COMET: Code Offload by Migrating Execution Transparently

Mark S. Gordon† D. Anoushe Jamshidi† Scott Mahlke† Z. Morley Mao† Xu Chen⋆

†EECS Department,
University of Michigan
Ann Arbor, MI, USA
{msgsss,ajamshid,mahlke,zmao}@umich.edu

⋆ AT&T Labs - Research
chenxu@research.att.com

Abstract

In this paper we introduce a runtime system to allow unmodified multi-threaded applications to use multiple machines. The system allows threads to migrate freely between machines depending on the workload. Our prototype, COMET (Code Offload by Migrating Execution Transparently), is a realization of this design built on top of the Dalvik Virtual Machine. COMET leverages the underlying memory model of our runtime to implement distributed shared memory (DSM) with as few interactions between machines as possible. Making use of a new VM-synchronization primitive, COMET imposes little restriction on when migration can occur. Additionally, enough information is maintained so one machine may resume computation after a network failure.

We target our efforts towards augmenting smartphones or tablets with machines available in the network. We demonstrate the effectiveness of COMET on several real applications available on Google Play. These applications include image editors, turn-based games, a trip planner, and math tools. Utilizing a server-class machine, COMET can offer significant speed-ups on these real applications when run on a modern smartphone. With WiFi and 3G networks, we observe geometric mean speed-ups of 2.88X and 1.27X relative to the Dalvik interpreter across the set of applications with speed-ups as high as 15X on some applications.

1 Introduction

Distributed Shared Memory (DSM) systems provide a way for memory to be accessed and modified between computing elements. It was an active area of research in the late 1980s. Classically, DSM has been applied to networks of workstations, special purpose message passing machines, custom hardware, and heterogeneous systems [18]. With the onset of relatively low performance smartphones, a new use case for DSM has presented itself. In our work, we apply DSM to offloading – the task of augmenting low performance computing elements with high performance elements.

Offloading is an idea that has been around as long as there has been a disparity between the computational powers of available computing elements. The idea has grown in popularity with the concept of ubiquitous computing where many low-powered, well-connected computing elements would exist that could benefit from the computation of nearby server-class machines. The popular approach to this problem is visible in specialized systems like Google Translate and Apple iOS's Siri. For broad applicability, COMET and other recent work [9, 8] have aimed to generalize this approach to enable offloading in applications that contain no offloading logic. These systems when compared to specialized offloading systems can offer similar benefits that compilers offer over hand-optimized code. They can save programmer effort and in some cases outperform many specialized efforts. We believe our work is unique as it is the first to apply DSM to offloading. Using DSM instead of remote procedure calls (RPC) offers many advantages including full multi-threading support, migration of a thread at any point during its execution, and in some cases, more efficient data movement.

However, in applying DSM, we faced many challenges unique to our use case. First, the latency and bandwidth characteristics of a smartphone's network connection to an external server are much worse than those of a cluster of workstations using a wired connection. Second, the type of computation is significantly different: while previous DSM systems focused on scientific computing, we aim to augment real user-facing applications with more stringent response requirements. Despite these challenges, we show in §5 that performance improvements can be significant for a wide range of applications across both WiFi and 3G networks.

Our design aims to serve these primary goals:

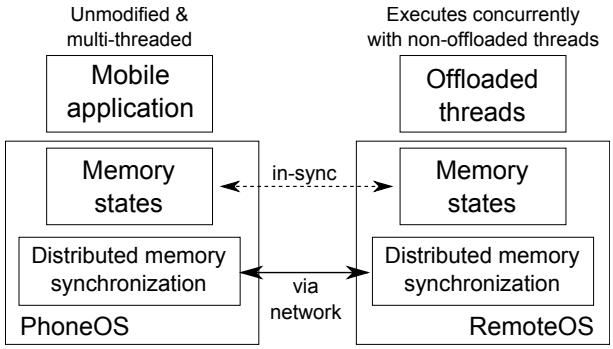


Figure 1: High-level view of COMET's architecture

- Require only program binary (no manual effort)
- Execute multi-threaded programs correctly
- Improve speed of computation
- Resist network and server failures
- Generalize well with existing applications

Building upon previous work, our work offers several new contributions. To the best of our knowledge, we propose a new approach to lazy release consistency DSM [20] that operates at the field level granularity to allow for multiple writers of the same memory unit. This allows us to minimize the number of times the client and server need to communicate to points where communication *must* occur. Using DSM also allows us to be the first offloading engine to fully support multi-threaded computation and allow for threads to move between endpoints at any interpreted point in their execution rather than offloading only entire methods. Despite these features, COMET is designed to allow computation to resume on the client if the server is lost at any point during the system's execution. Finally, we propose a very simple scheduling algorithm that can give some loose guarantees on worst case performance. Figure 1 gives an overview of the design of the system.

We implemented COMET within the Dalvik Virtual Machine of the Android Gingerbread operating system and conducted tests using a Samsung Captivate phone and an eight core server for offloading. Our tests were comprised of a suite of nine real-world applications available on Google Play including trip planners, image editors, games, and math/science tools. From our tests, we found that COMET achieved a geometric mean speedup of 2.88X and average energy savings of 1.51X when offloading using a WiFi connection. 3G did not perform as well with a geometric speedup of 1.27X and a modest increase in energy usage. With lower latency and higher bandwidth, we expect better offloading performance for 4G LTE networks compared to 3G.

The rest of the paper is organized as follows. In §2, we summarize works in the field of DSM and offloading and discuss how they relate to COMET. §3 presents the overall system design of COMET. §4 describes how we implemented COMET using Android's Dalvik Virtual Machine. §5 presents how we evaluated COMET including the overheads introduced by the offloading system, performance and energy improvements for our benchmark suite, and two case studies demonstrating some interesting challenges and how COMET solves them. §6 covers some limitations of our system, and §7 ends with final remarks.

2 Related Work

A significant amount of work has gone into designing systems that combine the computation efforts of multiple machines. The biggest category of such systems are those that create new programming language constructs to facilitate offloading. Agent Tcl [14] was one such a system that allowed a programmer to easily let computation "jump" from one endpoint to another. Other systems that fall into this broad category include Emerald [19], Network Objects [5], Obliq [6], Rover [17], Cuckoo [21], and MapReduce [11]. Other work instead focuses on specific kinds of computation like Odessa [26] which targets perception applications, or SociableSense [27] designed for harvesting social measurements. COMET takes an alternative approach by building on top of an existing runtime and requiring no binary modifications.

Other related offloading systems include OLIE [15], which applied offloading to Java to overcome resource constraints automatically. Messer et al. [23] proposed a system to automatically partition application components using MINCUT algorithms. Hera-JVM [22] used DSM to manage memory consistency while doing RPC-style offloading between cores on a Cell processor. Helios [25] was an operating system built to utilize heterogeneous programmable devices available on a device.

