Using a Multi-Tasking VM for Mobile Applications

Yin Yan, Chunyu Chen, Karthik Dantu, Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York
{yinyan, chunyuch, kdantu, stevko, lziarek}@buffalo.edu

ABSTRACT

This paper discusses the potential benifits of switching Android's *single VM per application* runtime environment to a multi-tasking VM environment. A multi-tasking VM is a type of a Java virtual machine with the ability to execute multiple Java applications in one memory space. It does so by isolating the applications to prevent interferences. We argue that using a multi-tasking VM for mobile systems provides better control over application lifecycle management, more flexible memory management, and faster inter-application communication. To support this argument, we discuss a preliminary design, implementation, and evaluation for an alternative to Android's communication mechanism, `Binder`, and demonstrate the benefits afforded by a multi-tasking VM.

Keywords

Mobile systems; Multi-tasking virtual machine; Runtime

1. INTRODUCTION

With the adoption in Android, Java-based runtime environments have become the most popular execution model for mobile applications. They have many characteristics ideally suited for mobile environments; for example, they use bytecode and executes it in a VM, making it portable across different ISAs; they automatically manage memory, allowing the underlying system to employ efficient memory management policies; they are also type-checked, lowering the cost of development. Due to these benefits, Android now supports countless devices, ISAs, users, and developers.

The default execution mode for Java is a *single VM per application*. This mode of execution is the default in almost all Java deployments, whether it is on a server, a desktop, or a mobile phone. However, this is not the only mode of execution available. Historically, *Multi-Tasking VM* (Multi-VM, or simply MVM) [4] has been providing the ability to run *all* applications in a single VM. MVM runs each application in an isolation unit called a *partition* [17], but all applications run as a single process.

In this paper, we argue that the MVM mode of execution is the better choice for Android. With the increasing CPU power and massive on-board memory, it is common to run multiple cooperative applications on a single device. The latest Android release, Marshmallow, allows running more than one foreground activity from different applications at the same time. Executing all applications in a single MVM instance gives a global view of application behaviors with regards to system resources such as memory and I/O. This global view allows the MVM to better accommodate more application requests and efficiently manage available resources. Additionally, running all applications in a single process simplifies communication between applications; the VM can avoid context switches and copying data from one memory space to another.

To concretely demonstrate these benefits, we first examine an MVM architecture, and discuss the feasibility of replacing Android's VM (Dalvik/ART) with an MVM. We examine all the salient features of Android's VM and compare how an MVM can realize the same features. Our conclusion is essentially that *there is no feature that cannot be replicated* in an MVM architecture. To understand the advantages, we pick one essential sub-system (inter-application communication), and show how an MVM can implement the same feature more efficiently. Our overall findings are that (i) an MVM architecture has tangible benefits in comparison to existing Android, (ii) the changes necessary to use an MVM are only required in the framework level, and (iii) all the benefits can be made transparent to applications. Although single process execution for multiple programs itself is not a new idea (*e.g.*, Singularity [7, 8, 9]), our focus is not on designing a new OS, but on replacing a component in an existing OS to demonstrate additional benefits.

