# C-RAM: Breaking Mobile Device Memory Barriers Using the Cloud

Andreas Pamboris and Peter Pietzuch

**Abstract**—Mobile applications are constrained by the available memory of mobile devices. We present C-RAM, a system that uses cloud-based memory to extend the memory of mobile devices. It *splits application state* and its associated computation between a mobile device and a cloud node to allow applications to consume more memory, while minimising the performance impact. C-RAM thus enables developers to realise new applications or port legacy desktop applications with a large memory footprint to mobile platforms without explicitly designing them to account for memory limitations. To handle network failures with partitioned application state, C-RAM uses a new snapshot-based fault tolerance mechanism in which changes to remote memory objects are periodically backed up to the device. After failure, or when network usage exceeds a given limit, the device rolls back execution to continue from the last snapshot. C-RAM supports local execution with an application state that exceeds the available device memory through a user-level virtual memory: objects are loaded on-demand from snapshots in flash memory. Our C-RAM prototype supports Objective-C applications on the unmodified iOS platform. With C-RAM, applications can consume 10× more memory than the device capacity, with a negligible impact on application performance. In some cases, C-RAM even achieves a significant speed-up in execution time (up to 9.7×).

**Index Terms**—Cloud-based memory, application state partitioning, snapshot-based fault tolerance, user-level virtual memory

✦

## 1 INTRODUCTION

While the compute, memory and network resources on mobile devices such as smartphones and tablets have increased substantially over the years, their capabilities fundamentally lag behind those of servers and desktop machines. In particular, due to form factor constraints, the amount of main memory in mobile devices must be balanced against the incurred reduction in battery lifetime [1]. With the advent of even smaller wearable devices such as smart watches and glasses [2], [3], memory is likely to become an even more constrained resource in the forthcoming years.

At the same time, mobile application developers have started realising more memory-demanding applications: users now expect mobile games to include sophisticated AIs that search large game state [4], [5], image and video editing applications that maintain large media objects in memory to apply video transcoding and image effects in real-time [6], [7] and augmented reality platforms such as Google Glass [8] to process large volumes of data with advanced computer vision algorithms.

For many applications, available device memory therefore becomes a limiting factor [9]. On an Apple iPhone 4S phone, for example, an application is left with just 213 MB of usable main memory, out of a total of 512 MB, with the difference reserved for the iOS operating system. Applications that exhaust the available memory are automatically terminated by iOS. Considering that a single 8-megapixel photo has over 30 MB of bitmap data, an image editing application is terminated for having just seven photos resident in memory.

A. Pamboris and P. Pietzuch are with the Department of Computing at Imperial College London.
E-mail: {ap5309, prp}@doc.ic.ac.uk

To support applications with a large memory footprint, developers often adopt a *client/server model*, with the server side implementing the memory-intensive functions. Such a model, however, entails several challenges: developers must understand the memory requirements of different parts of their application to decide on what objects should be maintained remotely; they must ensure that the additional client/server communication does not degrade application performance; and, perhaps most importantly, they face the challenge that the application now requires network connectivity to the remote server in order to function.

Instead, we propose to *split application state* automatically between a mobile device and a remote node based on fine-grained profiling of the application. An application can thus utilise a total amount of memory that exceeds the device memory without requiring additional developer effort. A major challenge, however, is to enable the split application to continue functioning even when access to the remote state was lost due to network failure. In addition, to increase adoption, we only want to modify applications but not the mobile platform (i.e. iOS or Android) itself.

We describe C-RAM, a system for Objective-C applications that splits application state on the unmodified iOS platform, while being resilient to network failures. It achieves this by automatically partitioning the application source code without developer intervention. The contributions of C-RAM are:

**Application state partitioning.** Based on the results of an offline profiling step, C-RAM applies an optimisation-based partitioning algorithm to split the *application state*, as represented by Objective-C objects, between the mobile device and a remote node. The goal of the partitioning is to enable memory-intensive application workloads while minimising application response time. Each object is placed

permanently either on the local or on the remote node, while respecting resource limits. Access to remote objects is supported transparently via proxy objects, which relay invocations using remote procedure calls (RPCs) [10].

**Snapshot-based fault tolerance.** C-RAM uses a new fault tolerance mechanism, which allows the mobile device to recover missing objects after losing network connectivity. Consistent snapshots of changes to the local and remote application state are stored on the device. After failure, application state is rolled back to the last snapshot, and execution resumes from an earlier point in time. To control the network overhead of snapshot transmission, C-RAM has a tunable snapshot interval and also disables offloading when network usage exceeds a configurable threshold.

**User-level virtual memory.** After failure, C-RAM must execute the application locally but with a state size that is larger than the available memory. Since iOS does not support kernel-level virtual memory with on-demand paging, C-RAM implements a virtual memory scheme at user level: objects from state snapshots remain stored in flash memory and are loaded into main memory on-demand. This permits offline execution, albeit with degraded performance.

C-RAM does not require modifications to iOS or Objective-C. We evaluate a C-RAM prototype with three memory-intensive applications: a two-player board game with an AI component, a spreadsheet application and an image retrieval application. We show that, by partitioning application state, C-RAM can support large memory sizes—e.g. $10\times$ the available device memory— without degrading application performance. In some cases, it can even speed up execution by a factor ranging from $2\times$ to $9.7\times$, depending on the application workload and network properties.

Its fault tolerance mechanism incurs only a modest overhead and permits a choice between reducing network usage or the amount of lost state. After network failure, local execution exhibits a 25% performance reduction caused by the on-demand loading of objects from flash memory as part of C-RAM's user-level virtual memory scheme.

In the remainder of the paper, §2 discusses the implications of limited memory on mobile devices; §3 presents the C-RAM design and partitioning mechanism; §4 describes the snapshot-based fault tolerance mechanism and the user-level virtual memory scheme; §5 presents evaluation results from using C-RAM with real-world applications; and §6 discusses related work. We conclude the paper in §7.

# 2 BACKGROUND

This section explores the implications of limited memory on mobile devices and discusses existing approaches for overcoming such limitations.

## 2.1 Memory Limitations on Mobile Devices

Owing to their smaller form factor, today's mobile devices lack the memory capacity of most desktop and server systems [11]. While technological advances have allowed for a device's circuit board space to be used more efficiently—e.g. using packaging methods that combine vertically processor cores and memory—still almost 100% of the top silicon on CPU dice is used for memory [12]. To further complicate matters, the bulky packaging that is required by larger memory also affects the power and cooling requirements of devices because the memory subsystem is a major energy consumer [1], [13].

The memory on smartphones typically varies from 512 MB (e.g. Apple iPhone 4) to 1 GB (e.g. Apple iPhone 6). A substantial portion of this is reserved for the operating system and multitasking. For example, on an iOS device with 512 MB (or 1 GB) of memory, an application is left with approximately 213 MB (or 550 MB, respectively), assuming no background activity [9]. On an Apple iPad 3 with 1 GB of memory, this translates to allowing roughly 45 frames of uncompressed video to be resident in memory, corresponding to 0.75 seconds of video at full frame rate—a single frame requires 12 MB of bitmap data. Buffering a second of full-screen video would therefore exhaust the available memory.

Low system memory is one of the most common causes of application crashes [14], making it perhaps the most critical resource for mobile devices. Unlike desktop operating systems such as Mac OS, which allow for virtual memory to "spill over" to stable storage, mobile platforms such as Android and iOS provide no swap space to fall back on, mainly due to the implications on application responsiveness and battery lifetime [15]. Instead, they terminate applications under low-memory conditions.

Another consequence of the limited availability of memory is the fact that platforms such as iOS do not support proper multitasking [16]. While an *active* application can be moved to a *background* state, iOS suspends it automatically to conserve memory. The iOS app store review process imposes further constraints: short-lived background execution is confined to specific application functions, such as audio, location tracking, downloads or VoIP calls.

## 2.2 Existing Approaches

**Memory-efficient application development.** Mobile application developers are given guidance how to implement memory-efficient applications. They must comply with manual memory management policies and follow specific practices for allocating, deallocating and accessing application state: e.g. reusing memory buffers, allocating memory at the finest granularity, managing multiple memory pools efficiently and reacting promptly to low-memory warnings at runtime [17], [18], [19].

This places the burden of efficient memory management on developers. It leads to additional development effort, and assumes that developers have a good understanding of various low-level memory management techniques available on different mobile platforms.

**Restricting mobile applications.** For applications that are intrinsically memory-intensive, mobile versions remain noticeably inferior in terms of functionality compared to their
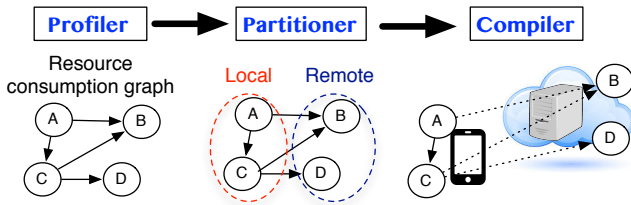
Fig. 1: Overview of the C-RAM system

desktop counterparts. For example, desktop games with advanced AIs, which require large in-memory data structures for the AI computation (e.g. as used by the Latrunculi board game [20] under Mac OS) have restricted mobile versions in order to comply with the memory limits of mobile devices—the iOS version of Latrunculi [21] offers fewer AI difficulty levels to avoid exhausting the device memory. Similarly, Adobe's Photoshop editor, which utilises over 2 GB of RAM in the desktop version [22], has a mobile version, Photoshop Touch, that imposes constraints on image sizes due to insufficient device memory [23].