Closer to our use case, MAUI [9] enabled automated offloading and demonstrated that offloading could be an effective tool for performance and energy gains. In their system, the developer was not required to write the logic to ship computation to a remote server; instead he/she would decide which functions *could* be offloaded and the offloading engine would do the work of deciding what *should* be offloaded and collecting all necessary state. Requiring annotation of what could be offloaded limited the reach of the system leaving room for CloneCloud [8] to extend this design, filling this usability gap by using static analysis to automatically decide what could be offloaded. Other works, like ECOS [13], attempt to address

the problem of ensuring data privacy to make offloading applicable for enterprises.

JESSICA2 implemented DSM in Java for clusters of workstations. JESSICA2 implemented DSM at the object level using an object homing technique. This approach is not suitable for our use case because each synchronization operation triggers a network operation. Moreover it is not robust to network failures.

Munin [7] was one of a couple early DSM prototypes built during the 90s that targeted scientific computation on workstations. Shasta [29] aimed to broaden the applicability of software DSM by making a system that worked with existing applications. These were followed up by systems like cJVM [4] and JESSICA2 [30] which brought together DSM and Java. However these systems are constructed for low latency networks where the cost communication is relatively low encouraging designs with more communication events and less data transfer. Additionally, as is the case with cJVM and JESSICA2, the DSM design is not conducive to failure recovery.

Our work can be seen as a combination of the efforts of these DSM systems with the offloading frameworks of Maui and CloneCloud. Unlike much of the work on offloading discussed above, we focus not on *what* to offload but more on the problem of *how* to offload. COMET makes use of a new approach to DSM to minimize the number of communication events necessary while still supporting multi-threaded applications. We attempt to demonstrate COMET's applicability by evaluating on several existing applications on Google Play.

3 Design

This section contains the overall design of COMET to meet the five goals listed in §1. Although some specific references to Java are made, the design of COMET aims to be general enough to be applied to other managed runtime environments (such as Microsoft's Common Language Runtime). Working with a virtualized runtime gives the additional benefit of allowing the use of heterogeneous hardware.

To facilitate offloading we use DSM techniques to keep the heap, stacks, and locking states consistent across endpoints. These techniques together form the basis of our distributed virtual machine, allowing the migration of any number of threads while being consistent with the Java Memory Model. Our DSM strategy can be considered both lazy and eager – lazy in the classic sense that our protocol acts on an acquire, and eager in the sense that we eagerly transmit all dirtied data when we do act. This strategy is useful for reducing the frequency of required communication between endpoints, necessary to deal with high latency between endpoints that often exists in wireless networks.

While it appears our design could be extended to more than two endpoints, we discuss how operations work specifically for the two endpoint case that our prototype supports. Our intended endpoints are one client (a phone) and one server (a high-performance machine). However, the design rarely needs to identify which endpoint is which and our work could be used in principle to combine two similarly powered devices.

3.1 Security

Under our design a malicious server can take arbitrary action on behalf of the client process being offloaded. Additionally we have no mechanism to ensure the accuracy of the computed data nor the privacy of the inputs provided. Therefore it seems necessary that the offload server be trusted by the client.

However there is no need for the server to trust the client. In this initial design the server only grants access of its computation resources to the client and it has no private data to compromise or dependency on the accuracy of computation.

3.2 Overview of the Java Memory Model

The Java Memory Model plays an important role in our design dictating what data must be observed by a thread. In the JMM, memory reads and writes are partially ordered by a transitive "happens-before" relationship. A read must observe a write if the write "happens-before" the read. The Java specification also supports threads and locks which directly tie into the JMM.

Within a single thread, all memory operations are totally ordered by which happened first during execution. Across threads, release consistency is used; when thread *A* acquires a lock previously held by thread *B*, a "happens-before" relationship is established between all operations in thread *B* up to the point the lock was released and for all future memory operations of thread *A*, meaning whatever writes thread *B* made before releasing the lock should be visible to thread *A*. Other miscellaneous operations like volatile memory accesses or starting a thread, can also create "happens-before" relationships between accesses in different threads.

Still there remain cases where there may be several writes that a read could observe. These situations are called data races and usually indicate a buggy program. This situation, however, is occasionally intentional. For example, let W be the set of writes that "happen-before" the read. Then the VM may let the read observe any maximal element of W or any write not ordered with respect to the read.

An existing method to follow the JMM while offloading is to offload one thread at a time and block all other local execution before the offloaded thread returns [9]. This prevents multiple threads from making progress simultaneously, reducing the benefit of offloading. This also eliminates the possibility of offloading computation (e.g., a function) that calls synchronization primitives unless it can be checked automatically that the code will not deadlock.

A slight modification is to let local threads continue executing, until they access shared state across threads [8]. This again limits the usefulness of offloading and, for example, prevents offloading a thread that may grab a lock later on. Additionally, this kind of scheme can introduce deadlocks into the program's execution if the offloaded thread tries to wait for data from another thread. COMET overcomes these limitations by relying on DSM and VM-synchronization techniques to keep the distributed virtual machine in a consistent state without limiting what can execute at any time.

3.3 Field-based DSM

Our key contribution over past DSM systems is the use of field level granularity to manage memory consistency. By doing things at this granularity, we can avoid tracking anything more than a single bit indicating the dirtiness of each field. Our DSM mechanism allows for multiple readers and writers to simultaneously access the same field without communication.

This is possible with DSM for Java, but not for other less-managed runtimes, because reads and writes can only happen at non-overlapping memory locations of known widths. In particular, this means we can always *merge* changes between two endpoints, even if they have both dirtied a field, by just selecting one of the writes to "happen-before" the other as described in §3.2. If instead we worked at a coarser granularity, it would be unclear how to merge two dirty memory regions. Even a copy of the original memory region does not allow us to merge writes without a view into the alignment of fields in memory. This is particularly important in Java where values must not appear "out of thin air."

3.4 VM-Synchronization

At the heart of our design is the directed VM-synchronization primitive between two endpoints; the pusher and the puller. In full, the primitive synchronizes the states of the virtual heap, stacks, bytecode sources, class initialization states, and synthetic classes.

Synchronizing the virtual heap is accomplished by tracking dirty fields of objects discussed in §3.3. During a VM-synchronization, the pusher sends over all of the dirty fields it has in the shared heap. The puller then reads in the changes and overwrites the fields sent by the pusher. Both endpoints mark the fields involved as clean.