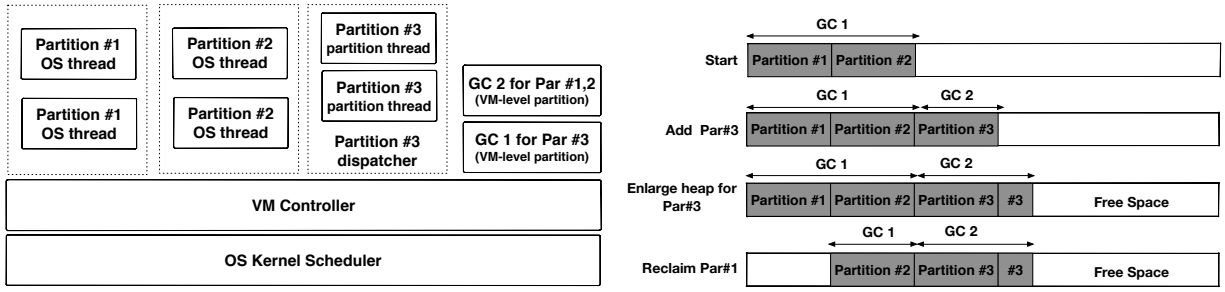
2. MULTI-TASKING VM

The use of an MVM architecture was first proposed to achieve better scalability as well as security for server style Java applications [4]. Later, MVM was also expanded to support multiple users instead of only a single user [6]. The design philosophy of MVM is similar to Singularity [7, 8, 9], a dependable and robust micro-kernel system built with a type-safe language. Singularity separates its computation units as Software Isolated Processes (SIPs)—each program, device driver, or system extension is executed in its own SIP under the same address space. The inter-process communications are enabled by typed bi-direction channels. To

HotMobile '16, February 26-27, 2016, St. Augustine, FL, USA
 ISBN 978-1-4503-4145-5/16/02…$15.00
DOI: http://dx.doi.org/10.1145/2873587.2873596

	Android Dalvik/ART	Multi-tasking VM
Application isolation	The access of sensitive framework APIs is controlled with manifest configuration, and file permissions and other application resources are protected per process.	An OS-like access of control unit can be implemented in MVM.
Application initialization	Forks zygote, creates a new VM instance in a new process, and loads application classes.	Creates a new partition for an application, configures a time slice for the application that runs in the newly created partition, and loads application classes.
Application context switch	Switching between processes requires to switch memory addresses, page tables and kernel resources, and flush processor caches.	Threads switching between different partitions only requires switching processor states, *e.g.*, scheduling counter or register contents, which can be done fast and efficiently.
Application termination	The exit of an application components do not terminate the process of the application. The process is killed only when its user manually kills it or the system runs low on memory. The Low Memory Killer (LMK) is responsible for selecting and killing processes.	Applications are executed in their dedicated partitions. MVM provides VM-level APIs to enforce all of the threads in one partition to exit, and reclaim their memory and other resources.
Application suspension and resume	The OS scheduler switches between processes.	The VM controller switches partitions in the same process.
Intra-app communication	Message passing constructs	Message passing constructs
Inter-app communication	Android Binder IPC	Shared memory in MVM
System service	Executed in one dedicated process, accessed via `Binder` calls	Executed in one dedicated partition, accessed via inter-application communication mechanism.
Memory Management	Virtual memory with per-app garbage collection.	A single memory address space with enforced memory boundaries and configurable garbage collection strategies for different partitions.

Table 1: Application Management Between Android's Dalvik/ART and MVM



(a) Time Partition (b) Memory Partition

Figure 1: MVM Partition Architecture

guarantee the safety and correctness, all application code is associated with manifest configurations for static validation and analysis.

Similarly, MVM has the ability to execute multiple runtime environments without interferences under one single VM instance. The spatial and temporal isolations can be achieved via class loading [3, 5] or resource partitioning [11]. In our discussion, we will focus on a resource-partitioned MVM, Fiji VM [17, 12], which isolates the interferences by partitioning applications into execution units, and enforces application lifecycle management and memory boundaries through the use of a VM controller. For the inter-application communication, MVM can develop communication primitives at the VM level over region-based memory, and higher-level protocols are more easily encoded without necessitating copying. More details will be discussed in Section 2.1 and Section 2.2. We use the term "partition" to mean the execution unit in MVM in the later sections. Compare to Singularity, MVM does not provide built-in verification tools. However, it defines sophisticated interfaces for different types of resource management. For example, the CPU cycles and memory boundaries can be either statically configured or dynamically adjusted between partitions via the VM controller, the controller also facilitates finer-grained control for handling interrupts, task preemption, and memory reclamation.

The rest of section explores the application features that are affected by replacing Android's VM with an MVM and some of the added benefits of doing so. To explain the functional differences, we provide a side-by-side comparison of the basic features that are essential to Android's Java Runtime Environment and their analogues in MVM in Table 1. We categorize these features into three major aspects: application lifecycle management, memory management and inter-application communication.

2.1 Application Lifecycle Management

MVM manages the lifecycle of its applications by partitioning each application's tasks into separate time slots. Thus, application execution is isolated within its own temporal partition. MVM provides VM-level APIs to allow a VM controller to manipulate the initiation, suspension, and

termination of the tasks in each partition with fine-grained control. Each partition can create OS level threads, which underpins Android created threads. All threads, regardless of which partition they belong to, are in the same OS process.

