Storing a Persistent Transactional Object Heap on Flash Memory

Michal Spivak Sivan Toledo
School of Computer Science, Tel-Aviv University
{mspivak,stoledo}@tau.ac.il

Abstract

We present the design and implementation of TINYSTORE, a persistent, transactional, garbage-collected memory-management system, designed to be called from the Java virtual machine of a Java Card. The system is designed for flash-based implementations of Java Card, a variant of the Java platform for smart cards. In the Java Card platform, objects are persistent by default. The platform supports transactions: a sequence of accesses to objects can be explicitly declared to constitute a transaction. TINYSTORE supports explicit transactions and atomically executes individual accesses that are not part of transactions; it also supports garbage collection, even on systems with a small constant amount of RAM. TINYSTORE uses a novel approach and specialized data structures to efficiently manage flash memory. We demonstrate its effectiveness by comparing it to a traditional EEPROM-based memory management system for Java Cards.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Smartcards; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; D.4.2 [*Operating Systems*]: Storage Management—Allocation/deallocation strategies, Garbage collection, Secondary storage; H.2.4 [*Database Management*]: Systems—Transaction processing

General Terms Algorithms, Design, Measurement

Keywords persistent heaps, persistent object stores, flash, NOR flash, transactions, Java Card, smart cards

1. Introduction

We present a novel approach to storing a Java heap on flash memory. The Java Card standard defines a Java platform for smart cards. It is a mainstream platform for tiny embedded computer systems. Between 1997 and 2005, more than 600 million Java Cards have been issued [14]. In the Java Card platform, objects are by default persistent. Furthermore, mutations are atomic and explicit transactions are supported. Today, Java-Card objects are stored on EEPROM (electrically-erasable programmable memory); in the near future, they are likely to be stored on flash memories. In this paper, we present a heap sub-system that allows Java Cards to use flash memory in a novel way. To the best of our knowledge, it is the first such system: existing Java Cards either use EEPROM, which does not require a sophisticated heap system, or they emulate EEPROM on flash in a relatively inefficient way.

We believe that the applicability of our system extends beyond Java Cards. Our system, along with the persistent and atomic memory model of the Java Card platform, should be useful in many other small embedded systems that require persistent storage.

EEPROM, which until recently has been used on all smart cards, is a persistent random-access device that allows single bytes to be read and written, so data structures that were originally designed for RAM can be used. To achieve atomicity of mutation operations and of transactions, part of the EEPROM is reserved for a commit buffer, which serves as a redo or undo log [3, 16, 20, 21].

The smart-card industry is currently exploring the possibility of moving from traditional EEPROM to flash memories. Flash is a type of EEPROM in which entire blocks, called *erase units*, are erased together. An erasure clears all the bits in the block. These bits can subsequently be set (sometimes in bytes and sometimes individually); set bits can only be cleared again by erasing the whole block again. Bulk erasures reduce the per-byte cost of erasures and lead to denser memories [19]. The specific type of flash devices that are used in smart-cards and other small embedded systems is called NOR flash, which is memory mapped. Some NOR devices allow individual bits to be set; this feature is called reprogramming.

Because the hardware characteristics of flash, especially NOR flash, are different than the characteristics of RAM and from the characteristics of magnetic disks, flash-specific data structures have been developed over the last 15 years or so [5]. These data structures are designed to minimize the number of erasures, which are slow and wear out the device (each EEPROM/flash memory cell can sustain a limited number of erasures, usually between 10,000 and 1,000,000); to exploit fast reads and, when available, the ability to program single bits; and to even out the wear over memory cells, in order to avoid early wear of a particular cell.

In EEPROM-based Java Cards persistent objects are stored on EEPROM using heaps that are similar to RAM-based heaps. In such implementations, e.g. [10], an object is stored in a fixed memory address and is mutated in place. Only the (optional) garbage collector can move objects in memory. This approach is suitable for EEPROM, in which single bytes can be erased and rewritten.