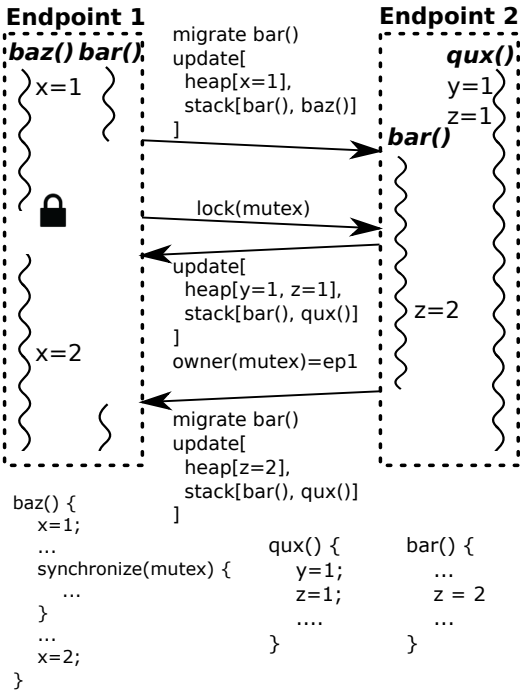


Figure 2: At the beginning, baz() and bar() are running in separate threads on ep1, while qux() is running as a different thread on ep2. ep2 holds the ownership of mutex at the beginning, but no thread lock on it.

Stack synchronization then is done by sending over the stacks of any shared threads that are running locally. This includes each method called, the program counter into each method, and any method level registers. This encodes enough information about the thread's execution state that the puller could resume execution. This makes the act of migrating a thread trivial after a VM-synchronization.

An example of the primitive's operation is shown in Figure 2. In the example, *ep1* pushes one VM-update (indicated by "update[...]" in the diagram) to *ep2*, and two VM updates go from *ep2* to *ep1*. The pusher operates by assembling a heap update from all changes to the heap since the last heap update and an update to each locally running stack. The puller then receives the update and merges the changes into its heap and updates the stacks of each received thread. Note that the pusher's heap and thread stacks do not change as a result of this process. For example in the third VM-synchronization shown in Figure 2, only the modification to z needs to be sent because x and y have not changed since the previous update from *ep2* to *ep1*.

This primitive is our mechanism for establishing a "happens-before" relationship that spans operations on different endpoints. This includes each of the situations mentioned in §3.2. Any operations that need to establish a "happens-before" relationship, such as migrating a thread or acquiring a lock when it was last held elsewhere, will use this operation.

3.5 Locks

From §3.2 we found that a "happens-before" relationship needs to be established whenever a lock is acquired that was last held on another thread. To solve this problem we assign an owner to each lock, indicating the endpoint that last held the lock. In Java, any object can be used as a lock. Thus, each object is annotated with a lock-ownership flag. When attempting to lock an object that an endpoint does not own, the requesting thread can simply be migrated or the endpoint can make a request for ownership of the lock. Which choice should be followed is dictated by the scheduler discussed in §3.7.

In the latter case, the other endpoint will usually respond with a VM-update and a flag indicating ownership has been transferred. Figure 2 demonstrates this behavior when *baz()*, running on *ep1*, attempts to lock on the object *mutex* that is originally owned by *ep2*. This causes a VM-update to be sent to *ep1* as well as ownership of the *mutex* object.

Java also supports a form of condition variables that allow wait and signaling of objects. This comes almost for free because waiting on an object implies you hold a lock on the object. When you wait, the lock is released and execution is suspended until the object is signaled to continue. Then the lock is re-acquired, which will cause the appropriate synchronization, if required. COMET only needs to, in some situations, send a signal to wake up a waiting remote endpoint. Some additional tracking of how many threads are waiting on a given condition variable is maintained to serve this purpose.

Volatiles are handled in a similar fashion to locks. Only one endpoint can be allowed to perform memory operations on a volatile field at a time. This is stronger synchronization than what is required, but it suffices for our design. Fortunately volatile operations are fairly rare, especially in situations when offloading is desirable.

3.6 Native Functions

In §3.4 and §3.5, we have described the key parts of our offloading system. Still, we have to decide what computation to offload. The obvious question is why not offload everything? This works fine until a native function is encountered. These are functions usually implemented in C with bindings to be accessed from Java. COMET cannot, in general, offload these functions. These functions are usually implemented in native code for one of the following three reasons: (1) reliance on phone resources (file system, display), (2) performance, and (3) reliance on readily-available C libraries.

Each of these cases provides its own challenges if you wish to allow both endpoints to run the corresponding function. It is very common for these functions to rely on hidden native state, frequently making this a more difficult task than it might first appear. So, in general, we assume that native methods cannot be offloaded unless we manually mark them otherwise. As applications rarely include their own native libraries [12], we can do this once as a manual effort for the standard Android libraries. Indeed, we have manually evaluated around 200 native functions as being suitable to run on any endpoint.

Additionally, to support failure recovery we need to be even more careful about what native methods are allowed to run on the server. Methods that modify the Java heap or call back into Java code can be dangerous because if a VM-synchronization happens while the native method is executing the client has no way to reconstruct the state hidden on the native stack of the partially executed native method. Some native methods warrant a special exemption (e.g. Java's reflection library) . For other non-blocking native methods, we may simply force a pending VM-synchronization to wait for the method to exit before continuing.

3.7 τ-Scheduling

The last component of the system is the scheduler. Analogous to schedulers used by modern operating systems, the scheduler is charged with the task of moving threads between endpoints in an attempt to maximize throughput (alternative goals are possible as well). During a push, the scheduler decides which local threads should be migrated and which non-essential lock/volatile ownership flags should be transferred. Additionally, the scheduler should initiate a push when it wants to migrate a thread or to avoid requesting ownership of a lock.

For our first iteration of a solution to this problem, we have used a basic scheduler that relies on past behavior to move threads from client to server where they remain as long as possible. This is achieved by tracking how long a thread has been running on the client without executing client-only code (a native method) and migrating the thread when this time exceeds τ, where τ is some configurable parameter. For our prototype, we initially choose τ to be twice the round trip time (RTT). Over the execution of the application, τ is replaced with twice the average VM-synchronization time.

This choice of τ has the nice property of limiting

the damage of mistakes to twice the runtime without offloading. This is because we need to have made forward progress for a time of at least τ before migrating. If we immediately had to migrate back, we have selected τ so that it should take a time of τ before we are running locally again. Thus the τ parameter lets us tune risk and potential benefit of the scheduler.

4 Implementation

We built COMET by extending the Dalvik Virtual Machine (DalvikVM) targeted for the Android mobile operating system. The DalvikVM is an interpreter for the Dalvik byte-code based on the Java VM specification. Our code was written on top of the CyanogenMod's Gingerbread release. While the DalvikVM is intended to be run on Android, it can be compiled for x86 architectures and run on common Linux distributions. Because the JIT was not yet available on x86, we were forced to disable the JIT for our prototype. The additions to the code-base sum to approximately 5,000 lines of C code with the largest component, the one managing most DSM operation, at 900 lines of code.

