioning is carried out following the pseudo code in Figure 5. First, the proposed scheme builds the initial graph, G', which locates the threads with their primary locks, according to the metadata table (line 2). Second, it checks whether there exists an overcommitted vertex in G' whose weight exceeds the computing capacity of a node, U_{max} (line 3). If there is an overcommitted vertex in the input, the KL

```
1: function LT_PARTITIONING(G)
2:     G' ⟵ build_initial_graph(G)
3:     flag ⟵ check_overcommitment(G')
4:     G'' ⟵ redistribute_threads(G', flag)
5:     G* ⟵ KL_algorithm(G'')
6:     Result ⟵ distribute_nolock_threads(G*)
7: end function
```

Fig. 5: Pseudo code of lock-thread partitioning.

algorithm cannot place the vertex in any partitions. Therefore, the threads allocated to the overcommitted vertices must be redistributed to other vertices (line 4). The vertex with the lowest load takes the surplus threads from the overcommitted vertices in such cases. The proposed scheme then carries out the KL algorithm for the resulting graph, G'' (line 5). Finally, the threads that do not use any locks will be placed with arbitrary nodes with the available capacity (line 6).

In the example, this procedure will yield two partitions, $\{$L1$\}$ and $\{$L2, L3$\}$, where $K = 2$ and $U_{max} = 110$. Node 1 accommodates $\{$L1$\}$ and node 2 holds $\{$L2, L3$\}$ with their accompanying threads. Because T2 does not use any locks, it is randomly allocated to node 2.

After partitioning, the locks and threads are migrated to their designated nodes according to the result, and then the threads resume execution. The lock home is changed only by lock-thread repartitioning, and the thread acquiring a remote lock is always migrated to the lock home in this scheme.

Applications usually undergo multiple phases during execution, and the phase changes dramatically alter the lock access patterns of their threads. Therefore, the repartitioning of locks and threads is required when an application enters a new phase. However, because it is technically challenging to dynamically detect an application phase change, our approach performs the repartitioning in every user-given interval, and the lock access counts are measured during the interval.

Repartitioning may adversely affect performance if a large number of thread migrations are initiated as a result. In order to prevent this, the repartitioning scheme enforces the maximum thread migration threshold on the KL algorithm. If the result from the last stage of the KL algorithm requires the thread migrations to be more than the threshold, the last stage will be discarded, and the previous stage result will be used as the final outcome. When the previous stage also requires a greater number of thread migrations than the threshold, the previous stage will be used. This will be repeated until a result that does not require thread migrations in excess of the threshold is found. If no stage matches the condition, then the last stage result, which was initially discarded, will be used instead. The optimal value for the maximum thread migration threshold can be deduced from a cost-benefit analysis. However, this is beyond the scope of this paper, so the threshold was arbitrarily set to a half of the total number of threads for the evaluation.

Additionally, the lock-thread repartitioning scheme further reduces the number of object faults. When two threads access

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

the same lock, they likely share data. Lock-thread repartitioning packs the threads acquiring the same lock into a single node. Therefore, when a thread accesses an unshared object that the thread has never before accessed, that unshared object possibly exists in the same node.

For garbage collection, a node in the field-based DSM marks the locally reachable objects in a bit vector and exchanges the bit vector with each other. In turn, each node marks the remotely reachable objects as such, and the newly marked objects are synchronized with the other end. After this, all unreachable objects are removed.

This distributed garbage collection mechanism generates significant overhead. However, with the aid of lock-thread repartitioning and object fault handling, distributed garbage collection is no longer necessary. Lock-thread repartitioning gathers as many threads as possible in a single node that shares the same objects. Thus, most of the necessary shared objects are locally reachable. Even when a remotely reachable object is removed from a node and then accessed later, the object fault handler is able to resolve the access.

C. Thread-state checkpointing

The proposed model is designed not only for casual applications but also for business or industrial applications. Consequently, abandoning context and restarting a process upon failure is unacceptable. To prevent these unbearable losses, our DSM model provides a failure recovery mechanism.

To perform failure recovery, it must maintain all required data to reconstruct threads lost by the failure.