**Specialising mobile applications.** As more features are added to mobile applications over time, their growth also increases their memory footprint. Eventually, the application may have to be split into multiple more specialised versions, each focusing on a subset of the original functionality. As evidence for this trend, popular feature-heavy mobile applications such as the Facebook, LinkedIn and Foursquare clients have begun splitting themselves into multiple unifunctional applications [24]. For example, Facebook recently removed the messaging component from its client application and released it as a stand-alone application, thus reducing its application footprint considerably [25].

**Using a client/server model.** Mobile applications that adopt a *client/server model* can exploit the memory resource of remote servers. Developers must however decide on a split between the client- and server-side for an application. This requires them to understand the resource requirements of different application components, and judge the amount of network traffic exchanged.

For simplicity, developers often adopt an extreme when splitting an application between client and server functionality: they place *all* functions on the remote server, thus turning the mobile application into a *thin client* [26]. A thin-client model, however, requires network communication for all functionality, even if it could have been implemented on the mobile device only. We already witness that, without network connectivity, a large class of today's smartphone applications cease to function [27].

## 3 APPLICATION STATE PARTITIONING

C-RAM helps developers create new mobile applications or port existing desktop applications to a mobile platform without having to worry about restrictions on the available device memory. It automatically partitions application state across the mobile device and a remote server, thus allowing the application to have a memory footprint beyond the capacity of the device.

C-RAM performs automatic source code rewriting to create a partitioning of an Objective-C application. Objective-C applications are composed of a set of classes. Each class contains a number of fields and methods, which access these fields. Classes are instantiated as objects at runtime, which constitutes the application state. C-RAM partitions applications at a *class* granularity, i.e. all objects of the same class are assigned to a given partition. By distributing objects among a local and remote node, C-RAM splits the application state and distributes the computation.

As shown in Figure 1, C-RAM has three main components: (1) the *Profiler* collects information about the memory and CPU consumption of application objects; (2) this is used by the *Partitioner* to derive an efficient partitioning that utilises the remote memory; and (3) the *Compiler* realises the partitioning by rewriting the application source code. Next we discuss these components in more detail.

### 3.1 Dynamic Resource Profiling

C-RAM uses dynamic profiling to measure the memory and CPU consumption of the application and to identify the call graph dependencies of an application under a set of workloads. The Profiler collects information about (i) the amount of memory used by objects of each class and (ii) the average execution times of object methods on both the local and remote nodes.

A *profiling run* $p$ relates to one particular execution trace in a workload. Multiple profiling runs $P$ capture the application behaviour for different inputs, chosen to represent average workloads. C-RAM does not attempt to explore all possible workloads exhaustively, but rather to identify parts of the application that are consistently memory-intensive and thus would benefit from remote placement.

C-RAM profiles each application twice: a *local profiling run lp* records application behaviour on the local device; a *remote profiling run rp* executes the application using a device emulator on the remote node. This is used to quantify the memory that is consumed by the different application objects. It also provides an upper bound on the performance that a partitioning may achieve when the whole application executes remotely, ignoring communication overheads.

**Profiling methodology.** The Profiler uses two techniques to obtain profiling information. First, it utilises DTrace [28], an efficient framework for fine-grained dynamic tracing, to collect information about the memory consumption of objects and interactions with platform-native functionality, such as GUI libraries or the hardware sensors. DTrace introduces probes, i.e. points of instrumentation, in supported OS kernels. Probes can be composed to create custom probes that are queryable at runtime. Custom probes are compiled to binary code and patched dynamically into a kernel, with only a modest runtime overhead. The Profiler creates custom probes to log the memory activity of objects and identify native method callers. This technique is used only during remote profiling runs because applications that exhaust the available device memory are forcibly terminated on the mobile device.

Second, the Profiler automatically adds instrumentation to the application source code to measure the duration of each method. To correctly account for the time spent in nested method invocations, it uses a per-thread global stack to record durations. At the start of each method, the contents of the call stack are used to construct a class call graph, which is output on completion of a profiling run.

The latter technique is used during both the local and remote profiling runs. The goal is to compare the execution times of individual methods on each type of node to infer the impact of offloading on application performance. Such a comparison is only meaningful for application workloads that do not exceed the available device memory. To assess the accuracy of this technique, we measured the impact of the code instrumentation on application performance and observed a modest 8% overhead.

**Profiling output.** The primary output of each profiling run $p$ is a *memory consumption relation $M$*. Given an application class $x$ and a time $t$, $M(x,t)$ denotes the amount of memory consumed by all objects of class $x$ at $t$. This information is used to ensure that the total amount of memory consumed by objects assigned to the local node does not exceed its memory capacity.

The Profiler also outputs a *CPU consumption graph*, $G = (C, T^L, T^R, E)$, where $C$ is a set of application classes, $T$ provides execution times for classes, and $E$ specifies call dependencies. For a class $x \in C$, $T^L(x)$ and $T^R(x)$ return the average execution times of class $x$ on the local and remote nodes, respectively. $E(x,y)$ is a pair $(calls_{x,y}, data_{x,y})$ that states that an average of $calls_{x,y}$ calls from methods in class $x$ to $y$ occurred, and each call used an average of $data_{x,y}$ bytes for its arguments.

In the CPU consumption graph, a special node called *native* amalgamates all library functions that offer I/O and device-specific functionality. A list of these functions is created manually once for a given platform such as iOS. The Profiler uses DTrace to identify objects that call native functions and thus cannot be placed remotely by C-RAM.

## 3.2 Partitioning Algorithm

C-RAM assigns each application class to a *local* or *remote* set, which contain the classes hosted by the local and the remote node, respectively. Based on the Profiler output, the Partitioner decides how to partition classes so that the memory consumed by local objects during execution does not exceed the available device memory.

In addition, the Partitioner considers the execution times of each object on both the local and remote nodes, as well as the communication cost that each possible partitioning entails. The goal is to provide the illusion of effectively "unlimited" local memory by ensuring that the partitioned application has a performance that is no worse than that of the original version. To assess the ability of a partitioning to achieve this goal, the Partitioner compares the performance of a partitioned version and the original application for workloads that do not exhaust the available device memory.

Given the above, a valid partitioning must satisfy the following three constraints:

**Constraint 1:** An application class belongs either to *local* or *remote*, but not both.

**Constraint 2:** The local node's memory capacity *mem* is sufficient to accommodate all local objects. (We assume that the remote node's capacity is effectively unlimited.)

$$\forall rp \in P, \ \forall t \sum_{x \in local} M(x,t) < mem$$

**Constraint 3:** Classes in *remote* do not call native library functions only available at the local node:

$$\forall x \in remote \ \nexists E(x, native) \land \nexists E(native, x)$$

The Partitioner derives a set of valid partitionings $V$, which contains pairs of the form $v = (local, remote)$. It then selects the partitioning $v \in V$ that minimises the total execution time $O$, in an attempt to mask the network overhead due to the partitioned execution.

We use a function $T_{net}(x)$ to express the time needed to make a remote call over a network $net$ such as Wi-Fi or 3G, with $x$ bytes for its arguments. $T_{net}$ is derived experimentally by benchmarking the execution time of remote calls with different arguments. This, combined with the CPU profiling results, is used to estimate $O$ as follows:

$$O(v) = \sum_{x \in local} T^L(x) \ + \sum_{y \in remote} T^R(y) \ +$$
$$\sum_{\substack{x \in local \\ y \in remote}} calls_{x,y} \times T_{net}(data_{x,y}) \ + calls_{y,x} \times T_{net}(data_{y,x})$$

The execution time $O$ for a partitioning $v$ is the sum of: (i) the execution time on the local node, i.e. the sum of all $T^L(x)$ for classes in *local*; (ii) the execution time on the remote node, i.e. the sum of all $T^R(y)$ for classes in *remote*; and (iii) the communication delay of calls between objects residing on different nodes. This is equal to the number of calls between any such pairs of classes, $calls_{x,y}$, multiplied by the average delay of each call, i.e. $T_{net}(data_{x,y})$. The Partitioner selects the best partitioning using a standard integer linear programming (ILP) approach.

## 3.3 Application Transformation

Next we describe how the Compiler transforms the Objective-C application to implement a given partitioning. It splits the source code into a *local* and a *remote* code partition that are deployed on the local and remote nodes, respectively.

**Proxy objects.** Objects interact transparently across partitions using proxy objects. A proxy object relays method calls to an object residing on the other partition through remote procedure calls (RPCs) [10]. An RPC sends a request message to the remote node, which executes the target method with the supplied parameters. While the remote node executes a call, the caller thread blocks waiting for a response message. Proxy objects are small, and their size is independent of the size of the remote object.