We propose a radically different approach, designed for (future) Java Cards that use flash rather than EEPROM memory. In our approach objects are not mutated in place. Instead, the new values of fields are written to new memory locations, along with mapping data that allows the system to find the most recent value for every field of every object. The design of our heap is closer to the designs of flash file systems and flash removable memory cards than to the design of a RAM-based heap.

TINYSTORE is implemented in C and is designed to be called by the Java virtual machine (JVM) of the Java Card. It provides the following features: (1) Memory management services: functions to create persistent objects and arrays, and functions that allow the JVM to explicitly delete objects and arrays if a garbage collection is not used. (2) Memory access services: functions to read or write fields of objects or arrays. (3) Access-optimization services: TINYSTORE allows the JVM to choose a read-optimized or a modify-optimized representation for objects, on a per-object basis. (4) Transaction support: support for arbitrary transactions and for atomic individual modify operations. (5) Garbage-collection services: TINYSTORE provides the JVM with services that allow the JVM to implement a mark-sweep garbage collector on the persistent objects and arrays with only a small constant amount of RAM. (6) Array utility function (copy, fill, compare), both an atomic and non-atomic, as required by the Java Card standard. (7) Wear leveling: the system includes a mechanism to ensure that the wear (number of erasures) of all the erase units is roughly uniform.

The rest of this paper is organized as follows. Section 2 describes the Java-Card memory model. TINYSTORE's requirements and design goals are described in Section 3. Section 5 describes the design of TINYSTORE. Section 6 describes how we evaluated TINYSTORE and the results of this experimental evaluation. Section 7 presents our conclusions from this research.

2. Java Card Memory Model

The persistence model of the Java Card platform is different from that of Java variants designed for more capable computer systems. In most Java platforms (J2ME, J2SE, J2EE), objects are stored in volatile memory and persistence is achieved through the use of files and/or databases. Attempts to create versions of Java and similar languages in which the objects themselves are persistent [1, 2, 7, 8, 11, 12, 13, 15] have not been successful enough to become widely adopted. But in Java Cards, the objects themselves are persistent [3, 20, 21]. Java Card objects are always persistent, automatic variables (the stack) are always transient, and arrays can be created as either persistent or transient.

As in all the other variants of Java, fields in new objects are automatically initialized to their default value (zero, `null` or `false`). As we shall see, this causes an interesting trade off.

The Java Card platform defines three atomicity models for persistent data. By default, accesses to data items (fields and array elements) are atomic and accesses must be serialized (at least logically). The second atomicity model is based on explicit transactions. This mechanism allows the programmer to group together a set of operations that are executed atomically; either all of the operations complete successfully or they are all rolled back to the state that existed before the transaction began. Transactions can abort due to an explicit API call, due to lack of resources, or due to a system failure (e.g., loss of power). During a transaction, the fields appear to be updated. These updates, however, are not committed until the program explicitly ends the transaction. Any objects that are created during an aborted transaction are deleted and their memory is reclaimed. The Java Card platform does not support nested or concurrent transactions.

A third atomicity model is only used for a filling or copying a section of an array, either atomically (`arrayCopy`) or non-atomically (`arrayCopyNonAtomic` and `arrayFillNonAtomic`). The non-atomic operations provide an optimization opportunity to the heap/JVM implementor.

The Java Card standard allows implementations to employ an automatic garbage collector, but the garbage collector is not mandatory. Most current Java Cards do not include a garbage collector. Java Cards without a garbage collector can reclaim memory using a specialized explicit coarse-grained object deletion mechanism. For further information on the Java Card platform, see [3, 20, 21]. Some have proposed to automatically collect only garbage in RAM, but not garbage in the persistent store [17].

3. Design Goals and API

We designed TINYSTORE to meet the requirements of Java Card implementations that use NOR flash as their persistent storage.

The most important hardware characteristics of smart cards in general and Java Cards in particular are a small RAM and slow clock speeds. For example, top-of-the line smart-card chips announced by STMicroelectronics and by Sharp both have 1 MB of flash memory but only 32 and 8 KB of RAM, respectively.

Our assumption regarding the flash memory are as follows. We assume that the Java Card employs a flash device that is (1) memory mapped for reading, (2) requires special device-driver routines for programming and erasing, (3) is partitioned into fairly large erase units, and (4) allows reprogramming. These assumptions are reasonable for Java Cards.