4.1 Threads and Communication

To effectively handle multi-threaded environments, threads are virtualized across different endpoints. The DalvikVM uses kernel threads to run each Java thread. In our system, kernel threads are paired between endpoints to represent and act on behalf of a single virtualized thread. Figure 3 shows pairs of parallel threads that represent a single thread virtualized across endpoints.

To facilitate operations across endpoints, a full-duplex byte stream connects two parallel threads. This abstraction is useful because all of the communication can be expressed easily between parallel threads. Figure 3 shows the path that data travel to get from a thread to its corresponding parallel thread. When a thread has data to write, it sends a message to the controller. The controller then multiplexes the data being written by all threads over a single TCP connection, applying a compression filter to save bandwidth and time. The remote controller will demultiplex and decompress the message, sending the data to the kernel thread parallel to the sender. This allows for multiple byte-streams while avoiding the need to establish multiple connections between endpoints.

To allow messages to be demultiplexed, virtual threads are assigned IDs (this is the same identifier *Thread.getId()* will return). When the controller receives a message, it can use the ID to look up which thread to deliver the message to or create the thread if it does not already exist. When assigning IDs we set the high bit of the ID with the

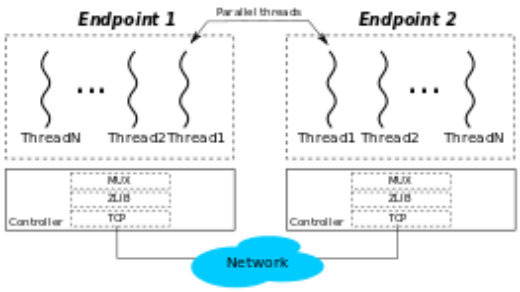


Figure 3: Communication between endpoints

endpoint ID that created the thread to avoid ID collisions. When a thread exits, it sends a message to the parallel thread so that the thread may exit on both endpoints (and joins so the thread can complete) and the IDs may be released.

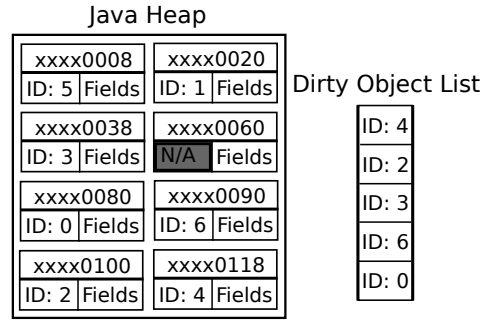
4.2 Tracked Set

In §3.4, we mentioned that updates only need to be sent for objects that can be accessed by both endpoints. To identify such objects, we introduce the the notion of the tracked set of objects.

The tracked set contains all class objects and is occasionally updated to include other local objects, for example objects present on a stack during a VM-synchronization. Global fields are considered to be part of the class object they are defined in, and thus are included in the tracked set. Additionally during a push operation, the tracked set is closed, meaning that all objects reachable from objects in the tracked set are added to the tracked set as well. See §4.5 for how tracked objects can eventually be garbage collected.

To support our DSM design, every tracked object is annotated with a bitset indicating which fields are dirty. Figure 4 illustrates how this data is stored and accessed. Each write to a field of a tracked object causes that field to be marked as dirty. Additionally, to be able to quickly find only the objects with dirty fields, a dirty object list is maintained, to which an object is added the first time one of its fields is made dirty. As a result, it becomes easy for a push operation to find all updated fields and add untracked objects that appear in those modified fields to the tracked set. After a push, all of the dirty bits are cleared and the dirty object list is emptied.

Similar to existing systems [9, 8], COMET assigns IDs to objects in the tracked set. This allows each endpoint to talk about shared objects in a coherent way, as pointers are meaningless across endpoints. For efficiency reasons, objects are assigned incremental IDs when added to the tracked set, so an object lookup turns into an array lookup into the tracked set table as shown in Figure 4. Similar



Tracked Set Table

ID	Address	Lock owned	Volatile owned	Object Dirty	Field 0 dirty	Field 1 dirty	•••
0	xxxx0080		X	X	X	X	
1	xxxx0020	X					
2	xxxx0100	X	X	X	X		
3	xxxx0038	X	X	X		X	
4	xxxx0118			X		X	
5	xxxx0008	X	X				
6	xxxx0090	X	X	X	X	X	
7	null						

Figure 4: Tracked Set Table

to thread ID assignment, the high order bit of object IDs is filled with the endpoint ID to avoid conflict so both endpoints can add objects to the tracked set. The overhead associated with tracking field writes and maintaining the tracked set structures is examined in §5.2.

4.3 VM-Synchronization

Now we are ready to explain the core of our design, the VM-synchronization operation. VM updates are always performed between parallel threads with one endpoint sending updates, the pusher, and the other endpoint receiving the updates, the puller. There are three major steps to each synchronization operation.

First, the pusher and puller enter into an executable exchange protocol where any new bytecode sources that have been loaded on the pusher are sent over to the puller. Usually, this step is just as simple as the pusher sending over a message indicating that there are no new binaries. Otherwise, the pusher sends over a unique binary identifier, so the puller can try to look up the binary in its cache. If it is not found, the puller can request the entire binary which will be stored it in its cache for future use.

Second, the pusher sends over information about each thread. To simplify this operation, all locally running threads are temporarily suspended. We can track what portion of the stack needs to be resent since the last update and send each frame higher up on the call stack. Each interpreted frame contains a method, a program counter, and the registers used by that method. Using register maps produced by the DalvikVM, we can detect which registers are objects and add them to the tracked set. In addition to the stack information, we also transmit information about which locks are held by each thread. This enables the thread on the puller to acquire any activated locks in addition to allowing functions like *Thread.holdsLock()* to execute properly without any communication.

Finally, the pusher sends over an update of the shared heap. It goes through the dirty object list, finds which fields have changed, and sends the actual modifications to those fields. If we attempt to transmit an object field, the referenced object is added to the tracked set and the ID of that object is transmitted. Otherwise an endian neutral encoding of the field is transmitted. Lastly, it clears all of the dirty flags and the dirty object list and resumes all other threads. Performance data on how long this takes and how much data needs to be sent is available in §5.3.2.

After the executable exchange, the puller first buffers the rest of the VM-synchronization. Then it temporarily suspends each of the local threads as done during the push. It then merges in the update to the heap first. This often involves creating new objects that have not yet been seen or could involve writing a locally dirty field. In the latter case the JMM gives us liberty to use either value in the field but we choose to overwrite with the new data so the dirty bit can be cleared. After this, the puller reads in the changes to the stack, again using the register maps from the DalvikVM to convert any object registers from their IDs to their corresponding object pointers. After this completes, the puller can again resume threads locally. Additionally, heap updates are annotated with a revision ID, so that they can be pulled in the same order they were pushed.