Metadata, such as the thread locations and home nodes of lock objects, is located in a computing node, which is randomly chosen among all nodes at the first offloading point, to save the mobile device from offering metadata to the computing nodes. However, the mobile device still maintains a copy of it in case the metadata becomes unreachable due to a network or node failure.

Even if both the metadata and shared objects can be preserved upon node failure, the mobile device cannot reconstruct the threads of the failed node because the unshared objects and thread stacks of the failed node do not exist in the other nodes. In order to support fault tolerance, we propose a thread-state checkpointing scheme. When enabled, each node monitors and logs the write accesses to the fields of its unshared objects. Upon a synchronization event, regardless of the synchronization target, a node compresses and transfers the log together with the thread stacks to the mobile device. These checkpointing data will be used for thread reconstruction during the recovery process.

Unlike tracking, checkpointing is carried out only for unshared object accesses. Consequently, checkpointing does not require a locking mechanism for field accesses. Considering that the tracking overhead mostly stems from the use of a global lock of the JVM, the checkpointing overhead will be relatively negligible. Moreover, the mobile device does not need to reflect the logs to its address space and instead keeps them in their compressed form. Therefore, the processing and memory space overhead for thread-state checkpointing will be insignificant. However, as a log message will be delivered to the mobile device at every synchronization event, the network traffic and energy consumption may fairly increase.

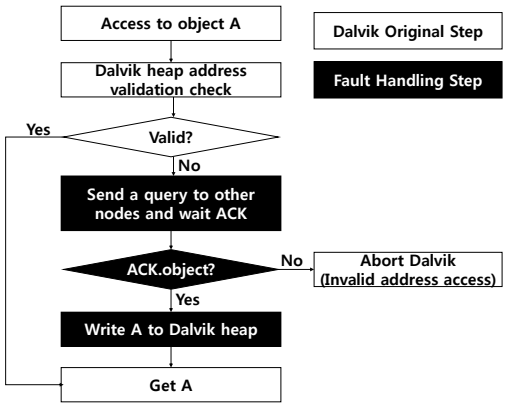


Fig. 6: Shared object fault handling procedure.

IV. Implementation

We implemented METEOR in the Dalvik VM in order to reuse the basic components, such as thread migration and τ-scheduler, of COMET. However, our approach can be easily applied to ART because no modifications were done in the machine code translation layer.

A. Multi-node support

When there are multiple computing nodes, a peer-to-peer style executable file transmission between a mobile device and computing nodes generates a large amount of network traffic from the mobile device. In METEOR, the representative of the computing node pool receives the *.dex* files, which are Android executable files, loaded in the mobile device during the first synchronization. The representative node, in turn, propagates the executable files to the other nodes in the pool. Once all computing nodes receive the .dex files, they become fully functional for the application instance.

When a thread running in a computing node needs to load a new .dex file, the Dalvik VM sends a query for the .dex file to all computing nodes. If a node possesses the .dex file, the owner will send the .dex file to the requesting node. If there are no nodes with the .dex file, the node directly downloads the file from the mobile device. In this way, the network traffic of the mobile device for the executable file transfer remains the same regardless of the number of computing nodes because a .dex file needs to be sent by the mobile device only once.

In order to support multiple computing nodes, a thread placement algorithm, which maps an offloaded thread to a computing node, is necessary. METEOR determines the location of threads at every lock-thread repartitioning. When the mobile device forks a new thread instance and offloads it, the node with the lowest computing load, which is determined during repartitioning, will accommodate the new thread.

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

B. Shared object management

For selective object tracking, we added two fields, the shared object flag and the thread ID of the last accessing thread, to *ObjectInfo*, which is a metadata structure for managing an object. The shared object flag is initially set to false. When a thread accesses a field in an object, the Dalvik VM compares the thread ID of the currently accessing thread with the last accessing thread ID of the object. If the currently accessing thread is not the one that accessed last, the shared object flag will be set to true and the object will be put into the tracked object set to be monitored from that moment on.

We modified the object access administration routine of the Dalvik VM for handling shared object faults. On the flowchart illustrated in Figure 6, the uncolored steps show the original object access procedure of the Dalvik VM and the shaded steps were added for shared object fault handling. The Dalvik VM checks whether the object address to be accessed is a valid address within the heap range. If it is not valid, the unmodified Dalvik VM will abort the execution. The modified Dalvik VM instead sends an object query to the mobile device and all computing nodes. When there is a node that holds a valid object at the given address, the node sends the object to the faulting node and also changes the status of the object to *shared*. The Dalvik VM aborts execution when no node has an object at the given address.