After state partitioning, an object either resides on a partition or is represented by a proxy object. As a consequence, pointers to objects passed as arguments to a remote call
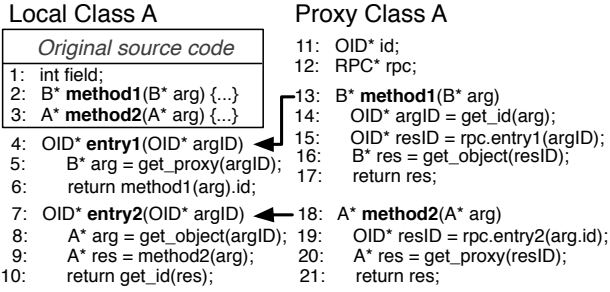
```
Local Class A                    Proxy Class A
┌─────────────────────────┐      11:  OID* id;
│   Original source code   │      12:  RPC* rpc;
├─────────────────────────┤
1:  int field;             ┌──── 13:  B* method1(B* arg)
2:  B* method1(B* arg) {...}│      14:    OID* argID = get_id(arg);
3:  A* method2(A* arg) {...}│      15:    OID* resID = rpc.entry1(argID);
                            │      16:    B* res = get_object(resID);
4:  OID* entry1(OID* argID) ◄      17:    return res;
5:    B* arg = get_proxy(argID);
6:    return method1(arg).id;
                                   18:  A* method2(A* arg)
7:  OID* entry2(OID* argID) ◄───── 19:    OID* resID = rpc.entry2(arg.id);
8:    A* arg = get_object(argID);  20:    A* res = get_proxy(resID);
9:    A* res = method2(arg);       21:    return res;
10:   return get_id(res);
```

Fig. 2: Example of a partitioned class

```
1:   RPC* requestRPC (string class_name)
2:     switch(class_name) {
3:       case "A":
4:         A* obj = /* allocate new class A object */;
5:         RPC* rpc = /* create new class RPC object for obj */;
6:         return rpc;
7:       case "B": ...
8:   void release (int objID)
9:     void* obj = get_proxy(objID);
10:    if (obj == NULL) obj = get_object(objID);
11:    obj.release();
12:  void retain (int objID)
13:    void* obj = get_proxy(objID);
14:    if (obj == NULL) obj = get_object(objID);
15:    obj.retain();
```

Fig. 3: Allocator class for managing object life-cycles

must be converted by the callee to the associated local or proxy objects. The same applies to pointers returned by remote calls, which must be converted by the caller.

To make these conversions, the Compiler assigns a unique *object identifier* to each object on creation. The object identifier is a pair of values: a boolean identifying the partition and the memory address on that partition. Objects can be referenced in cross-partition interactions through a global *identifier table*, which maps identifiers to local and proxy objects. When processing RPC calls, the referred objects are retrieved based on their object identifiers.

**Source code rewriting.** The Compiler rewrites classes as shown in Figure 2. In this example, we consider two classes, A and B, that interact across partitions. Class A contains two methods, *method1* and *method2*, which accept one parameter each: pointers to class B and class A objects, respectively; *method1* returns a pointer to a class B object and *method2* returns a pointer to a class A object. The figure illustrates how class A is transformed into a *local* class and a *proxy* class, which are placed on the local and remote nodes, respectively.

In general, *local classes* retain their original fields and methods (lines 1–3), with some modifications to handle incoming remote calls. Each method receives a corresponding *entry point method* to serve incoming RPC requests. Given an object identifier, it retrieves required local and proxy objects from the identifier table and invokes the method of the local object (lines 4–10). *Proxy classes* contain the object identifier (line 11) of the object residing on the other partition, and a reference *rpc* to an RPC object used to initiate remote calls (line 12).

Our C-RAM prototype uses the *Internet Communications Engine* (ICE) [29], an object-oriented RPC toolkit with support for Objective-C. It automatically generates RPC classes with the same method signatures as the underlying classes to serialise and deserialise method parameters. In the proxy class, the two methods are replaced with wrappers that execute the remote calls. The remote calls are invoked through the RPC object, after translating method parameters with object pointers to the corresponding object identifiers (lines 14 and 19). Returned pointer values are translated to local (line 16) or proxy objects (line 20) using the identifier table before returning them to the caller.

**Object life-cycle.** In Objective-C, the lifetime of an object is managed by means of reference counting. *NSObject* is the root class of most Objective-C class hierarchies, providing basic functionality. One of *NSObject*'s fields is *retainCount* and denotes the number of ownership claims on an object. When a new object is allocated, it has a *retainCount* of one. When a method acquires ownership of an object, it calls its *retain* method (inherited from *NSObject*) to increment the *retainCount*; relinquishing ownership is done by calling *release*. When *retainCount* becomes zero, the object's *dealloc* method is called automatically to free the allocated memory.

This reference counting mechanism, however, is unaware that C-RAM distributes objects across the local and remote nodes. To allocate and deallocate partitioned objects correctly, the Compiler therefore synthesises a new *Allocator* class, shown in Figure 3. It provides a *requestRPC* method that, given a class name, allocates a new object on the remote partition and returns the associated RPC object (lines 1–7). This object is then wrapped by the corresponding proxy class on the partition issuing the request. Calls to *release* and *retain* on local and proxy objects are intercepted and forwarded to the associated proxy and local objects across partitions. This is done via the corresponding methods provided by the *Allocator* class (lines 8–15).

**Library access.** C-RAM treats shared iOS library classes, whose source code is not available, the same way as custom application classes. Since Objective-C uses late-binding, the implementation of library methods can be replaced at runtime using *method swizzling* [30]: when the Objective-C runtime loads a binary, all objects have their fields and method implementations defined in memory, along with a map associating method names with implementations. These mappings can be modified at runtime by replacing the original implementation with a user-defined function. C-RAM is thus able to replace existing methods with replacement methods that relay method invocations to the corresponding library objects residing on the remote node, as described in §3.3.

### 3.4 Discussion

Objective-C is a superset of the C programming language, and, at present, C-RAM only supports object-oriented Objective-C code, i.e. code that does not utilise pure C data structures to maintain application state. If application state is stored in C data structures, it may be accessed directly using arbitrary pointers, e.g. by adding a byte offset to a pointer. Such direct state access would complicate the Compiler's task of identifying all accesses to remote

application state in order to relay them to the appropriate node during execution. In future work, we plan to extend the Compiler to generate automatically wrapper classes for C data structures, converting pointer references to object method calls.

## 4 SNAPSHOT-BASED FAULT TOLERANCE

Partitioned application state makes it hard to mask network failures. To revert to local execution after failure, it is necessary to recover remote state after it becomes inaccessible. C-RAM employs a fault tolerance mechanism in which the local node periodically retrieves a snapshot of the remote state, which is used to resume execution after failure.

This raises several challenges: (i) how to take *consistent* snapshots across local and remote state; (ii) how *frequently* to take snapshots, balancing freshness with overhead; (iii) how to *implement* snapshots in Objective-C without changes to the runtime system; and (iv) how to handle remote state after failure that is *larger* than the available memory of the device.

### 4.1 State Snapshots

A complete snapshot of the application state consists of a *local* and a *remote* snapshot. A *local snapshot* has the information needed to resume execution from a consistent, well-defined point. It includes: (a) all active objects on the heap; (b) the state of all machine registers, obtained using inline assembly code; (c) the current user call-stack; and (d) a table with object identifiers for all local objects. The local snapshot is written as a byte array to flash memory. Execution can resume from a local snapshot by overwriting the allocated heap, machine registers and call-stack with the data from the snapshot.

A *remote snapshot* is a byte array of serialised remote objects, including: (a) their object identifiers; (b) the values of all primitive fields; and (c) the object identifiers of fields pointing to other objects. Previously remote objects can be recreated locally from a remote snapshot as follows: (1) primitive data fields are restored with the values from the snapshot; (2) fields with pointers to other previously *remote* objects are set to the corresponding object identifiers so that the objects can be loaded on-demand (see §4.5); and (3) fields with pointers to *local* objects, which used to point to proxy objects on the remote node, are assigned addresses of the corresponding local objects based on the object identifiers from the snapshot.

### 4.2 Snapshotting Mechanism

Local and remote snapshots must be taken at exact execution points, i.e. before control of execution is transferred to and from the remote node via remote calls. This ensures that the local node always has a consistent view of the entire application state (local and remote).

C-RAM's snapshotting strategy, however, decouples the transmission of snapshots from remote calls and permits an arbitrary snapshot transmission frequency. The benefit of



Fig. 4: C-RAM's snapshot-based fault tolerance mechanism

this approach is that when frequent remote calls modify a large amount of remote memory, C-RAM can be configured to reduce the impact of increased snapshot transmission delays on application response time. This, however, comes at the cost of potentially losing the most recent application updates after failure. Since the most recent remote snapshot available locally may be inconsistent with the current local state, C-RAM may need to roll back the application to an earlier point in time.

**Snapshot creation.** A remote snapshot is taken when execution returns from the remote node to the local node. A challenge is that remote calls may be *nested*, i.e. a remote call may in turn execute a new call passing control back to the local node. A local node is said to have *control of execution* when it has no remote calls in progress, otherwise execution is controlled by the remote node. Nested calls do not transfer control.