MVM can provide various scheduling options to manage threads that belong to different partitions, exposing the trade-off between fine-grained control and management overhead. The simplest way to manage partition threads is to directly leverage thread scheduling from the operating system; the threads of each partition are grouped and scheduled with fixed-length time slices via operating system thread scheduling. The threads in the current active partition are scheduled within partition time slices. When the current partition exceeds its time slice, the VM controller suspends all threads in this partition and resumes threads in the next partition. Figure 1a shows various scheduling options potentially available. Partition 1 and 2 are utilizing direct OS thread scheduling for their execution.

Alternatively, MVM can also provides VM-level scheduling for better control with a task dispatcher that manages a thread pool. The threads in a partition are scheduled through the dispatcher as a VM partition thread, as shown with partition 3 in Figure 1a. The VM controller can manipulate the partition lifetime via VM-level APIs to apply different scheduling policies of the task dispatcher. The execution of the task dispatcher introduces additional complexity; additional safe points [1] need to be checked during execution to allow the VM controller to safely suspend, resume, and terminate threads from the dispatcher. In a nutshell, direct OS thread scheduling is lightweight but with less control. The VM-level scheduling approach requires the execution of the dispatcher with additional safe point checking, but it provides better control. Normally, MVM supports a hybrid approach, allowing the developers to choose how their partitions are scheduled based on performance requirements.

The biggest benefit of performing application scheduling at the VM level is that context switches between applications are extremely fast. This occurs because the TLB and caches are not flushed—the cost of a context switch is simply the cost of switching between threads. In a loosely coupled, event driven system like Android, where many applications communicate, this has an additional benefit—namely a hot cache. Switching at a communication point has a good probability of retaining communication structures in the cache, yielding additional performance benefits. Singularity also enjoys fast context switching.

2.2 Application Memory Management

Although MVM provides a dynamically configurable memory management at runtime, it enforces memory boundaries for space isolation. Namely, one application cannot allocate objects or hold references to other application's heap memory. To reclaim dead objects, each application can have its own GC (typical in a partitioned MVM), but there maybe one GC that is shared by many applications (typically in an MVM that uses classloader based isolation). MVM may use a combination of GC approaches and even leverage other automatic memory management schemes, like scoped memory [2] (typical in more specialized MVMs). The key observation is that in all MVM schemes, the applications are executed in the same address space and it is up to MVM to ensure that heap boundaries are enforced. Figure 1b demonstrates basic heap management scenarios in MVM memory management for a partitioned MVM. In this case we assume a partitioned MVM, which assigns GCs to monitor potentially more than one partition [2].

Like Android's Dalvik/ART, each partition starts with a memory quota (the application's initial heap size—this can be uniform across all applications or can be tailored based on the application's need), and the heap size can be increased at runtime, if necessary. The initiation or termination of a partition induces the allocation or reclamation of the heap memory for that partition, in much the same way as application termination on Android causes a reclamation of the host VM and its assigned heap memory.

The main benefit of an MVM GC scheme is that MVM can arbitrate memory allocation requests from different applications executing in different partitions. As a direct consequence, MVM can not only decide when, but also where to run the garbage collection. This is especially useful for managing memory on a memory constrained device. Lets consider what happens on Android when memory usage is low. In such situations, Android triggers its Low Memory Killer (LMK) to selectively kill applications. This is done via an importance metric. Crucially, the Low Memory Killer does not understand how much "garbage" a given application has resident in its heap memory. To skirt this issue, Android allows the OS to trigger GC events in a given VM. However, once the LMK is triggered, memory is reclaimed on an application level.