If threads are always designed to access a shared object with acquisition of a proper lock, our implementation guarantees that only one node responds to the object query because a shared object fault occurs exclusively for the first sharing access, and there must be only one node holding the object as its local object at that point. When threads access a shared object without using a mutual exclusion mechanism multiple nodes may send different copies of the same object. However, this does not violate the Java standard because the Java memory model employs the happens-before relationship [26], and the happens-before orders are determined by the mutual exclusion mechanisms.

C. Lock object identification

For the lock-thread repartitioning scheme, METEOR must be able to distinguish lock objects from other objects without developers' annotations. To help understand the lock object identification of METEOR, we first introduce the lock object management of the original Dalvik VM.

Dalvik manages lock objects using the thin lock notion [27]. In Java, any object can turn into a lock. An object becomes a thin lock when it is first used as a lock. When multiple threads try to acquire a thin lock, it inflates to become a fat lock. In a metadata structure for an object, Dalvik places a member variable, *lock*, to classify a lock object. Because a fat lock requires a mechanism to efficiently handle complicated locking requests, a monitor object is allocated and referenced by the *lock* variable for a fat lock.

Because a thin lock is associated with a specific thread, lock-thread repartitioning only counts fat locks as the locks to partition. This approach reduces the number of locks to partition, lowering the processing overhead of repartitioning, and simplifying the implementation at the same time.

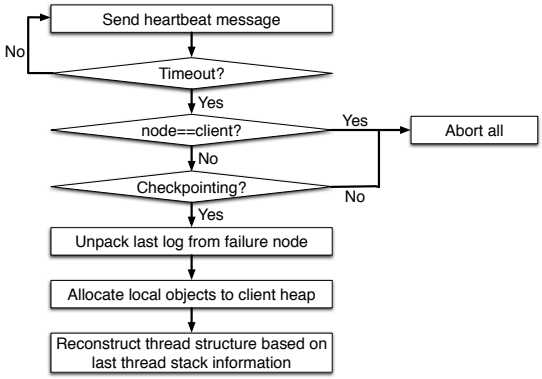


Fig. 7: Failure detection and recovery procedure.

D. Lock-thread repartitioner

Lock-thread repartitioning is conducted by the repartitioner. The repartitioner is the node that was assigned the least amount of computing load during the last repartitioning instance. Before the first repartitioning, the node with the metadata acts as the repartitioner. After the first repartitioning, the new repartitioner node takes charge of the metadata control because repartitioning requires intensive accesses to the metadata.

The repartitioning interval, p, is a configurable parameter, and the default value of p is 10 s. If the proportion of relocating threads from repartitioning is less than e, which is another configurable parameter with the default value of 20%, then the repartitioning interval is prolonged to double the current interval because the overhead may not sufficiently offset the performance gain. If the proportion of relocating threads exceeds e, the repartitioning interval is reset to p.

E. Fault detection and recovery

Figure 7 illustrates the fault detection and recovery procedure. In order to detect node failure, the repartitioner regularly sends heartbeat signals to all nodes, including the mobile device. If a node fails to respond to the heartbeat signal within a time-out interval, the repartitioner regards the node as a defunct node and notifies the mobile device.

If the defunct node is the mobile device or the thread-state checkpointing is disabled, all computing nodes immediately abort the execution of all threads. Otherwise, the mobile device initiates the recovery procedure.

For recovery, the mobile device reconstructs the thread stacks and unshared heap objects lost in the defunct node based on the compressed checkpoint data received from the last synchronization with the defunct node and resumes the execution of the reconstructed threads.

If the mobile device is alive and has not received any heartbeat signals for the time-out interval, the mobile device considers the repartitioner to be a defunct node. In this case, the mobile device will conduct the recovery process and will additionally appoint the node with the least amount of load to the repartitioner.