TINYSTORE supplies an API to the rest of the Java virtual machine. The most fundamental operations in the API are operations that allow the JVM to create and reference persistent objects or arrays. The functions that allocate a new object or array (`CreateNewObject, CreateNewArray`), return a reference (`REF`), a global unique identifier (GUID) for the object or array. The JVM passes to TINYSTORE the number of fields (or the length of the array), the total size of the object/array, and a header. The header changes rarely, and TINYSTORE does not parse it. The access functions (`GetField, PutField, ArrayLoad, ArrayStore`) read or write a single field or element, given a reference, into or from a RAM buffer. The API also supports several global operations on byte arrays. The choice of array operations mirrors the static methods on arrays in the Java Card standard. They are `arrayCopy, arrayCopyNonAtomic, arrayFillNonAtomic` and `arrayCompare.`

When an object or an array is read frequently, accesses through `GetField` or `ArrayLoad` are inefficient. Accessing the object or array through a physical pointer is more efficient (and possible since NOR flash is memory mapped). TINYSTORE provides a function that returns the physical address of a field or array element. The function returns the number of bytes of the object or array that are contiguous in memory (to allow breaking large arrays into sections). TINYSTORE also receives an argument stating whether the JVM needs to be notified if the object or the array move to another physical address. If the JVM intends to use the physical pointer only until the next call to TINYSTORE, it needs no notification. But if the JVM plans to use the pointer beyond the next call to TINYSTORE, then it must request notification. TINYSTORE also provides functions for reading and writing the JVM header. Because JVM headers are not parsed by TINYSTORE, it returns their physical address when the JVM needs to read them. Therefore, the function that returns the address of a header also needs to know whether notification is needed or not. When TINYSTORE needs to notify the JVM that the physical address of certain objects or arrays are no longer valid, it calls a call-back function.

The JVM can instruct TINYSTORE to explicitly deallocate an array or an object. This mechanism is intended for use on Java Cards without a garbage collector.

The API also supports automatic mark-sweep garbage collection for more sophisticated cards. The garbage-collection support in the API is designed to ensure that the JVM can collect unreachable objects while using a small constant amount of RAM and while using flash memory efficiently. The TINYSTORE API supports an explicit mark function; the mark flag is stored on flash, but separately from the object. To allow the collector to remember the identities of objects that have already been visited, the API supports an on-flash stack data structure.

Finally, the API supports explicit transactions.

4. Low-Level Data Structures

TINYSTORE uses three low-level data structures that we developed for an earlier NOR-flash storage system, a file system called TFFS [6]. This section describes these data structures, which are not new.

4.1 Logical Pointers and the Structure of Erase Units

Flash memory is divided into contiguous *erase units* (sometimes called erase blocks). An erasure operation is applied to an entire erase unit and sets all the bits in the unit to one. TINYSTORE assumes that all the erase units have the same size. If some units are physically smaller, they can be either aggregated by the device driver into standard-size units or left unused by TINYSTORE.

TINYSTORE's coarsest memory-management layer manages entire erase units using fairly standard flash-management techniques [5]. At least one unit is always kept free, in the erased state. When the other units fill up, one of them is chosen as a victim for reclamation. The valid contents of the victim are copied into this free unit. The victim unit may also contain data that is already obsolete, such as old values of fields and objects. These data are, of course, not copied. Once all the valid contents are copied, the victim unit is erased and becomes the new free unit. This process is called reclamation.

Usually, the system selects a unit with a significant amount of obsolete data for reclamation. However, to achieve wear leveling, occasionally a random victim is chosen.

TINYSTORE assigns each erase unit to store one kind of data. All the units in use begin with a header. The header describes the kind of data that the unit currently stores, the number of erasures that is has already sustained, an error-detection code, and possibly additional information required for the kind of data it stores.