As an example, take Figure 4 as the initial state of the pusher. The heap update will contain five object definitions corresponding to the five dirty objects. If we assume that each of these objects only has the two shown fields, then in total seven field updates will be sent out. If one of those fields was an object field that pointed to `xxxx0060`, it would be assigned ID 7, and instead we would send out six object definitions and nine field updates. In either case, after the heap update, the pusher's dirty object list will be empty and all of the dirty flags are cleared.

4.4 Thread Migration

Next we can discuss the operations built on top of the VM-synchronization. In the core of the offload engine is the thread loop. At most instances in time, at least one of a pair of parallel threads is waiting in the thread loop for a message from its peer. These messages are used to initiate the actions described below. Additionally, a

special *resume* message can be sent to tell the thread to exit the message loop and continue on with its previous operation.

The primary operation is the *migrate* operation which indicates that the requesting thread wishes to migrate across endpoints. The VM-synchronization handles most of the challenges involved. In addition, an *activate* and *deactivate* operation is performed on the puller and pusher respectively. *Activate* is responsible for grabbing any locks and setting lock-ownership of objects that are currently held in the execution of the thread. After that, the thread either calls directly into an interpreter or falls back into the interpreter that called into the thread loop depending on where the thread now is in its execution. *Deactivate* is comparatively simpler and just needs to release any held locks and ownership over them.

Transferring lock ownership is another important operation. As discussed in §3.5 and shown in Figure 4, each object is annotated with a lock ownership flag so that a "happen-before" relationship can be established correctly when needed. Initially each object is owned by the endpoint that created it. When a thread attempts to lock an object its endpoint does not own, it needs to request ownership of the object from the remote endpoint (or alternatively the scheduler could decide to migrate the thread). It does so by sending a *lock* message to the other endpoint along with the object ID it wishes to gain ownership of. The parallel thread wakes up and responds either with a heap update if the endpoint still owns the object or a failure message if ownership has already been transferred (i.e., by another virtual thread). In case of failure, the initial thread will simply repeat the entire process of trying to lock the object again. Special care must be taken so that exactly one endpoint always owns the lock, with the small exception of when ownership is being transferred and nobody owns the lock.

Volatiles are handled in much the same way as locks. In addition to the lock ownership flag annotated to objects, there is also a volatile ownership flag. This flag mirrors the lock flag so that a "happens-before" relationship is established when an endpoint that does not hold volatile ownership of an object needs to access a volatile field of that object.

4.5 Garbage Collection

The tracked set as described so far will keep growing indefinitely. Occasionally, some of the shared state will no longer be needed by either endpoint and it can be removed. To resolve this issue we have a distributed garbage collection mechanism. This mechanism is triggered after a normal garbage collection has failed to free enough memory and the VM is about to signal it is out of memory.

To begin distributed garbage collection, both endpoints *mark* every object that is reachable locally. Then a bitvector indicating whether each tracked object is locally reachable is sent to the remote endpoint. Receiving this bitvector from the all remotely *marked* objects are *marked* locally in addition to any other objects reachable. If any new objects are marked this way on either endpoint the bitvectors must be sent again. This process continues until both endpoints have agreed on which tracked objects are reachable. In pathological cases this process could take quite a few round trips to converge. In these cases the client could just disconnect and initiate failure recovery.

4.6 Failure Recovery

From the design of COMET failure recovery comes almost for free. To properly implement failure recovery, however, the client must never enter a state that it could not recover from if no more data was received from the server. Therefore each operation that can be performed must either wait for all remote data to arrive before committing any permanent changes (e.g. data is buffered in the VM-synchronization) or the change must be reversible if the server is lost before the operation completes. To recover from a failure the client needs only resume all threads locally and reset the tracked object set.

In the case of resuming execution, the server is required with each synchronization to send an update of all of the thread stacks. This way if the server is lost it has the necessary stack information to resume execution. The client, however, needs only to send stacks of threads it is attempting to migrate. Detection of server loss at this point is simple. If the connection to the server is closed or if the server has not responded to a heartbeat soon enough.

5 Evaluation

In this section we evaluate the overheads of COMET (§5.2), and the performance and energy improvements that COMET enables for a set of applications available on Google Play as well as one hand-made computation-intensive benchmark (§5.3). In §5.4 we take a deeper look at how COMET works in some unique situations as a series of short case studies.

5.1 Methodology

We tested COMET on a Samsung Captivate smartphone running the Android 2.3.4 (Gingerbread) operating system. Because the phone has some proprietary drivers, we implemented COMET within a CyanogenMod [10] build of the Android operating system. Our server is a 3.16GHz, eight core (two quad core Intel Xeon X5460 processors) system with 16GB of RAM, running Ubuntu

μBenchmark	T & S	T & !S	!T & S	!T & !S
write-array	30.9%	30.4%	8.96%	7.58%
write-field	36.7%	35.1%	10.6%	9.17%
read-array	8.17%	6.05%	9.71%	5.75%
func-read	12.2%	9.26%	9.76%	9.58%

Table 2: Overheads of COMET relative to an uninstrumented client. Results are shown for when heap tracking(T) and the scheduler(S) are enabled/disabled. An exclamation mark indicates that the specified function is disabled.

11.10. COMET was tested using 802.11g WiFi in the Computer Science and Engineering building at the University of Michigan, as well as a real 3G connection using AT&T's cellular network. To gather energy data, we used a Monsoon Power Meter [24], which samples the battery voltage and current every 200μs to generate a power waveform. We integrated these power waveforms over time to collect our energy readings.

We evaluated COMET using nine real applications from Google Play that are diverse in their functionality, but all have non-trivial computation that may benefit from offloading. Table 1 lists the package names and their functionality. Additionally, two purely computational applications were chosen, Linpack (available from Google Play) and Factor (hand-coded), to highlight the capabilities of COMET. Factor features multi-threaded execution to illustrate COMET's capability to offload multiple threads simultaneously, which will be discussed in detail in §5.4.

In order to create repeatable tests, we used the Robotium 3.1 user scenario testing framework [28]. This framework allowed us to script user input events and reliably gather timing information for all of our benchmarks. All test data has been gathered from five test runs of each benchmark for each network configuration.

5.2 Microbenchmarks

We test the overheads of our COMET prototype with four microbenchmarks: *Write-array* that writes to an array in one linear pass, *Write-field* that writes to one index in an array, *Read-array* that reads an entire array in one linear pass, and *Func-read* that performs array reads through many small function calls.

Table 2 displays the results of these microbenchmarks. All results are relative to a client running an uninstrumented Dalvik VM using its portable interpreter. Results are shown for all combinations of when heap tracking or the scheduler are enabled or disabled.