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

Operation	Mobile Dev.	Comp. Node
Synchronization	3.82%	2.95%
Repartitioning	0.00%	4.46%
Object fault handling	0.00%	0.01%
Object tracking	1.50%	2.78%
Checkpointing	1.31%	1.49%
Other overheads	3.29%	3.35%
Application	90.08%	84.96%

TABLE IV: Execution time composition of CPU Bench using METEOR with four computing nodes.

Application	Transferred Data (KB)		
	METEOR		FBDSM
	Default	+CHKPT	
Chess	284.9	1009.8	3582.4
Sudoku	28.1	683.4	1490.2
GO	128.2	1082.9	1679.5
Edge detect	9814.9	21096.1	23024.7
Face detect	10009.5	22081.8	27921.9
Fractal	51.1	910.3	2011.3
CPU Bench	124.5	1326.7	2249.9
Sphinx	3083.9	15494.7	20015.5

TABLE V: Network traffic generated with four computing nodes. (CHKPT and FBDSM denote checkpointing and field-based DSM, respectively)

V. Evaluation

A. Evaluation setup

Table II shows the system configurations for the mobile device and computing nodes. We used up to four computing nodes and a mobile device for evaluation.

The applications used in the evaluation are listed in Table I. All applications are available on Google Play, except Sphinx. We built Sphinx based on Sphinx4, an open source speech recognition engine [28].

To assess energy consumption, we measured the power input to the mobile device with a digital power meter. All results are average values from 10 repetitions of the experiments. Thread-state checkpointing is enabled unless otherwise stated.

The original COMET system does not support multiple computing nodes. For comparative evaluation with the field-based DSM, we implemented a simple thread placement algorithm for COMET that dispatches a thread with the thread ID of *TID* to the computing node with the node ID of *TID % (the number of computing nodes)*.

B. Overhead analysis

For the basic overhead analysis, we analyzed the composition of the execution time under METEOR with four computing nodes. Table IV shows the results of the execution time of CPU Bench.

Repartitioning mostly contributes to the overhead of a computing node, yet only approximately 4.5% of the total processing time. The thread migrations and synchronizations involved in repartitioning are counted as part of the repartitioning operation. Because large numbers of thread migrations and accompanying synchronizations occur for a repartitioning instance, a lock-thread repartitioning instance takes 10.0 ms on average. The overhead for both object tracking and checkpointing was negligible because thread-state checkpointing does not require lock acquisition for field accesses, as explained in Section III-C, and the object fault rarely occurred for the reason described in Sections III-A and III-B.

The synchronization overhead comprised the largest portion of the mobile device overhead. However, the overall synchronization overhead was significantly reduced because both the amount of synchronization data and the number of synchronization events were significantly reduced by the proposed schemes. Thread-state checkpointing expended less than 1.5% of processing time. The overhead not related to the consistency model was less than 3.5%, and most of this came from the metadata management.

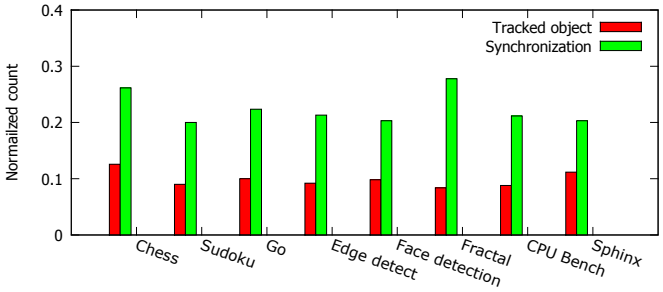


Fig. 8: Number of tracked objects and synchronization events normalized to field-based DSM.

The proportions of the actual code execution time in both the mobile device and the computing node were significantly higher than that of the field-based DSM, which was approximately half of the total execution time, even with a single computing node, as shown in Table III.

Table V shows the amounts of network traffic of the mobile device. METEOR was measured with and without thread-state checkpointing. The .dex files for applications were assumed to be cached in the computing nodes to clearly identify the communication from the DSM model.

Due to selective object tracking, the amount of data to be delivered during synchronization in METEOR was significantly smaller than that in the field-based DSM model. In addition, the field-based DSM model requires the mobile device to send all globally reachable objects, which add up to 700-800 KB during the first synchronization between the mobile device and each computing node, while METEOR does not need to send these thanks to the object fault handling mechanism.