Most of the erase units are used to store variable size memory chunks that we call *sectors*. Sectors store objects, sections of arrays, headers of objects and arrays, and search-tree nodes. These *regular* erase units are identified by a logical number stored in their header. When a regular unit is reclaimed, the unit that the data is copied into assumes the logical number of the reclaimed unit. A regular unit is divided into sector descriptors and sectors. Each sector descriptor points to a sector within the erase unit. Sector descriptors have a fixed size, and are stored in a variable-size array that starts at the lowest address of the unit and grows upward. Sectors are allocated contiguously starting at the high end of the unit, growing downward. Sector descriptors and sectors never collide.

Pointers within TINYSTORE data structures do not point to physical addresses; they are *logical pointers*. A logical pointer consists of a logical erase unit number and a sector number. To find the physical address of the sector pointed to, TINYSTORE finds the logical unit, finds the sector descriptor with the given number, and uses it to locate the sector it points to. When a unit is reclaimed, memory is allocated in the new unit such that descriptors are copied to the same position in the descriptor array, but the sectors themselves are compacted. For more details on these mechanisms, see [6].

TINYSTORE uses erase units for two other kinds of data, a log (journal) and for garbage-collection support. These data structures are described in Sections 5.5 and 4.3.

4.2 Efficient Pruned Versioned Search Trees

TINYSTORE uses search trees called *pruned versioned trees* [6] to find objects, array sections, and other data items. These trees were developed specifically to support efficient transactional search-tree operations on NOR flash memories.

Pruned versioned trees essentially keep one or two versions of a search tree (a dictionary data structure with ordered keys) in one data structure, which formally is a directed acyclic graph with one or two roots. One version is always a read-only search tree, which only supports look-up operations. The other version, if it exists, is a read-write version. In the read-write version, new key-value pairs can be added, the values of existing pairs can be replaced, and pairs can be deleted. The read-write version can be committed or aborted at any time. If the read-write version is committed, it becomes read-only and the old read-only version disappears. If the read-write version is aborted, it simply disappears and can no longer be accessed.

Look-ups, creations and deletions of pairs, and value changes all cost $O(\log(n+k))$ operations, where n is the size of the read-only dictionary and k is the number of modifications made to the read-write version. Committing and aborting costs $O(k)$ operations. The total size of the data structure is $O(n+k)$ (plus of course the size of the key-value pairs themselves).

4.3 The Structure of the Log

TINYSTORE uses a log (a journal) to perform certain operations atomically. These operations include sector allocations, JavaCard transactions, creation and deletion of objects, and erase-unit reclamations. (Field modifications that are not part of a transaction are performed atomically but without using the log.) The log is also used to find key data structures at boot time.

The log is represented by an array of fixed size records called *entries*. This array fills an entire erase unit (except for the erase-unit header). This erase unit is marked as such in its header. It does not have a logical number, because it does not store sectors and because there are no logical pointers into it. The log contains entries that point to allocated sectors. These entries allow the system to mark the data structure stored in a sector as comitted or aborted. TINYSTORE uses many types of entries, each of which corresponds to a particular data type that it uses. There are also special record types that point to key data structures (the REF-tree and the NOTIFY-tree, which are described below) and record types that record the progress of erase-unit reclamations.

The overall structure of the log is identical to that of TFFS, and some of the record types are used in both systems, but TINYSTORE also uses some specialized record types.

5. The Design of the Java Card Memory Management System

TINYSTORE introduces design innovations and features that are not present in TFFS and other prior work. In particular, TINYSTORE supports physical addressing of objects and arrays, object-optimized field modifications and array-optimized element modifications, global operations on arrays, and garbage collection. This section describes these mechanisms.

5.1 Mapping References to Flash Addresses

The JVM names objects and arrays in TINYSTORE's API calls using references. The references are unique identifiers that TINYSTORE generates for each object and array when they are created.

We use a versioned search tree, which we call the REF tree, to map references to the flash addresses of the objects/arrays. The representations of objects and arrays are not trivial (they are not stored in contiguous compacted blocks of memory), but each representation is accessed starting at some single address. Each leaf in the tree maps one reference. We distinguish between three kinds of objects and arrays: *objects*, *small arrays*, and *split arrays*. The leaf contains the reference itself (this is used in tree operations), its kind, a logical pointer to the representation of the object/array, and some additional data items that vary among kinds. These data items are described below, together with the description of the object/array representations.