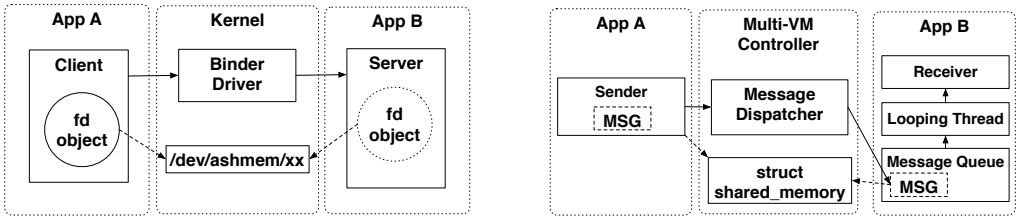
In MVM, the VM controller understands the memory consumption of each application. It can adjust heap boundaries between partitions to respond to shortage of memory in individual partitions. Additionally, MVM can even trigger GC on a partition whose app is currently not active and reallocate the collected memory to another partition that requires more memory. Thus, the memory requests are satisfied without sacrificing the correctness of the applications, and without resorting to drastic measures such as terminating other applications. What this means is that the system as a whole can make more *holistic* memory management decisions. We believe this will also allow for more programmatic definitions of *global* memory management schemes instead of the heuristic-based LMK mechanism. It is this global memory management that differentiates MVM from OS based solutions like Singularity.

2.3 Inter-App Communication

Android applications rely on Android's `Binder` IPC mechanism for communication and data transfer. For example, Android's messaging objects, `Intent` or `Message`, transfer their data object `Bundle` via Android's `Binder` calls. Since `Binder` calls require a data copy between the communicating applications, it is limited in the size of data it can transfer. To exchange a large chunk of data, developers have to use an external medium such as `ContentProvider` or `ashmem`.

[1] A safe point is a compiler injected check, to ensure GC runs periodically by suspending threads. This can be leveraged to bound the amount of time needed to suspend a thread. We note that for a given architecture, this bound can be calculated precisely, affording an added level of determinism to the system.

[2] This is useful when you may want to support different types of GC. For instances a real-time GC for applications with timing guarantees and a more throughput-oriented GC for multimedia applications.



(a) Android's `ashmem` (b) Multi-VM Shared Memory

Figure 2: Communication with Shared Memory

Nevertheless, `Binder` is a building block leveraged by both `ContentProvider` and `ashmem`, not for large data transfer, but for exchanging small meta data required for data transfer.

Thus, our preliminary exploration in this paper takes a deeper look at `Binder` and replaces it with an alternative mechanism in MVM. We discuss this in more detail in the next section. Below, we briefly lay out our thoughts on how we could replace `ContentProvider` and `ashmem` in MVM.

`ContentProvider` is one of the core components of an Android application; it encapsulates data and provides standard interfaces that allows data in one process to be transferred to code running in another process. It is designed for data persistence in mind, and the access of data via `ContentProvider` interfaces requires expensive I/O operations. A `ContentProvider` implementation in MVM would look much the same as it does in Android itself. Since the primary requirement is persistence, the MVM does not provide any added benefit since the underlying file system governs the majority of the access costs. We observe, however, that typical kernel permission mechanisms need to be move to the MVM to retain the semantics that Android expects. For instance, associations between file descriptors and the process IDs would need to be maintained.

In contrast, `ashmem` is a Android Linux module that facilitates direct memory sharing between processes. As shown in Figure 2a, `ashmem` allows applications to allocate a shared memory region, map it to a physical memory address, and handle it as an file descriptor. Since the file descriptor is created via `mmap()`, it is only valid in its own process. To share the file descriptor, developers have to wrap it with an `MemoryFile` instance, and pass the `MemoryFile` object through a `Binder` call. Applications that hold the reference to the same `MemoryFile` instance have to cooperate with others explicitly via Android's communication mechanism to use the memory region abstracted by the file descriptor.

Our observation is that `ashmem`-like functionality can be implemented in an MVM very naturally, using our `Binder` replacement discussed in the next section. Our `Binder` replacement is essentially a shared memory substrate and provides a common interface that applications can leverage to communicate. We illustrate how we can build such a mechanisms in the next section and show that it can outperform kernel-based approaches for data transfer in Section 4.