Edge detect, Face detect, and Sphinx generated a significant amount of traffic because they deal with multimedia data. However, all other applications induced marginal amounts of network traffic. Although thread-state checkpointing dramatically increased network traffic, the amount of overall communication by METEOR was significantly smaller than that by the field-based DSM model.

978-1-5090-4327-9/17/$31.00 ©2017 IEEE

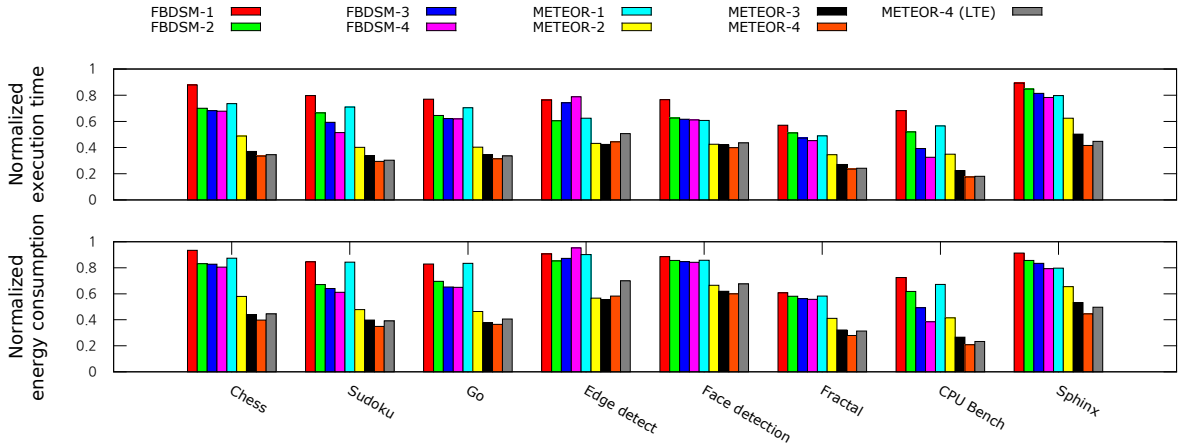


Fig. 9: Execution times (top) and energy consumption (bottom) using various configurations normalized to unmodified Android.

C. Performance analysis

In order to analyze the effectiveness of the selective object tracking and lock-thread repartitioning, we measured the number of synchronizations and the number of tracked objects while executing applications with four computing nodes. The results are shown in Figure 8. The shared objects were only 8.37% of the total objects on average, and the number of synchronizations was cut by more than 70% for all applications.

The normalized execution time and energy consumption under various configurations are shown in Figure 9.

Although, for single node experiments, repartitioning did not contribute to the performance; selective object tracking enabled METEOR to perform faster than the field-based DSM model in all cases.

With the exception of Edge detect, the performance of METEOR improved as the number of computing nodes grew, and the differences between one and two and between two and three computing nodes were notably larger than the corresponding results obtained from the field-based DSM. However, we found that the performance improvement saturated at four or fewer computing nodes for most applications. This is because the current applications were designed without considering computation offloading, and thus they cannot provide sufficient parallelism and a sufficient computation load. We believe that adaptive computing node provisioning based on the workload characteristics would reduce the possible inefficiency resulting from excessive computing nodes. We also believe that future applications will be aware of the computation offloading service and dynamically adjust the degree of parallelism depending on the circumstances.

The performance differences by network type were less than 3%, with the exception of Edge detect, Face detect, and Sphinx. This does not coincide with the existing research results [5], [11], [12], [18] in which a wireless network connection significantly degraded the performance and energy efficiency of the computation offloading service. This is because the high speed of the Long-Term Evolution (LTE) network partially offset the additional energy consumption. Based on this observation, we expect that the benefit from the computation offloading will naturally increase according to the development of wireless network technology.

The energy consumption results were similar to that of the execution time. However, the energy consumption relative to the execution time was high for Edge detect, Face detection and Sphinx because they induced a significant amount of network traffic, which consumed additional energy. Naturally, this tendency was stronger on LTE than WiFi.

To evaluate the failure recovery mechanism, we repetitively conducted fault-injection experiments. During the experiments, any points on the network, which connects the mobile device and computing nodes, were randomly disconnected at arbitrarily determined time points. The results showed that all applications were successfully recovered in all experiments.