5.2 On-Flash Representation of Objects

The representation of an object in TINYSTORE consists of three parts: an object descriptor, a JVM header, and a linked list of object records, as illustrated in Figure 1. The JVM header is not parsed or used in any way by TINYSTORE. It is used by the JVM to store information about the object, primarily its type. The object descriptor is a record containing the object reference (its GUID/REF), pointers to the JVM header and to the first element of the object-record list, the number of fields in the object and their total size, and an optional array of boolean flags, one per field. Pointers in the descriptor are, of course, logical. The boolean flags, whose role is explained later, are called *initialization flags*.

Each object record consists of three parts: a pointer to the next record in the list, an array of boolean flags called *valid flags*, one per field, and a data segment containing space for each field. Lists always contain at least one record. TINYSTORE does not allow lists to grow beyond a certain length, which is a compile-time parameter (we usually set it at 2).

The object descriptor is contiguous, the JVM header is contiguous, and each object record is contiguous. But these different parts of the representation need not be stored contiguously.

This data structure represents a given state of an object as follows. Consider the ith field of an object. If the object descriptor contains initialization flags and the ith flag is in the erased state '1', then the value of the field is the initial Java value (zero, `false` or `null`). If there are no initialization flags or the ith initialization flag is '0', then the value of the field is the value stored in the first object record whose ith valid flag is in the erased state. The representation invariant guarantees that such a record exists if there are no initialization flags or the ith initialization flag is '0'. The current values of the fields are not necessarily stored in a single record. A record may contain the current values of some fields, old values for other fields, and empty space for a third group of fields.

Now consider the modification of a field, say the ith field in an object. To keep the presentation simple, we ignore the issue of atomicity now; we shall discuss atomic field modifications later. If the descriptor contains initialization flags and the ith flag is in the erased state, it is cleared and the new value is written to the space reserved for the field in the first object record in the list. This space is guaranteed to be in the erased state. Otherwise, we traverse the list to find the first record in which the ith valid flag is in the erased state. This record contains the current (pre-modification) value of the field. If it is not the last record in the list, we clear its ith valid flag and write the new value to the next record. If the record containing the current value is the last in the list, but the list can still grow (it is below its maximal length), we allocate a new object record and store a pointer to it in the last record in the existing list. Now we can clear the valid flag of the current value and write the new value to the newly-allocated record.

If the list does not have room for a new value for the field and it has reached the maximal length, we must compact it. To compact the list, we allocate a new object record and traverse the old list, copying the current value of every field except i to the newly-allocated record. We also write the new value of field i to the newly-allocated record. Finally, we allocate a new object descriptor and copy the contents of the old descriptor to it, except for the pointer to the object-record list, which is set to point to the new list.

We sometimes compact an object when the JVM requests its physical address by calling `GetPhysicalPointer`. If, at the time that TINYSTORE processes this request, the current values of the fields are stored in more than one record, then TINYSTORE compacts the objects. Also, if there are initialization flags and some of them are in the erased state, then explicit default values are written to the single record that now stores all the fields' values. Subsequent field modifications are processed normally, which causes the

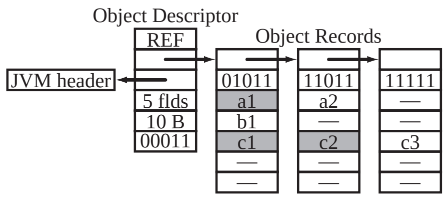

Figure 1. On-flash representation of an object. The example above shows an object with three modified fields. Field slots shown in grey are old (obsolete), and field slots with a dash are empty. The first field (denoted 'a') has been modified twice, the second field modified once, the third twice, and the other two have not been explicitly set at all.

object to consists of multiple records. This causes the invalidation of the physical pointer that was returned to the JVM using a mechanism that we describe later in the paper.

5.3 On-Flash Representation of Arrays

The representation of small arrays in TINYSTORE is similar to the representation of objects, but the representation of large arrays is more sophisticated.