3. MVM INTER-APP COMMUNICATION

To illustrate the power of MVM, we show in this section how easily we can provide an inter-application communication mechanism without crossing the process boundary between applications. Figure 2b shows one possible design that we implement to replace Android's `Binder`. The design is based on asynchronous message passing with shared memory. As Figure 2b shows, each application has a *looping thread* associated with a *message queue* in its own partition. Other threads that have a reference to the *message queue* can send messages to the *looping thread* by synchronizing with the *message queue*. Unfortunately, MVM isolates applications in both time and space at runtime, the sender thread in one partition cannot directly refer to the *message queue* in another partition. The only way to do the inter-partition references is using native objects in a controlled manner.

Thus, our design utilizes inter-partition locks and shared memory regions as part of the functionalities in the internal MVM controller. MVM only exposes Java APIs to the application layer, encapsulating the native code. Our communication mechanism consists of three main constructs, implemented at the VM controller level: `struct shared_memory`, `Message`, and `MessageDispatcher`.

`struct shared_memory` is a shared memory object, implemented in native code. It consists of a pointer to shared data, a reference counter, *a pthread lock*, and *a conditional variable*. The lock and condition variable are used for synchronization between read and write operations from different partitions to ensure consistence of the shared data.

`Message` is a messaging object class. The instantiation of a `Message` object associates the object with the allocation of the native `struct shared_memory`. Since we do not want to expose the native object directly to developers, we decided that the `Message` instances must be instantiated and recycled via `MessageDispatcher`. Code 2 presents the primary functions of the `Message`, it has a `Pointer` field that points to the `shared_memory` variable.

`MessageDispatcher` is a Java singleton class that contains both Java and native interfaces, as shown in Code 1. It controls the initiation and reclamation of the `Message` objects, and has the references to the *message queues* in all of the partitions. `MessageDispatcher` is responsible for dispatching `Message` objects to the receiving partitions. Notice that MVM cross-partition references are forbidden, the `Message` objects from sender partition can not be directly referenced in the receiving partitions. Thus, when a sender calls `MessageDispatcher.sendMessage(msg)` function, the `MessageDispatcher` first updates the reference counter of `shared_memory`. Then, it creates a new `Message` object in each receiving partitions, and assigns the `Pointer` fields of these newly created `Message` objects. By doing this, each receiving partition has a `Message` object in its own heap memory that points to the same `shared_memory`. Then, `MessageDispatcher` can recycle the `Message` object from sender. After the receivers process the `Message` objects in the `looping thread`, they decrease the reference counter on `shared_memory`. When the reference counter is updated to zero, `shared_memory` is freed.

Since our inter-application communication is enabled via shared memory and native locks, additional care needs to be taken to insure that the native objects are recycled, and to limit their number allocated at runtime similar to kernel-

```
class MessageDispatcher {
  public static MessageDispatcher getInstance();
  ...
  public Message getMessage() { ... }
  public void recycleMessage(Message msg) { ... }
  public void sendMessage(Message msg) { ... }
  private static native void allocate_msg(Pinter msg);
  private static native void free_msg(Pointer msg);
}
```

Code 1: MessageDispatcher

```
class Message {
  Pointer shared_memory
  ...
  /*package*/ Message(){ ... }
  public Byte[] read(int s, int l) { ... }
  public void write(int s, int l, Byte[] d) { ... }
  /*package*/ static void allocate(Pointer pointer);
  /*package*/ static void free();
}
```

Code 2: Message

```
struct shared_memory {
  char* id;
  void* data;
  int ref_counter;
  pthread_mutex_t lock;
  pthread_cond_t cond;
};
typedef struct shared_memory shared_memory;
//A list holds all shared_memory variables
shared_memory* shared_memory_list;
```

Code 3: Native struct: shared_memory

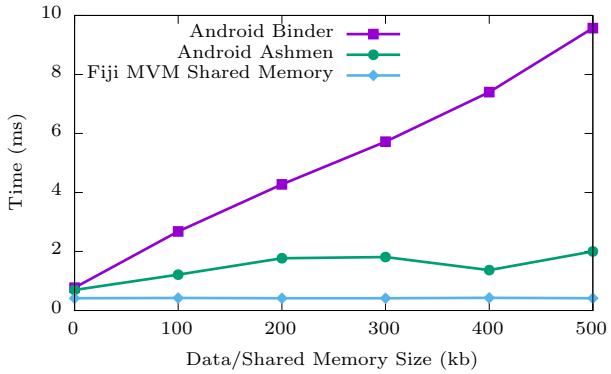


Figure 3: Communication Cost with Increasing Data Sizes

based process isolation. To achieve this, we use an object pool. This requires enforced reclamation of shared memory and native locks at the VM level, but we note that our choice of MVM supports such reclamation.