When performing heavy writes to an object the write tracking code is triggered and causes performance degradation, which is shown by the *write-array* and *write-field* tests when tracking is enabled. There is never any tracking of reads so the overhead in *read-array* and *func-read* gives the overheads from modifications to the interpreter in those cases.

While these overheads seem significantly high, these microbenchmarks represent worst case scenarios that are unlikely to appear when running real applications. The high costs can be offset by benefits from computation speed-ups and may be reducible with more work.

5.3 Macrobenchmarks

This section discusses high level tests of the applications listed in Table 1. We have divided each application's computation into "UI" and "Computation" components based on which portions of the applications we believe are doing interesting computation. The input events used to operate our tests do not come from actual user traces and in some cases include additional delays to ensure synchronization. Therefore the ratio of "UI" to "Computation" time means very little but we still present the "UI" times to give some indication of how these applications might be used and because the "UI" portion of execution is difficult to discount in our energy measurements.

Moreover our application suite and the functionality exercised in our tests were not chosen based on any real user data. Therefore these tests serve only as an indication that our system can work on some applications in the wild. A user study is required to get some metric on how well COMET can do in the wild when the system has matured.

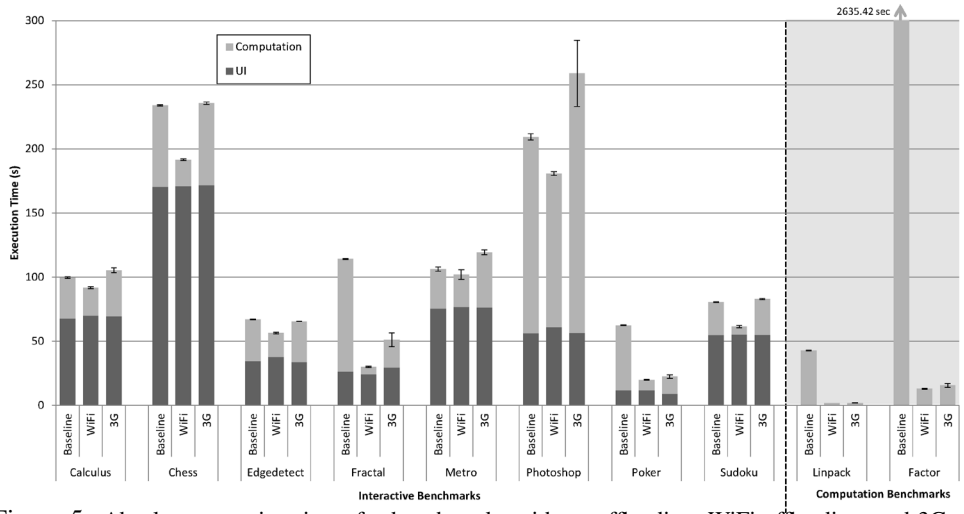
5.3.1 Performance and Energy Benefits

The primary goal of COMET is to improve the speed of computation. As a side-effect, we also expect to see improvements in energy-efficiency if computation-intensive portions of code are executed remotely on a server. Because there is no standard set of benchmarks in this area of research and readers may be unfamiliar with the applications chosen as benchmarks, we present our performance results as absolute times in seconds and energies in Joules in Figures 5 and 6. Each figure also shows the computation speed-ups and energy efficiency improvements relative to the same benchmarks run without offloading enabled.

Figure 5 shows that COMET achieves significant computation speed-ups for most benchmarks when offloading using WiFi. The geometric mean of computation speed-up that is observed for offloading interactive applications over WiFi is 2.88X. The high latency and poor bandwidth of 3G made it much less successful than WiFi however. Most of our benchmarks did not see performance improvements with the exception of Fractal, Poker, Linpack, and Factor. In the other benchmarks the scheduler found it too costly to offload so only small performance impacts

	Benchmark	Package Name	Description
Interactive Benchmarks	Calculus	com.andymc.derivative	Math tool, computes discrete integrals using Riemann sums
	Chess	com.alonsoruibal.chessdroid.lite	Chess game, does BFS to find best moves
	Edgedetect	com.lmorda.DicomStudio	Image filter, detects and blurs edges
	Fractal	com.wimolife.Fractal	Math tool, zooms and re-renders Mandelbrot fractals
	Metro	com.mechsoft.ru.metro	Trip planner, Finds route between subway stations using Dijkstra's Algorithm
	Photoshop	com.adobe.psmobile	Image editor, performs cropping, filtering, and image effects
	Poker	com.leslie.cjpokeroddscalculator	Game aid, uses Monte Carlo simulation to find odds of winning a poker game
	Sudoku	de.georgwiese.sudokusolver	Game aid, solves sudoku puzzles given known values
Computation Benchmarks	Linpack	com.greenecomputing.linpack	Computation benchmark, standard linear algebra benchmark suite testing floating point computation
	Factor	Not available on Google Play	Computation benchmark, hand written application that uses multiple threads to factors large numbers

Table 1: Description of benchmarks used in this evaluation



Interactive Benchmarks	WiFi	3G
Calculus	1.47x	0.89x
Chess	3.06x	0.99x
Edgedetect	1.75x	1.03x
Fractal	15.01x	4.05x
Metro	1.22x	0.72x
Photoshop	1.28x	0.76x
Poker	6.15x	3.87x
Sudoku	4.18x	0.92x
GEOMEAN	2.88x	1.28x
Computation Benchmarks	**WiFi**	**3G**
Linpack	21.38x	20.94x
Factor	201.68x	168.34x
GEOMEAN	65.66x	59.37x

Figure 5: Absolute execution times for benchmarks with no offloading, WiFi offloading, and 3G offloading. Times are broken down to reflect portions of time that the benchmarks are doing something computationally interesting versus navigating the UI. Whiskers show the standard deviation. Computation speed-up figures relative to not offloading are shown on the right.

were seen. On average, offloading using 3G allowed a 1.28X speed-up for our interactive applications.

As a back-of-the-envelope calculation, we can estimate how many WiFi clients running applications from our benchmark an eight core server could handle by computing the average CPU utilization on the server for each application. This comes out to 28% percent utilization which suggests that a COMET server could sustain about 28 active clients.

As a consequence of reducing the absolute runtime of the application and the amount of heavy computation done on the client, energy improvements are observed in all but one case when offloading using WiFi. Figure 6 details the energy costs of each test. The short latency and high bandwidth of WiFi allows a client to spend less energy transmitting and receiving data when transferring state and control over to a server. Thanks to this, COMET was able to improve energy efficiency on average by 1.51X for interactive applications. Again due to 3G's network characteristics and higher energy costs we found that offloading with 3G usually consumed more energy than it saved. Fractal was the only interactive benchmark that saw significant energy improvement when using 3G.