After failure, the latest remote snapshot received by the local node may be inconsistent with the local state. Therefore, C-RAM also takes a local snapshot before transferring control of execution to the remote node. This is then used to *roll back* the local state to that snapshot, before rebuilding the remote state to restart local execution after failure.

**Snapshot transmission.** C-RAM's snapshotting mechanism relaxes the requirement that each transfer of control must include the transmission of the corresponding remote snapshot. Instead, a *background thread* periodically transfers updated remote snapshots at a configurable frequency.

After a failure, C-RAM periodically checks if connectivity to the remote node was restored. When this is the case, it takes a snapshot of the objects that were originally part of the remote partition and sends it to the remote node. The remote node then updates pointers between objects across partitions using proxy objects. Finally, remote code offloading is enabled again.

**Example.** The operation of the snapshotting mechanism is shown in Figure 4. The local node initiates three remote calls $R_1$–$R_3$. The background thread transfers the latest remote snapshot to the local node every $T$ seconds. Remote snapshots $RS_1$ and $RS_2$ are taken at $t_6$ and $t_{14}$, respectively, before control of execution returns to the local

node. This ensures that they are consistent with the local snapshots taken earlier (at $t_3$ and $t_9$).

Before the remote call $R_1$ returns, a remote snapshot $RS_1$ is taken at $t_6$. Note that local execution continues at $t_8$, before the transfer of $RS_1$ finishes at $t_{13}$. The transfer of $RS_1$ is concurrent with the next remote call $R_2$ at $t_{12}$, which returns at $t_{16}$.

Consider the case in which the next remote call $R_3$ fails at $t_{18}$. The corresponding remote snapshot $RS_2$ was taken at $t_{14}$ but was not yet transferred to the local node before the failure. At this point, the local node must reconstruct the remote objects based on the last available remote snapshot, which is $RS_1$. The local node thus reverts to the consistent corresponding local snapshot, which in our example is the local snapshot $LS_2$ from $t_9$. Any updates to the local state after $t_9$ are not reflected in the remote snapshot $RS_1$ and have to be discarded. Based on the snapshots $RS_1$ and $LS_2$, the local node has access to the entire application state, effectively reverting application execution back to $t_{10}$.

## 4.3 Snapshot Frequency

By setting $T = 0\,s$, remote snapshots are transferred to the local node *synchronously* after each remote call, which ensures that the local node always has an up-to-date copy of all remote objects. The advantages of opting for a non-zero snapshot frequency $T$ are twofold: (1) applications can transmit remote snapshots asynchronously to the local node, in parallel with any computation that takes place on either the local or remote node. The transmission cost is thus amortised over time, reducing its impact on application response time; (2) potentially less snapshotting data must be sent because multiple updates to the same object may be combined and only transferred once. The larger the snapshotting frequency $T$, the greater these benefits become. A large value of $T$, however, also results in potentially more application state being lost after failure.

## 4.4 Incremental State Snapshots

To reduce the size of snapshots, C-RAM creates *incremental* snapshots that only include modified objects since the last snapshot. As a result, an individual remote snapshot is no longer self-contained but depends on all previous remote snapshots when reconstructing the latest version of the remote node state during recovery.

For incremental snapshots, the Compiler statically analyses the source code to identify when object state is updated: for primitive data and pointer fields, updates are caused by assignments; for collection data structures, such as arrays and dictionaries, updates also include the addition or removal of elements. The Compiler synthesises code that, when a new object is allocated or its state is modified, it is added to a global list of modified remote objects, $dirtyObjs$. Object deallocations are modified to remove entries from this list. When a remote snapshot is taken, only objects in $dirtyObjs$ are included in the snapshot. For easy reference, each remote snapshot contains a hash table that maps each

object identifier to the remote snapshot with the most up-to-date version of that object.

Since creation and transmission of snapshots are decoupled, multiple incremental snapshots may have to be merged before transmission to avoid sending old state. C-RAM uses a global $snapshotList$ to store remote snapshot entries per object that are pending transmission to the local node. The list is updated when object state changes and new snapshots are taken.

## 4.5 Virtual Memory Scheme

When reverting to local execution after failure, the application state may be larger than the device memory. The iOS platform does not support virtual memory and instead terminates applications in low memory conditions.

To overcome this issue, C-RAM realises a *virtual memory scheme* at user level. Rather than restoring all objects locally before resuming with local execution, the Compiler synthesises code that loads remote objects from a snapshot on demand. Such objects are added to a $loadedObjs$ table, indexed by their object identifiers. The Compiler statically identifies points in the code at which methods of a remote object are invoked. Before such an invocation, it adds checks if, based on the object identifier, the object exists in $loadedObjs$. When the caller is a proxy object, the object identifier is passed in the $id$ field; if the caller is another instantiated remote object, it is derived from the object identifier stored in the caller's field that used to point to the called object in the remote node's address space. If the remote object exists in $loadedObjs$, the call proceeds; otherwise, the remote object is loaded from the remote snapshot and added to $loadedObjs$.

The total size of objects is limited by the amount of available device memory. When memory is exhausted, an object from $loadedObjs$ is evicted from main memory and potentially written back to flash memory. In our prototype implementation, objects are evicted at random subject to two rules: (1) objects that are on the call stack must not be evicted for correctness; and (2) priority is given to evict "clean" objects, i.e. objects that have not been modified, to avoid the cost of writing them back to flash memory. Dirty objects are identified as part of the incremental snapshotting technique from §4.4. Although more efficient eviction policies exist, e.g. taking temporal and spatial locality of references into account [31], [32], we defer their exploration to future work.

## 4.6 Network Usage Throttling

State snapshotting can cause a significant increase in network usage for applications that periodically update large amounts of remote state. While the corresponding impact on response time and energy consumption can be controlled by a tunable snapshot interval (see §5), users on metered mobile data plans would still be required to pay more when using C-RAM.

To address this problem, C-RAM has a mechanism for *network usage throttling*: offloading is disabled when

the traffic caused by C-RAM exceeds a threshold that represents a user's limit on network usage. The background thread that transmits remote state snapshots to the device also maintains the total network usage of the offloaded application, and transmits snapshots only if the network usage is below the specified threshold; otherwise, C-RAM reverts to local execution using its virtual memory scheme.

### 4.7 Discussion

The snapshot-based fault-tolerance mechanism of C-RAM assumes that only a single partition executes at any point in time. This ensures that local and remote snapshots are consistent with each other because remote calls synchronise the local and remote state. This is not the case, however, for multi-threaded applications, in which different threads execute on both the local and remote nodes in parallel. This means that state updates caused by local threads and incoming remote calls cannot be distinguished, necessitating a more complex technique such as log-based recovery [33].

Our mechanism also assumes that applications have no external side-effects such as network communication or file I/O. Since C-RAM does not record which operations of a method call that are not reflected in the state snapshots executed successfully before failure, all such operations are repeated on the local node during recovery. This may lead to unexpected behaviour for operations with side-effects.

Operations with side-effects typically use dedicated iOS library APIs, which C-RAM could be made to identify, similar to how platform native method calls are handled as part of the profiling step (see §3.1). In future work, we plan to extend C-RAM to record operations with side-effects and avoid repeating them during failure recovery.

## 5 EVALUATION

We evaluate C-RAM experimentally to investigate: (1) its effect on applications that consume large amounts of memory during execution; (2) how application response times are affected by application state partitioning using C-RAM; (3) the effect that C-RAM has on network and energy usage; and (4) the overheads of C-RAM's fault tolerance mechanism and its user-level virtual memory scheme.

### 5.1 Experimental Set-up

For our experiments, we use an Apple iPhone 4S as the local node and a 2.26 Ghz Intel Core 2 Duo machine with 8 GB of RAM as the remote node. We use an 802.11 WiFi network with an average round trip time (RTT) of 40 ms and a bandwidth of 6.3 Mbps, and a 3G network with an average RTT of 523 ms and a bandwidth of 0.4 Mbps.

We show how C-RAM enables developers to implement new or port existing memory-intensive applications to mobile platforms assuming unlimited device memory by applying C-RAM on three Objective-C applications. The first two were ported from Mac OS to the iOS platform, which only affected UI classes; the third was designed and implemented as a monolithic mobile application.

**Latrunculi board game.** Latrunculi [20] is an open-source board game, in which two players alternate moving pieces until all opponent's pieces have been captured. Its AI component uses the *minimax* algorithm to search a tree of future moves for the best next move. The search depth is configurable and determines the strength of the game AI.

**iSpreadsheet application.** This application operates on CSV files and supports a range of features, including loading/saving and editing spreadsheets; sorting by row/column; and calculating complex user-defined formulas over multiple cells with operations such as sum, average and median. Loading a spreadsheet file parses the CSV data and creates objects for each cell; saving generates the corresponding file from the spreadsheet objects. The application manages arbitrary amounts of state depending on the size of the spreadsheet file.