Singularity [7] provides bi-directional, two-party, typed channels that offer fast communication. More complex protocols can be implemented by using multiple channels. For instance, encoding ring-buffered communication requires the use of an intermediate channel and a thread to encode the buffer. Unfortunately, when multiple channels are leveraged, copying overhead occurs between channels. We note, however, that Singularity could be extended to support different kinds of channels (*e.g.*, ring-buffered channels) to mitigate this, but at the cost of adding complexity to the static verification process.

4. PRELIMINARY RESULTS

To demonstrate the benefit of our proposed architecture, we have conducted preliminary experiments that measure raw communication costs. Our goal is to compare the overhead of data transfer across three mechanisms—Android's `Binder`, Android's `ashmem`, and the shared memory mechanism of an MVM. We have performed all experiments using a Google Nexus S with Android v4.1.2 (Jelly Bean) with the performance governor. For MVM experiments, we have used Fiji MVM integrated with Android v4.1.2. Our workload consists of two applications, one sender and one receiver, and we vary the amount of data transferred between these two applications. We record a timestamp before the sender sends data, and another timestamp after the receiver receives the data. We compute the communication cost as the difference of the two timestamps. We collect 500 these pairs of timestamps for each data size.

Figure 3 shows the average data transfer costs for the three mechanisms with different data sizes. Since Android's `Binder` transfers actual data through */dev/binder* in kernel by copying the data from the sender to the receiver, it is expected that its performance increases linearly with the data size. The other two methods are more or less constant, since there is no actual data transferred between two applications. They only involve a context switch between two applications. However, we observe that the data transfer overhead of Android's `ashmem` is around 5 times slower than that of Fiji MVM, because Android runs each application in a separate process with a dedicated VM instance; the context switch between two processes is generally much more expensive than switching between two threads, even if those threads reside in different partitions.

5. DISCUSSION

While MVM brings the advantages described in previous section, it has the following disadvantages compared to using a single VM per app.

Access Control Since MVM runs all applications in a single process, it needs to have its own access control mechanism for shared resources, such as hardware devices and data files. This adds runtime overhead and complexity in terms of implementation as well as access control policy definition.

Isolation MVM provides spatial and temporal isolation to avoid interferences between applications in the Java runtime. Even if an application is misbehaving or exhibits a Java-side bug, the MVM controller can tear down the malfunctioning application and its associated partition without affecting the others. However, native code requires separate mechanisms for isolation and misbehavior as discussed below.

Native Code Native code is a major concerns for an MVM. Native code provides the capabilities for direct memory access to anywhere in the memory address space, giving rise to a potential mechanism to violate the MVM isolation. Similarly, objects allocated in native code are not subject to the reclamation. Both these aspects pose challenges to the adoption of the MVM for Android.

In fact, Singularity [7, 8, 9] has the same challenges and

overcomes them with code verification. Each piece of native code in Singularity must provide a specification and be verified against this specification. In Singularity, there is a limited set of hardware-related implementations that are written in C++ and assembly as privileged instruction. Because type and memory safety assure the execution integrity of functions, Singularity can place privileged instructions, with safety checks, in trusted functions. For example, privileged instructions for accessing I/O hardware can be safely in-lined into device drivers.

For MVM, such an approach is also a viable solution. We believe this can be done by abstracting away common native functionality into libraries. As a concrete example, we are currently investigating certain communication protocols on top of more specialized MVM communication primitives. Priority rollback protocol [18] and wait free pair transaction [1] can be used for fast inter-partition communication with bounded latency and known memory bounds. We believe we can leverage both as the foundation for building Android communication primitives.

6. RELATED WORK

The canonical approach to multitasking in the Java programming language is to start each application in a new JVM [10]. This typically requires spawning a new operating system process for each application and protect the application from each other, but uses large amounts of resources (memory, CPU time) and makes inter-application communication expensive.