We also measured the amount of time spent for the recovery process. Given a 4192 KB log from a disconnected node during the execution of Sphinx, the application was successfully resumed its execution after 1.58 seconds after the node failure.

VI. Conclusion

In spite of the many technical advantages, the DSM-based computation offloading approach misses a significant portion of the potential performance gain because the traditional DSM model is suboptimal for computation offloading.

This paper proposed an enhanced DSM model to support scalable, efficient and reliable computation offloading. Our evaluation showed that the proposed DSM model improved performance by up to 109% and reduced energy consumption by up to 52% compared to the previous DSM-based offloading scheme when four computing nodes were used.

VII. Acknowledgement

This research was supported by Basic Science Research Program (2015R1D1A1A0057749) funded by the Ministry of Education, and Research on High Performance and Scalable Manycore OS (B0101-15-0644) funded by the MSIP, Korea.

REFERENCES

[1] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya, "Mobile code offloading: from concept to practice and beyond," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80–88, 2015.

[2] F. Liu, P. Shu, H. Jin, L. Ding, J. Yu, D. Niu, and B. Li, "Gearing resource-poor mobile devices with powerful clouds: architectures, challenges, and applications," *IEEE Wireless Communications*, vol. 20, no. 3, pp. 14–22, June 2013.

[3] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: at the leading edge of mobile-cloud convergence," in *Proceedings of the 6th International Conference on Mobile Computing, Applications and Services (MobiCASE)*, 2014.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, 2012, pp. 13–16.

[5] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012, pp. 93–106.

[6] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile Cloud Computing: A survey," *Future Generation Computing Systems*, vol. 29, no. 1, pp. 84–106, Jan. 2013.

[7] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, 2002, pp. 217–226.

[8] R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. ACM, 2003, pp. 273–286.

[9] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, "Cuckoo: A computation offloading framework for smartphones." in *Proceedings of the 2nd International ICST Conference on Mobile Computing, Applications, and Services*, 2010.

[10] D. Kovachev, T. Yu, and R. Klamma, "Adaptive computation offloading from mobile devices into the cloud," in *Proceedings of the IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, July 2012, pp. 784–791.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010, pp. 49–62.

[12] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading." in *Proceedings of 2012 IEEE INFOCOM*, 2012, pp. 945–953.

[13] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS: Computation offloading as a service for mobile devices," in *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, ser. MobiHoc '14. ACM, 2014, pp. 287–296.

[14] V. March, Y. Gu, E. Leonardi, G. Goh, M. Kirchberg, and B. S. Lee, "Cloud: Towards a new paradigm of rich mobile applications," *Procedia Computer Science*, vol. 5, no. 0, pp. 618 – 624, 2011.

[15] X. Zhang, S. Jeong, A. Kunjithapatham, and S. Gibbs, "Towards an elastic application model for augmenting computing capabilities of mobile platforms," in *Proceedings of the Third International ICST Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, 2010.

[16] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[17] M. Kristensen, "Scavenger: Transparent development of efficient cyber foraging applications," in *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, 2010, pp. 217–226.

[18] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," in *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011, pp. 301–314.

[19] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, 1992, pp. 13–21.

[20] W. Zhu, C.-L. Wang, and F. Lau, "JESSICA2: A distributed Java virtual machine with transparent thread migration support," in *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2002, p. 381.

[21] Valgrind Developers, "Valgrind user manual," http://valgrind.org/docs/manual/manual.html.

[22] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Computer Society, 2002, p. 43.

[23] H. Saran and V. V. Vazirani, "Finding k-cuts within twice the optimal," in *Proceedings of the 32nd annual symposium on foundations of computer science*. IEEE Computer Society, 1991, pp. 743–751.

[24] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[25] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[26] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java language specification: Java se 8 edition," Oracle America Inc., 2015.

[27] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano, "Thin locks: Featherweight synchronization for Java," *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 258–268, 1998.

[28] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, "Sphinx-4: A flexible open source framework for speech recognition," Sun Microsystems, Inc., Tech. Rep. SMLI TR-2004-139, 2004.

978-1-5090-4327-9/17/$31.00 ©2017 IEEE