**Content-based image retrieval (CBIR) application.** This application uses computer vision techniques to retrieve images based on user queries. It can be used to identify similar products at a point-of-sale or faces in a video stream, similar to systems such as Picalike [34], Empora [35] or eBay's Image Search [36]. The CBIR application searches for images in a large in-memory database [37] based on an input image. Using an in-memory database allows for almost instant responses to search queries.

We apply C-RAM to the above applications, generating partitionings based on the profiled workloads. For the Latrunculi game, the profiling workload consists of a complete game play for different AI search depths. C-RAM creates a remote partition with the classes responsible for maintaining the board and the game AI.

The profiling workload for the iSpreadsheet application includes loading 2 MB–14 MB spreadsheets, sorting them, calculating formulas and saving them. The obtained partitioning places the classes responsible for the loading/saving functionality (i.e. parsing and storing CSV files in application objects and exporting the object representation of a spreadsheet to CSV format, respectively), sorting and calculation of formulas on the remote node.

Note that, since the spreadsheets are initially stored in CSV format on the device's flash memory, the loading and saving of the spreadsheet under this partitioning requires the transmission of the CSV data to and from the remote node. The transmission cost for this, however, is low because the CSV data is significantly smaller than the object representation of the spreadsheet.

For the CBIR application, the profiling workload consists of multiple queries by image content using variable-sized image databases. C-RAM places the classes responsible for maintaining the pre-processed image database and the processing of user queries on the remote node.

### 5.2 Memory Utilisation

First we observe the memory utilisation of the partitioned and unpartitioned versions of the above applications. Figure 5 shows how C-RAM mitigates device memory constraints by partitioning application state. We plot the
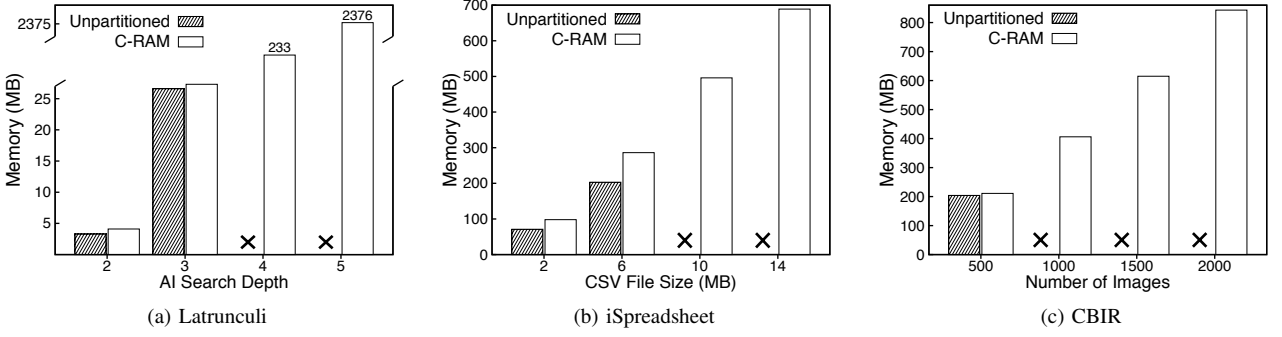
Fig. 5: Maximum memory utilisation for different application workflows (with and without C-RAM)

maximum memory utilisation of all three applications as a function of different application parameters that affect memory consumption. For the Latrunculi game, we vary the difficulty level by changing the AI search depth; for the iSpreadsheet application, we consider different input CSV files that are loaded—ranging from 2 MB–14 MB; and, for the CBIR application, we vary the amount of images included in the in-memory database, against which user queries are executed. We compare memory utilisation for (i) the *unpartitioned*; and (ii) the partitioned application versions using *C-RAM*.

For the Latrunculi game (Figure 5a), the memory utilisation increases exponentially due to the increasing complexity of the game AI. For a search depth above three, the unpartitioned application is terminated due to insufficient memory; using C-RAM, the game continues to run, exploiting the larger memory of the remote node.

For the iSpreadsheet application (Figure 5b), we measure the memory footprint when loading a spreadsheet from CSV files. The unpartitioned version cannot process CSV files larger than 6 MB due to a lack of memory, while the partitioned version continues to work.

Figure 5c shows the memory consumption of the CBIR application with different image database sizes. The unpartitioned version can only support up to 500 images, limiting the search space for user queries. Using C-RAM, however, the application can support an arbitrary number of images, which is only limited by the memory available on the remote node.

For all applications, C-RAM's peak memory utilisation slightly exceeds that of the unpartitioned version. This increase is due to the additional metadata needed by the snapshotting mechanism (see §4.1). It depends on the number of remote memory objects rather than the actual size of the remote application state. For both the Latrunculi game and the CBIR application, the peak memory utilisation increases by at most 12% compared to the unpartitioned versions. For the iSpreadsheet application, which has a large number of remote objects (in the order of tens of thousands of small spreadsheet cell objects), the increase is 28%. Note that this does not affect the local device because any metadata is maintained by the remote node, which is not memory-constrained.

**Discussion.** The above results show the effectiveness of

C-RAM in supporting memory-intensive applications on mobile devices, without having to explicitly change their design. The memory footprint of an application running on, for example, an Apple iPhone 4S is no longer limited to 210 MB, but, as shown in Figure 5, can exceed 2 GB of used memory. In doing so, C-RAM improves user experience for all three applications: it enables a stronger game AI, supports the processing of larger CSV files and allows search over more images.

## 5.3 Application Response Time

We also investigate the performance impact of C-RAM and its snapshot-based fault tolerance mechanism. Application performance is calculated based on the average time to respond to a series of user actions. For the Latrunculi game, we measure a single move with a given AI depth; for the CBIR application, we measure the time to conduct an image search over a fixed-size image database. We consider two workflows for the iSpreadsheet application: a *simple workflow* in which a user loads a spreadsheet; and a *complex workflow* in which a user loads a spreadsheet, sorts it by a row and then sorts it by a column. For all experiments, response time is measured over both WiFi and 3G network connectivity; the results correspond to averaged values over ten experimental runs.

First, we compare the performance of the unmodified applications to that of the partitioned versions using C-RAM. For these experiments, the workflows for each application are chosen such that they can also execute entirely on the device, i.e. without exhausting the device memory, for which a performance comparison is possible. In particular, we use an AI search depth of three for the Latrunculi game, a 4 MB CSV file for the iSpreadsheet application, and an in-memory database of 500 images for CBIR.

Second, we compare C-RAM against an approach that uses virtual memory with on-demand paging to enable applications to consume memory that exceeds the capacity of the device. Since by default iOS does not support kernel-level virtual memory, we instead use C-RAM's virtual memory scheme, which is implemented in user space. For this set of experiments, we use an AI search depth of 4 for Latrunculi, a 10 MB CSV file for iSpreadsheet and 1000 images for CBIR, all of which exhaust the device memory. **Comparison with original applications.** Figure 6 shows the speed-up achieved using *C-RAM* with different snapshot

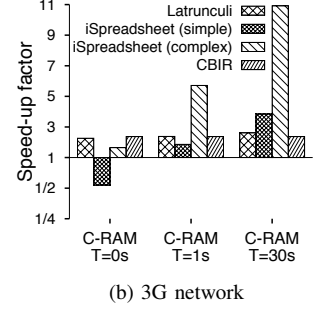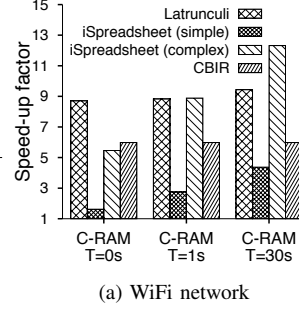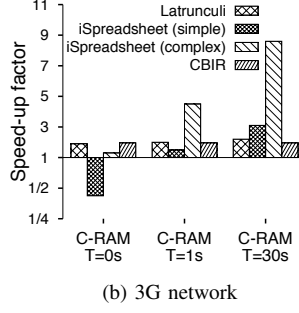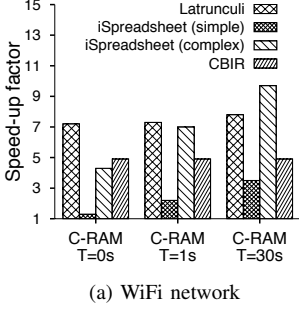(a) WiFi network     (b) 3G network     (a) WiFi network     (b) 3G network

Fig. 6: C-RAM vs. Original applications (response time)     Fig. 7: C-RAM vs. Virtual memory approach (response time)