5.3.2 The Amount of Transferred State

We now examine how much data COMET's VM-synchronization primitive transfers when our benchmarks offload execution using WiFi and 3G. Table 3 shows the amount of state transferred downstream and upstream, in KB, from the client. This data is averaged over the same five runs used to gather our performance and energy data.

Because all but one of our benchmarks were not designed by us, the authors of these applications likely did not intend for their code to ship state between a client and server. Furthermore, we made no effort or modifications

		Download (KB)		Upload (KB)	
Benchmark	Init. (KB)	WiFi	3G	WiFi	3G
Calculus	765.7	68.7	8.3	361.0	138.3
Chess	849.7	2777.0	0	860.9	0
Edgedetect	774.3	3818.2	0	641.6	0
Fractal	817.9	619.3	518.2	222.8	249.5
Metro	805.7	13.8	0	635.0	0.1
Photoshop	792.8	41757.0	587.0	25939.1	2309.8
Poker	773.5	2.6	1.1	185.6	138.0
Sudoku	876.4	292.7	0	300.0	0.3
Linpack	807.7	1873.6	1872.9	19.0	0.4
Factor	732.6	13.0	11.2	20.6	7.3
GEOMEAN	798.6	294.3	20.0	331.4	13.8

Table 3: Total number of bytes transmitted when offloading. Init refers to the initial heap synchronization.

to the binaries of these applications to optimize how state is packaged for offloading. Thus the size of state transfers may seem large when compared to figures presented in the literature [9]. Mixed DSM strategies will be used to reduce this cost in future work.

The first thing that occurs when COMET begins to offload an application's execution to a server is an initial heap synchronization including all of the globally reachable state. This first sync includes a great amount of data that will never change later in the execution and is thus considerably larger than future updates typically. This initial heap synchronization is typically between 750–810KB, regardless of the means of connectivity. This takes, on average, 1.69s for a WiFi connection and 6.39s over 3G to complete. As a point of comparison, the average push and pull operations between two endpoints are only 5.77ms over WiFi and 111ms over 3G.

COMET is capable of adapting to the available network conditions. When a low latency connection is available COMET will more eagerly use this connection to exploit more offloading opportunities. Figures 5 and 6 show that for the Photoshop test, even though over 60MB of state needs to be transferred, there can still be performance gains and energy savings. Conversely when network latency is high and bandwidth is limited, as is the case when operating using 3G connectivity, COMET determines that offloading computation is not advised and scales back its decisions to offload. Table 3 reflects this observation, as the KB of data communicated downstream and upstream when using 3G is significantly less than that of WiFi.

While the state transmissions observed in Table 3 may seem large, future technologies may make it less costly, in terms of time, energy, or both, to transmit data and execute remotely than to perform computation locally. Cellular carriers are currently adopting 4G LTE, which provides higher bandwidths and lower latencies than 3G [16]. This trend in cellular technology promises that in the coming years it will be practical to transmit the amount of data necessary to keep state synchronized when exploiting more offloading opportunities.

```
Loop: for (;;) {
  int op = iCode[frame.pc++];
  ...
  switch(op) {
    ...
    case Token.ADD :
      --stackTop;
      do_add(stack, sDbl, stackTop, cx);
      continue Loop;
    case Token.CALL : {
      ...
    }
  }
}
```

Figure 7: Excerpt from Rhino's Interpreter.java. Method-granularity offloading is going to have difficulties offloading code of this style.

5.4 Case Studies

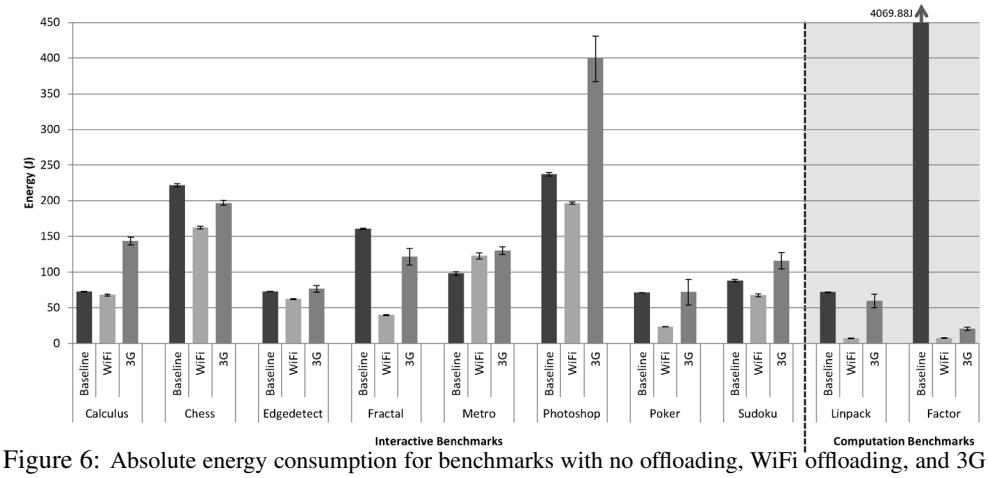
We now examine specific cases where COMET's design allows it to offload where other systems may not. Case study I focuses on the benefits of offloading at a fine granularity while case study II looks at the benefits of being able to offload multiple threads.

Case Study I: Offloading JavaScript

A natural question of this system is if it can work with derived runtimes. In particular, because of JavaScript's prevalence on the web, it would be interesting if JavaScript could be offloaded as well. Unfortunately, there are no existing web browsers that use a JavaScript engine written in Java. The nearest thing appears to be HtmlUnit [1], a "GUI-Less browser for Java programs" aimed at allowing automated web testing within Java. The JavaScript runtime backing HtmlUnit is called Rhino [2].

To test COMET's ability to offload JavaScript with Rhino, we ran the SunSpider [3] test suite with and without offloading. For the 18 benchmarks that could run correctly (some tests exceeded the Dalvik VM's stack size), we found that the entire test executed 6.6X faster with a maximum speed-up for a single test of 8.5X and a minimum speed-up of 4.7X. This suggests that COMET can effectively be applied to derived runtimes.

It is important to mention that method-granularity offloading would fall short in this scenario. Figure 7 gives an excerpt from Rhino's interpreter. The interpreter method cannot be offloaded as a whole as any subsequent client side only calls, such as accesses to the UI, would need to make an RPC right back to the client which could be quite expensive if there are even a few such calls. However, if we do not offload the interpreter loop, there is likely no other method that has a substantial running time.



Interactive Benchmarks	WiFi	3G
Calculus	1.07x	0.51x
Chess	1.36x	1.13x
Edgedetect	1.17x	0.95x
Fractal	4.02x	1.32x
Metro	0.80x	0.76x
Photoshop	1.21x	0.59x
Poker	3.05x	0.99x
Sudoku	1.30x	0.76x
GEOMEAN	1.51x	0.84x
Computation Benchmarks	WiFi	3G
Linpack	10.39x	1.20x
Factor	517.50x	185.24x
GEOMEAN	73.32x	14.93x

Figure 6: Absolute energy consumption for benchmarks with no offloading, WiFi offloading, and 3G offloading. Whiskers show standard deviation. Energy improvements relative to not offloading are displayed on the right.