Android's VM is this type of a virtual machine. Dalvik VM was introduced when the first version of Android was released on 2008. It takes its own bytecode format and executes them on mobile devices. To provide isolation between applications, Android applications run in their dedicated VM instances within separate OS processes. When an application is started, the *zygote* process forks itself, creates a new VM instance, and duplicates the preloaded classes and resources in a new process for runtime. To achieve hardware-specific optimization, Google replaced Dalvik VM with Android Runtime (ART) in Android v5.0 (Lollipop). One of the major changes is that ART includes an ahead-of-time compiler (AOT) which compiles the Dex bytecode to native ELF executable. The compilation is done when the app is installed. An alternative to this model is to execute applications in one Multi-VM (MVM), which we explore in this paper.

MVMs have been adapted for real-time applications, where it is hard to preserve predictability in the presence of dynamic memory management [13]. Our prototype is built on the top of the Fiji MVM [17, 12], which provides predictable temporal isolation with real-time capabilities. Fiji MVM allows us to statically configure the memory bounds for its payloads at compile time. Similarly, KESO [14] is another MVM designed for statically configured resource-constrained embedded systems. It uses ahead-of-time knowledge to generate a Java runtime that is specifically tailored towards a given application and configures spatial isolation and memory protection statically.

7. CONCLUSION AND FUTURE WORK

In this paper, we have explored the feasibility of using MVM for Android. Unlike the existing mode of execution where an application runs within its own VM, MVM runs all applications in a single process. This architecture enables fine-grained control over application lifecycle and memory management and also reduces the context switching cost significantly as there is no real context switch happens across processes. To demonstrate these benefits, we have presented a design of an inter-application communication mechanism using an MVM. This can replace Android's `Binder` IPC mechanism. Our performance comparison shows that our design reduces the cost of communication significantly. As part of our ongoing work, we are integrating the Fiji MVM with RTDroid [16, 15] to provide a mixed-criticality environment for multiple applications.

Acknowledgments: This work has been supported in part by an NSF CAREER award, CNS-1350883.

8. REFERENCES

[1] E. Blanton and L. Ziarek. Non-Blocking Inter-Partition Communication with Wait-Free Pair Transactions. In *JTRES*, 2013.

[2] G. Bollella and K. Reinholtz. Scoped Memory. In *ISORC*, 2002.

[3] G. Czajkowski. Application Isolation in the Java Virtual Machine. In *OOPSLA*, 2000.

[4] G. Czajkowski and L. Daynés. Multitasking without Comprimise: A Virtual Machine Evolution. In *OOPSLA*, 2001.

[5] G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. In *ECOOP*, 2002.

[6] G. Czajkowski, L. Daynès, and B. Titzer. A Multi-User Virtual Machine. In *USENIX ATC*, 2003.

[7] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys*, 2006.

[8] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *EuroSys*, 2007.

[9] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[10] Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/.

[11] M. Jordan. Resource Partitioning in a Java^{TM}Operating Environment. Technical report, 2006.

[12] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-Level Programming of Embedded Hard Real-Time Devices. In *EuroSys*, 2010.

[13] M. Stilkerich, I. Thomm, C. Wawersich, and W. Schröder-Preikschat. Tailor-made JVMs for Statically Configured Embedded Systems. *Concurr. Comput. : Pract. Exper.*, 24(8):789–812, June 2012.

[14] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. KESO: An Open-source multi-JVM for Deeply Embedded Systems. In *JTRES*, 2010.

[15] Y. Yan, S. H. Konduri, A. Kulkarni, V. Anand, S. Ko, and L. Ziarek. RTDroid: A Design for Real-Time Android. In *JTRES*, 2013.

[16] Y. Yan, S. H. Konduri, A. Kulkarni, V. Anand, S. Ko, and L. Ziarek. Real-Time Android with RTDroid. In *MobiSys*, 2014.

[17] L. Ziarek, , and E. Blanton. The Fiji MultiVM Architecture. In *JTRES*, 2015.

[18] L. Ziarek. PRP: Priority Rollback Protocol – a PIP Extension for Mixed Criticality Systems: Short Paper. In *JTRES*, 2010.