(a) Latrunculi     (b) iSpreadsheet (simple)     (c) iSpreadsheet (complex)     (d) CBIR
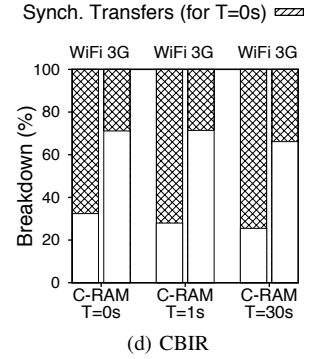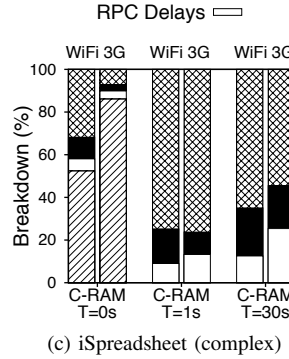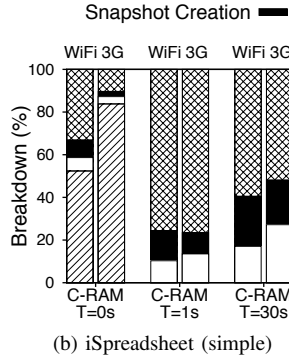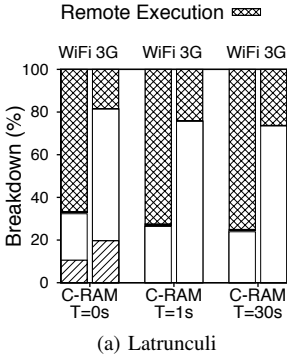
Fig. 8: Breakdown of application response times

frequencies $T$ and for different application workflows. In Figure 8, we break down the response time in terms of the time spent on (a) remote code execution; (b) local code execution; (c) state snapshot generation; (d) RPC invocations; and (e) synchronous transfers of remote snapshots (when $T=0$ s). The asynchronous transfers of remote snapshots (when $T>0$ s) are not included because they occur concurrently with remote execution—users witness the effects of a remote call before asynchronous transfers are completed. We will show that, in practice, some of the overhead of asynchronous transfers is reflected in the remote execution times (see Figure 9).

Configuring C-RAM with $T=0$ s gives a lower bound on the attainable speed-up because it transmits snapshots of all remote state updates in a synchronous fashion. As shown in Figure 6, with just one exception (the simple iSpreadsheet workflow over 3G), application performance still improves despite the additional transmission delays of snapshots. Over WiFi, the speed-up is higher than the corresponding speed-up over the 3G network, which is due to the higher latencies of RPC calls and smaller bandwidth of the 3G network: from $1.3\times$ to $7.2\times$ for WiFi and from $0.4\times$ to $2\times$ for 3G. The break-down in Figure 8 confirms that, for $T=0$ s, the network delays associated with synchronous snapshot transmissions contribute significantly to the overall application response times.

We repeat the experiment with $T=1$ s and $T=30$ s, representing frequent and sporadic snapshots, respectively. Consistently, a non-zero $T$ outperforms $T=0$ s by decoupling the transmission of snapshots from remote execution: from $3.5\times$ to $9.7\times$ for WiFi and from $2\times$ to $8.6\times$ for 3G. This

is due to the fact that the remote node continues to serve remote calls in parallel to any asynchronous transmissions of remote snapshots. For example, multiple calls related to UI updates that occur after loading a spreadsheet can be handled by the remote node before the transmission of the large snapshot has completed.

For some applications, namely the Latrunculi game and the CBIR application, the relatively modest performance improvement of using a non-zero $T$ does not justify the use of asynchronous checkpoints and the associated potential loss of state updates after failure. For the Latrunculi game, the amount of modified remote state after each board move is relatively small. (The state allocated by the game AI for the tree search—although large—is only transient and therefore not included in the snapshots.) The same applies to the CBIR application, which does not modify any non-transient remote state during the execution of a search query. This can be seen in Figures 8a and 8d, respectively, which show that the performance overhead due to snapshot creation and synchronous transfers is relatively small.

In contrast, for the iSpreadsheet application with the simple workflow, setting $T=0$ s can lead to a slow-down in performance (see Figure 6b). This is caused by the transmission cost of the large remote snapshots, which include the entire spreadsheet data (see Figure 8b).

**Comparison with pure virtual memory approach.** Next we compare C-RAM against an approach that uses only C-RAM's virtual memory scheme to allow an application to consume more than the available device memory. Figure 7 shows that, across all our applications and workloads, C-RAM achieves a significant speed-up because the
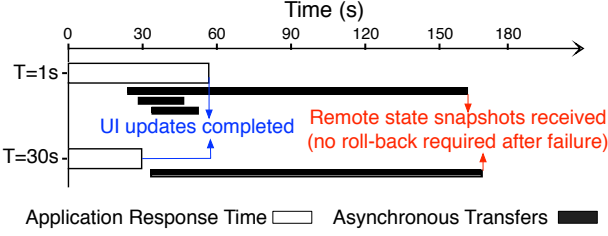
Fig. 9: Effect of $T$ for iSpreadsheet (complex)



Fig. 10: Effect of C-RAM on device energy consumption

virtual memory scheme introduces a 19%–28% overhead compared to the unmodified application execution. Virtual memory with on-demand paging carries a fundamental overhead, mostly due to the time to load and store application objects from/to flash memory during execution. This is a reason why current mobile platforms do not offer support for on-demand paging to work around memory limitations.

**Snapshot frequency.** The choice of the snapshot frequency $T$ has implications on application performance. Figure 9 shows the application response time for the iSpreadsheet application (complex workflow) over 3G with varying snapshot frequencies $T$. The black bars indicate the duration of asynchronous snapshot transfers relative to the application response time.

With a larger snapshot interval ($T=30$ s), the speed-up is higher because remote execution is no longer concurrent with snapshot transmission, thus improving the performance of the remote node. In addition, with higher snapshot frequencies, overlapping updates to remote objects are combined to obtain the latest version of the state, further reducing the transmitted data.

For $T=1$ s, three distinct remote snapshots are taken, i.e. one for each of the load, sort-by-row, and sort-by-column operations, leading to three asynchronous transfers. With $T=30$ s, only one remote snapshot is transferred, after all three operations have completed. However, using a higher snapshot frequency has the downside that execution may need to roll-back more during failure recovery because snapshots are received later by the local node (as indicated by the end times of the black bars).

**Discussion.** C-RAM's goal is to allow applications to utilise memory that exceeds the available device memory. We showed that, in contrast to an approach that merely uses a virtual memory scheme, C-RAM achieves this goal while reducing application response times: it benefits from the faster CPU on the remote node to mask the additional communication overhead due to its fault tolerance mechanism.

Different applications require different snapshotting frequencies. For applications that do not modify large amounts of state during remote execution, a snapshotting frequency $T=0$ s is almost as efficient as $T>0$ s. Since the latter may require rolling back execution after failure, such applications should use the former setting.

The asynchronous transmission of snapshots ($T>0$ s) yields better performance for applications that may potentially create large state snapshots by updating remote state frequently. It decouples the transmission of snapshots
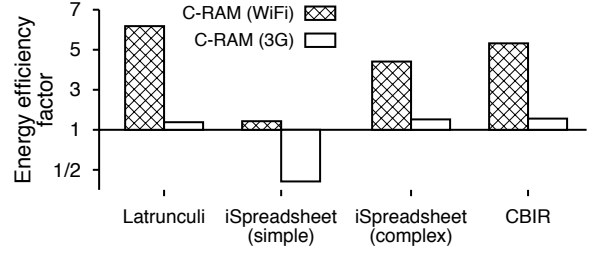
from remote execution. In general, the best strategy to be used for a given application can be decided during a post-partitioning profiling run when the application's behaviour with regards to remote state updates can be observed.

## 5.4 Energy Usage

To investigate C-RAM's impact on energy consumption, we use Apple's *Energy Usage Instrument* [38]. This tool captures information about the energy consumed on a device and outputs the relative energy usage. For this set of experiments, we set $T=0$ s to account for all snapshot transfers from the remote node to the device.

In Figure 10, we compare the energy usage of C-RAM with that of the unmodified applications for the same workloads as used in §5.3. During snapshot transmission and RPC calls, the energy usage level is approximately 12% and 21% (over WiFi and 3G, respectively) higher than during local execution. Nevertheless, we observe that, in the majority of cases, C-RAM is overall more energy efficient than the original applications because execution time is significantly reduced. In particular, over WiFi, C-RAM is able to achieve reductions in energy consumption by 1.4–6.2×. Energy savings diminish when using a 3G network (at most 1.6×) due to the fact that 3G consumes more energy and increases transmission delays.

In some cases, such as iSpreadsheet's simple workflow over 3G, energy consumption can increase considerably using C-RAM. This is due to the transmission of a large snapshot that contains the entire spreadsheet data when loading a spreadsheet initially. For this simple workflow, the energy savings due to remote execution are not enough to compensate for the associated snapshot transmission cost. Nevertheless, usually this overhead is a one-time cost that is followed by compute-intensive processing on the loaded data, and is therefore masked over time by the savings due to remote execution (as in the case of the complex workflow, which is a superset of the simple workflow).