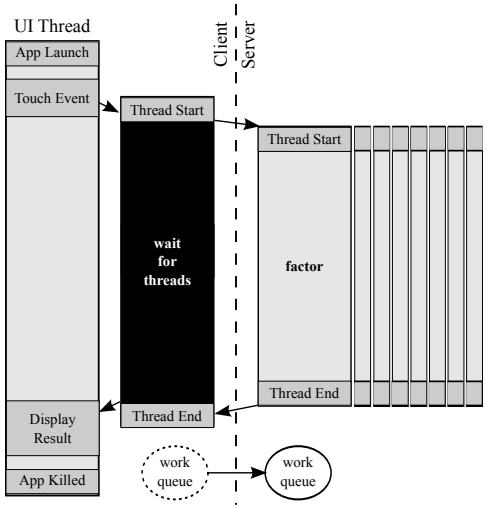


Figure 8: An illustration of multi-threaded execution being offloaded to a server for the Factor application. Time progresses from the top to the bottom of the diagram.

do_add for example is typically going to be fairly quick to execute. Therefore, offloading at a finer granularity than methods is necessary to offload with Rhino and programs like it. This feature is something that COMET offers over past offloading systems.

Case Study II: Multi-threading in Factor

The multi-threaded Factor benchmark shows the best case performance of COMET. It works by populating a work queue with some integers to be factored, and starting eight threads to process items from the queue. This also demonstrates that COMET can work well even when some amount of synchronization is required between threads. Figure 8 illustrates the offloading of the multi-threaded Factor benchmark.

The speed-up obtained by WiFi and 3G are 202X and 168X respectively. This is the difference between 44 minutes running locally and 13 seconds over WiFi demonstrating massive speed-ups using multiple cores. Other works cannot offload more than one thread at a time, especially when there are shared data accesses occurring [9, 8], and therefore can only get a speed-up of around 28.9X.

6 Limitations

Broadly, there are two important limitations of our work. First, COMET may decide to send over data that is not needed for computation. This is often wasteful of bandwidth and can make offloading opportunities more sparse. Two approaches that may help mitigate this challenge are using multiple DSM strategies like what is done in Munin [7] or applying static analysis to detect when data need not be sent.

The second limitation lies in the kinds of computation demanded by smartphones. We have not found a large number of existing applications that rely heavily on computation. Those that do frequently either implement offloading logic right into the application or write the computationally intensive parts of the application in C making it difficult to test with COMET. However tools like COMET can allow new kinds of applications to exist.

7 Conclusion and Future Work

In this paper, we have introduced COMET, a distributed runtime environment aimed at offloading from smartphones. We introduced a new DSM technique and VM-synchronization operation to keep endpoints in a consistent state according to the memory model of our runtime. This makes all virtualized code offloadable and allows multiple threads to be offloaded simultaneously. We demonstrated this system on nine real applications

and showed an average speed-up of 2.88X. In one hand-written application, we were able to reach as much as 202X speedup. To broaden the impact of our work, we plan on making the COMET system available upon publication.

Moving forward, the most promising line of work is in improving the scheduling algorithm used by COMET. The τ-Scheduler described here is the simplest reasonable scheduler that we could come up with and the focus of this work lies elsewhere.

8 Acknowledgements

We thank Tim Harris for his time and energy of serving as the shepherd for this paper and providing numerous constructive comments and detailed feedback for improving the quality of this work. We also thank the anonymous reviewers for their feedback and helpful suggestions. This research was supported by the National Science Foundation under grants CNS-1050157, CNS-1039657, CNS-1059372 and CNS-0964478.

References

[1] A Java GUI-Less Browser. `http://htmlunit.sourceforge.net/`.

[2] Rhino : JavaScript for Java. `http://www.mozilla.org/rhino/`.

[3] SunSpider JavaScript Benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proc. Int. Conf. Parallel Processing*, 1999.

[5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proc. ACM Symp. Operating Systems Principles*, 1993.

[6] L. Cardelli. Obliq - A language with distributed scope. In *Proc. of the Symposium on Principles of Programming Langauges*, 1995.

[7] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computation*, 1995.

[8] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patt. CloneCloud : Elastic Execution between Mobile Device and Cloud. In *Proceedings of the European Conference on Computer Systems*, 2011.

[9] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Ch, and P. Bahl. MAUI: Making smartphones last longer with code offload. In *Proc. MOBISYS*, 2010.

[10] CyanogenMod Community. CyanogenMod Android community rom based on gingerbread. `http://www.cyanogenmod.com/`.

[11] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *Proc. Int. Symp. Operating Systems Design and Implementation*. USENIX Association, 2004.

[12] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI*, 2010.

[13] A. Gember, C. Dragga, and A. Akella. ECOS: Practical Mobile Application Offloading for Enterprises. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2012.

[14] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the TCL/TK workshop*, 1996.

[15] X. Gu, K. Nahrstedt, A. Messer, D. Milojicic, I. Greenberg, I. Greenberg, and HP Laboratories. Adaptive offloading inference for delivering applications in pervasive computing environments. In *Proc. of IEEE International Conference on Pervasive Computing and Communications*, 2003.

[16] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2012.

[17] A. D. Joseph, A. E. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. ACM Symp. Operating Systems Principles*, 1995.

[18] A. Judge, P. Nixon, V. Cahill, B. Tangney, and S. Weber. Overview of distributed shared memory. Technical report, Trinity College Dublin, 1998.

[19] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 1988.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the international symposium on Computer architecture*, 1992.

[21] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: A computation offloading framework for smartphones. In *MOBICASE*, 2010.

[22] R. McIlroy and J. Sventek. Hera-JVM: A runtime system for heterogeneous multi-core architectures. In *Proceedings of the ACM international conference on object oriented programming systems languages and applications*, OOPSLA, 2010.

[23] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *ICDCS*, 2002.

[24] Monsoon Solutions Inc. Monsoon power monitor. `http://www.msoon.com/LabEquipment/PowerMonitor/`.

[25] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proc. ACM Symp. Operating Systems Principles*, 2009.

[26] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of Mobisys*, 2011.

[27] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. SociableSense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of MobiCom*, 2011.

[28] R. Reda et al. Robotium. `code.google.com/p/robotium`.

[29] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. *SIGOPS*, 1997.

[30] W. Zhu, C.-L. Wang, and F. C. M. Lau. JESSICA2: A distributed Java virtual machine with transparent thread migration support. In *Proceedings of the IEEE international conference on cluster computing*, 2002.