**Discussion.** Although C-RAM primarily aims to extend the memory of mobile devices, it can also save energy. However, similar to the speed-ups observed for execution, energy savings are not guaranteed for all applications—they are instead a consequence of C-RAM's state partitioning approach, which tries to mask the associated network overhead by distributing execution accordingly.
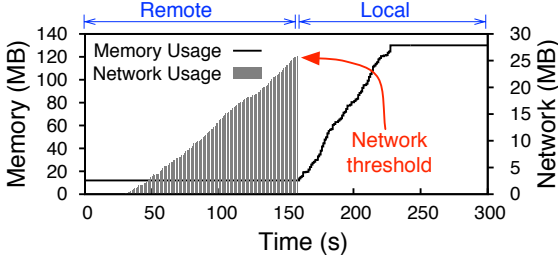
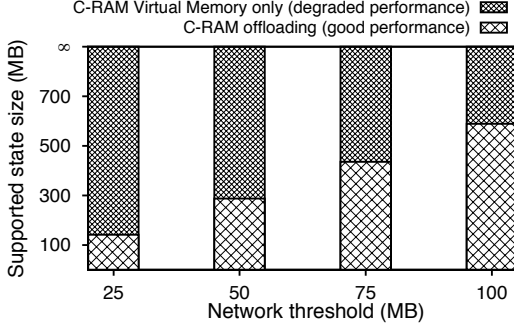Fig. 11: Network usage throttling for iSpreadsheet



Fig. 12: Effect of network threshold on supported state sizes

## 5.5 Network Usage

We also measured the network usage during the above experiment. For the Latrunculi game, most of the remote state is transient, therefore resulting in small snapshots (43 KB). Similarly, for CBIR, a snapshot of individual image objects is small because it does not contain the image itself but rather its unique path identifier (the image database is part of the application's bundle, which is present on both the local and remote nodes), totaling 32 KB of network traffic. For the spreadsheet application, snapshots are larger (21 MB and 29 MB for the simple and complex workflows, respectively) but still remain significantly smaller than the corresponding object representation of spreadsheets.

Figure 11 shows the impact of the network usage throttling mechanism (see §4.6). We set the network threshold to 25 MB and plot the device memory and network usage for the complex spreadsheet workflow over 3G. Initially, memory usage remains low because the bulk of the application state resides remotely. Network usage then increases due to the transmission of large state snapshots, but once the specified threshold is exceeded, C-RAM reverts to local execution to avoid further transfers. As a result, memory consumption increases due to the remote state that is instantiated on the device during local execution.

Based on the same set-up, Figure 12 shows how the network usage threshold affects the application. For different thresholds, we show how much application state C-RAM can offload to the remote node without having to revert to local execution (using C-RAM's virtual memory scheme). By increasing the threshold, more application state can be offloaded, thus augmenting device memory without trading off performance.
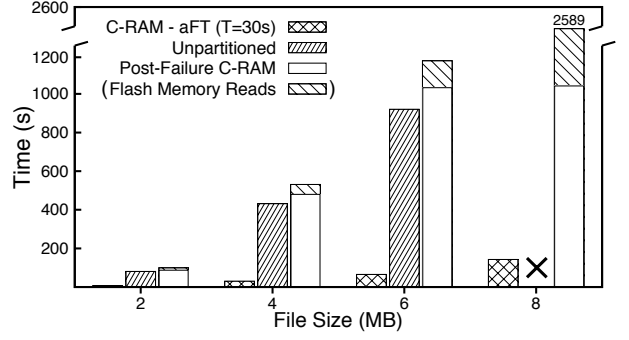


Fig. 13: Post-failure overhead for iSpreadsheet

## 5.6 Post-Failure Overhead

Next we explore the overhead of C-RAM's user-level virtual memory scheme, which loads objects on-demand after failure recovery (see §4.5). We compare the response time of the iSpreadsheet application when sorting spreadsheets of different sizes with (i) an unpartitioned application (local execution); (ii) remote execution using C-RAM; and (iii) post-failure execution using C-RAM, i.e. local execution with on-demand loading of remote objects from snapshots stored in flash memory. The values for post-failure execution include recovery times, which are on the order of milliseconds. We also indicate the time required to load objects from flash memory.

Figure 13 shows that, for all state sizes, C-RAM achieves a speed-up of approximately $14\times$ compared to unpartitioned execution. For state sizes above 8 MB, the unpartitioned execution exhausts memory, while C-RAM resumes local execution after a failure using its virtual memory scheme. After failure, local execution exhibits approximately a 25% reduction in performance—15% is caused by the object loading from flash memory, and the rest is due to the additional checks if objects are present in memory.

## 6 RELATED WORK

Next we discuss previous work on cloud-assisted execution, application partitioning, and checkpointing and logging.

**Cloud-Assisted Execution**. Balan et al. [39] first introduced *cyber foraging*, i.e. the opportunistic use of remote resources to augment smartphone capabilities. With the rise of cloud computing, this idea generated interest in the systems community, leading to proposals such as *Cloudlets* [40] and *Virtual Smartphone over IP* [26]. Both approaches treat smartphones as thin clients served by virtual device images on remote servers. In contrast to C-RAM, this requires a high-bandwidth network, and execution cannot continue in the absence of network connectivity.

Over time, cyber foraging took the form of more rigorous offloading systems such as *MAUI* [41], *Thinkair* [42], *CloneCloud* [43], *POMAC* [44] and *EdgeReduce* [45]. These approaches apply a more fine-grained partitioning with optimisation goals such as reducing application response times, energy consumption, or the network traffic between mobile applications and backend services. Applications are partitioned by either converting local method

calls into remote calls, migrating entire VM images from the device to a remote server, or intercepting method invocations at the VM instruction level, which are then redirected to a remote server for execution. The state of applications is typically transferred back and forth with every offloaded function call. Alternatively, approaches such as *COMET* [46] and *UpShift* [47] use distributed shared memory (DSM) to maintain replicas of the application state across nodes and propagate only changes when offloading. This requires support for DSM by the runtime system and incurs the intrinsic cost of DSM.

Other offloading approaches have focused more on reducing the overheads associated with cloud-assisted execution. *Cloudlet + Clone* [48] and *LOCO* [49] reduce communication delays due to offloading by either introducing a nearby middle layer between devices and distant cloud-based nodes or leveraging available devices in the vicinity of mobile users to serve as remote nodes. *COSMOS* [50] further improves performance by scheduling offloading requests intelligently to reduce contention for cloud resources. It also makes offloading decisions that account for the high variability in network conditions; similarly *Smartphone Energizer* [51] makes more informed offloading decisions based on a wider set of contextual information. Berg et al. [52] propose capturing partial results of offloaded code to handle cases where offloaded execution is interrupted by network failures.

While the above systems focus on compute, energy or network resources, C-RAM's goal is to extend memory. Therefore existing offloading approaches do not partition application state permanently between the local and remote nodes, but instead always keep the state locally. For each offloaded function call, application state is transferred to the remote node for computation and then migrated back. This restricts application memory usage to the available device memory. In contrast, C-RAM aims to facilitate memory-intensive applications via state partitioning, which allows applications to maintain a large amount of state remotely.

**Application Partitioning.** Application partitioning has also been investigated outside the context of cloud-assisted execution. *J-Orchestra* [53] automatically partitions Java applications using byte code rewriting. Similar to C-RAM, it exploits proxy objects but it is simpler due to Java's higher-level, type-safe object model, especially with regard to memory management. It also only offers a partitioning mechanism without policies and does not handle failures.

*Coign* [54] partitions binary applications built from COM components. It applies a graph-cutting algorithm to the component communication graph of an application. In contrast, C-RAM operates at the source-code level and does not assume an already modularised application architecture.

*Swift* [55] splits web applications in a secure fashion using information flow analysis, with the goal of improving UI response times. Developers express security policies through declarative annotations to guide the partitioning process, while static analysis is used to identify data flow between methods. A traditional max-flow min-cut algorithm outputs the placement of code and data. *Fabric* [56] is a system for building secure distributed information systems. It partitions applications across a set of heterogeneous storage, worker and dissemination nodes. Both Swift and Fabric assume high-level languages with annotations for application development whereas C-RAM targets existing applications written in a low-level C dialect.

*Wishbone* [57] partitions dataflow applications between sensors and servers to minimise network and CPU usage for high-rate data processing applications. It estimates resource requirements at compile time by profiling functions against sample data. In contrast to C-RAM, Wishbone assumes a sensing application structured as a data processing pipeline.

**Checkpointing and Logging.** Rollback-recovery techniques have been used for failure recovery and debugging. Toolkits like *DMTCP* [58] and *CLIP* [59] allow transparent user-level checkpointing of distributed applications, relying on single process checkpointers, namely *MTCP* [60] and *libckpt* [61]. Both copy entire memory regions and per-thread metadata to disk, which are used to roll back execution by restoring memory contents. libckpt supports additional optimisations such as incremental and copy-on-write checkpointing using page protection hardware to capture only the updated state since the last checkpoint.

While coordinated checkpointing of multiple processes requires suspending execution, the task of C-RAM is easier: with only two nodes, consistent snapshots are taken when control is transferred via remote calls. Merging local and remote snapshots, however, requires semantic knowledge, making techniques that copy entire heap regions or memory pages not applicable. Since access to page protection mechanisms is unavailable under iOS, C-RAM uses program analysis to detect updates to object state.

*Rx* [62] is a rollback-recovery technique used for recovering from software bugs. Based on checkpoints, shadow processes are forked and immediately suspended to be used as replacements after failure. Since the fork system call is not available under iOS, this approach cannot be used.

*Log-based techniques* [33], [63], [64] model application execution as a sequence of intervals starting with a non-deterministic event (e.g. user input or state changes based on a random number generator). They log all non-deterministic events to stable storage and maintain consistent checkpoints. As part of recovery, they replay events in the original order from the most recent checkpoint to recreate the application state.

The benefit of log-based techniques is that application state is reconstructed entirely after failure. This assumes, however, that all non-deterministic events can be identified and logged, which requires an understanding of the application logic, thus contradicting C-RAM's goal of automated application state partitioning. For example, in the Latrunculi game (see §5.1), the game AI's next move is chosen at random from a set of best moves. A log-based approach would have to log this random choice, requiring knowledge of the internals of the application.

# 7 CONCLUSIONS

C-RAM is a system that automatically partitions the state of Objective-C applications to allow them to execute on memory-constrained devices by leveraging the memory of remote nodes. Remote state may become inaccessible after network failure, and C-RAM overcomes this problem through a new snapshot-based fault tolerance mechanism. State changes are periodically backed up to the device, which can thus recover lost state after a failure. To support application states that are larger than the device memory after network failure, C-RAM loads objects from snapshots on-demand with the help of a virtual memory scheme implemented at user level.

C-RAM is implemented under iOS without platform modifications or changes to the Objective-C language. We showed experimentally that C-RAM supports the execution of application workloads that would otherwise exhaust local memory. It also takes advantage of the remote node's faster CPU to execute compute-intensive application code when possible, thus controlling the impact of partitioned execution on application performance and energy consumption.

# REFERENCES

[1] B. Diniz, D. Guedes, W. Meira, Jr., and R. Bianchini, "Limiting the Power Consumption of Main Memory," in *ISCA*, 2007.
[2] Mashable, *Wearable Devices*, 2014, http://goo.gl/RmqXq4.
[3] Young, Elisabeth, *90 Million Wearable Devices Expected to Ship in 2014*, 2014, http://goo.gl/PPXJvU.
[4] Apple, *Shredder Chess*, 2009, http://goo.gl/bHdB9y.
[5] ——, *Reversi*, 2008, http://goo.gl/ZoZHwp.
[6] ——, *Lumify Video Editor*, 2013, http://goo.gl/xfxDaz.
[7] ——, *Magisto - Magical Video Editor*, 2013, http://goo.gl/SRA44o.
[8] Google, *Google Glass*, http://www.google.com/glass/.
[9] D. Crawford, *Why mobile apps are slow*, 2013, http://goo.gl/7fK168.
[10] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," in *TOCS*, 1984.
[11] Samsung, *Mobile Memory Package*, 2014, http://goo.gl/iqYKou.
[12] iFixit, *Apple A6 Teardown*, 2014, http://goo.gl/XObLF7.
[13] Greenberg, Marc and Allan, Graham, *LPDDR4 DRAM Meets Mobile Power and Performance Demands*, 2014, http://goo.gl/Ac3itt.
[14] Levin, Jonathan, *Handling low memory conditions in iOS and Mavericks*, 2013, http://goo.gl/i4dSPV.
[15] Ars Technica, *The Apple Ecosystem*, 2013, http://goo.gl/m05hrn.
[16] Speirs, Fraser, *Misconceptions About iOS Multitasking*, 2012, http://goo.gl/7CGu2J.
[17] Apple, *Advanced Memory Management Programming Guide*, 2014, http://goo.gl/Y5JeSf.
[18] ——, *Optimizing Memory Performance*, 2014, http://goo.gl/aKejZT.
[19] ——, *Memory Usage Perf. Guidlines*, 2014, http://goo.gl/B7XSOg.
[20] Latrunculi, *Latrunculi*, 2006, mactrunculi.sf.net.
[21] Stricker, Sandro, *Latrunculi LE*, 2009, http://goo.gl/XyF3ZF.
[22] Mac Performance Guide, *Monitoring How Much Memory Is Used*, 2009, http://goo.gl/qhJn0g.
[23] Joemmac, *Photoshop Touch*, 2012, http://goo.gl/53KuTf.
[24] Mallin, Noah, *Why Are Facebook, LinkedIn and Foursquare Splitting Their Apps?*, 2014, http://goo.gl/x7Xzyg.
[25] TechTimes, *Facebook users on iOS*, 2014, http://goo.gl/aYwT2s.
[26] E. Y. Chen and M. Itoh, "Virtual Smartphone over IP," in *WOW-MOM*, 2010.
[27] AnywhereTS, *Why Thin Clients*, 2014, http://goo.gl/2YEhYd.
[28] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *USENIX ATC*, 2004.
[29] ZeroC, *Internet Comm. Engine (ICE)*, 2005, zeroc.com.
[30] CocoaDev, *MethodSwizzling*, 2013, http://goo.gl/lQmuYZ.
[31] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning Low Overhead Replacement Cache," in *FAST*, 2003.
[32] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in *CASES*, 2006.
[33] J. F. Bartlett, "A NonStop Kernel," in *SOSP*, 1981.
[34] Picalike, *Picalike Visual Technologies*, 2014, http://goo.gl/Y5MM8f.
[35] Empora, *Empora Group*, 2014, http://www.empora.com.
[36] Ebay, *More Like This*, 2014, http://www.ebay.co.uk/mlt.
[37] PASCAL, *The PASCAL Object Recignition Database Collection*, 2014, http://goo.gl/6eiQcS.
[38] Apple, *Energy Usage Instrument*, 2014, http://goo.gl/7Sv4Dp.
[39] R. Balan, J. Flinn, M. Satyanarayanan, H.-I. Yang, and S. Sinnamo-hideen, "The Case for Cyber Foraging," in *SIGOPS*, 2002.
[40] M. Satyanarayanan, P. Bahl *et al.*, "The Case for VM-based Cloudlets in Mobile Computing," in *IEEE PerCom*, 2009.
[41] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer With Code Offload," in *MobiSys*, 2010.
[42] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," in *INFOCOM*, 2012.
[43] B. Chun, S. Ihm *et al.*, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *EuroSys*, 2011.
[44] M. A. Hassan, K. Bhattarai *et al.*, "POMAC: Properly Offloading Mobile Applications to Clouds," in *HotCloud*, 2014.
[45] A. Pamboris and P. Pietzuch, "EdgeReduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies," in *MobileSoft*, 2015.
[46] M. S. Gordon, D. A. Jamshidi *et al.*, "COMET: Code Offload by Migrating Execution Transparently," in *OSDI*, 2012.
[47] C.-K. Lin and H. T. Kung, "Mobile App Acceleration via Fine-Grain Offloading to the Cloud," in *HotCloud*, 2014.
[48] C. M. S. Magurawalage, K. Yang, L. Hu, and J. Zhang, "Energy-Efficient and Network-Aware Offloading Algorithm for Mobile Cloud Computing," in *Computer Networks*, 2014.
[49] A. Ferrari, D. Puccinelli, and S. Giordano, "Code Offloading on Opportunistic Computing," in *PerCom Workshops*, 2014.
[50] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS: Computation Offloading As a Service for Mobile Devices," in *MobiHoc*, 2014.
[51] A. Khairy, H. H. Ammar, and R. Bahgat, "Smartphone Energizer: Extending Smartphone's Battery Life with Smart Offloading," in *IWCMC*, 2013.
[52] F. Berg, F. Dürr, and K. Rothermel, "Optimal Predictive Code Offloading," in *MOBIQUITOUS*, 2014.
[53] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Enhancing Java Programs With Distribution Capabilities," in *TOSEM*, 2009.
[54] G. C. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *OSDI*, 1999.
[55] S. Chong, J. Liu *et al.*, "Secure Web Applications via Automatic Partitioning," in *SIGOPS*, 2007.
[56] J. Liu, M. D. George *et al.*, "Fabric: A Platform for Secure Distributed Computation and Storage," in *SOSP*, 2009.
[57] R. Newton, S. Toledo *et al.*, "Wishbone: Profile-based Partitioning for Sensornet Applications," in *NSDI*, 2009.
[58] J. Ansel, M. Rieker *et al.*, "User-level Socket-based Checkpointing for Distributed and Parallel Computation," in *CoRR*, 2007.
[59] Y. Chen, J. S. Plank, and K. Li, "CLIP: A Checkpointing Tool for Message-passing Parallel Programs," in *Supercomputing Conf.*, 1997.
[60] M. Rieker, J. Ansel, and G. Cooperman, "Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux," 2006.
[61] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *USENIX ATC*, 1995.
[62] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating Bugs as Allergies," in *SIGOPS*, 2005.
[63] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault Tolerance Under UNIX," in *TOCS*, 1989.
[64] R. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," in *TOCS*, 1985.

**Andreas Pamboris** received his Ph.D. degree in computer science from Imperial College London. He is currently a post-doctoral researcher in the Department of Computing at Imperial College London, conducting research on mobile cloud and edge computing.

**Peter Pietzuch** received his Ph.D. degree in computer science from the University of Cambridge. He is currently a Reader (Associate Professor) in the Department of Computing at Imperial College London, leading the Large-scale Distributed Systems (LSDS) group that does research on scalable software systems of any kind.