Arrays larger than a compile-time threshold t are treated differently. We set the threshold t to 255 elements, which is also the maximal number of fields in a Java Card object. A contiguous record with enough space for all the elements of a large array would cause two problems. First, if the array is larger than one erase unit (or even smaller than but close to the size of a unit), then TINYSTORE's memory allocator would not be able to allocate the record. Allocating records that span more than one erase unit is a bad idea, because the resulting contiguous erase-unit groups interfere with the wear-leveling mechanism that flash devices need. Second, if one or a few elements are modified more frequently than the rest, then allocating a record with space for a new value for every element is wasteful.

Therefore, we allocate records for large arrays at a smaller granularity. We represent large arrays using an array descriptor, a JVM header, and a pruned versioned tree whose leaves are lists of array records, as shown in Figure 2. The array is partitioned into sections of t elements, except perhaps the last section. Each section is represented by a linked list of records. Each record contains space for t elements, along with t valid flags. A special structure that we call a section descriptor points to the first record of the section's list. The section descriptor also contains the optional initialization flags for the elements in the section. The section descriptors are the leaves of the pruned versioned tree. The search keys in the tree are the sequential section numbers. The pointer to the root of the tree is stored in the array descriptor.

Array Utility Functions

Since arrays are represented in a noncontiguous way, bulk operations such as copying, filling, and comparing arrays can be performed more efficiently by TINYSTORE than by the JVM. Therefore, TINYSTORE provides utility functions that correspond to array methods in the Java Card standard library. The JVM is expected to call these TINYSTORE functions to implement the methods in the standard library.

5.4 Invalidation of Physical Pointers

Two TINYSTORE services return a physical pointer to flash: `GetPhysicalPointer`, which returns a physical pointer to an object or to an element of an array, and `ReadHeader`, which returns a physical pointer to the JVM header of an object or an array. In certain cases TINYSTORE must notify the JVM that a physical pointer

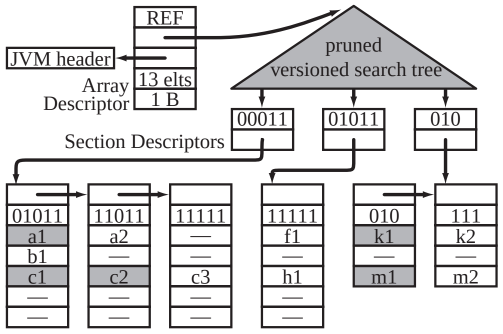

Figure 2. On-Flash representation of a large array. In the example, a section size is 5, the size of a field is 1 byte, and there are 13 elements. Field 1 was updated once, 2 was updated 3 times, 5 and 7 were updated once, 10 and 12 were updated twice. Fields 3,4,6,8,9 and 11 were not updated at all.

is no longer valid. We now explain when and how TINYSTORE invalidates physical pointers.

The physical address of an object or an array section changes when one of the following events occur: (1) A field of an object or an element of the same array section is modified (similarly, JVM headers move when they are modified); (2) the erase unit on which the object or the array section resides is reclaimed. (TINYSTORE does not move objects during garbage collection.)

To keep track of pointers that may require invalidation, TINYSTORE maintains an on-flash data structure called the *Notify Tree* containing all the references (REF's) of objects and arrays for which a physical pointer was returned to the JVM. A reference is stored in this data structure if a pointer to the object has been returned to the JVM, a pointer to the header, or a pointer to one of the sections of an array. The data structure consists of a pruned versioned tree whose leaves are arrays of REF's. We call these arrays *buckets*. The tree is indexed by physical erase unit number.

When the JVM requests a physical address, TINYSTORE adds the REF of the object/array to the bucket of the erase unit in the Notify Tree, avoiding duplicates. When TINYSTORE is about to move part of an object/array, it checks whether the REF of the object/array is listed in the bucket of the physical erase unit of the data to be modified. If so, the REF is deleted from the bucket and the JVM is notified of the invalidated REF. This notification should cause the JVM to evict the REF-to-physical-address mapping from its mapping cache. When TINYSTORE reclaims an erase unit, it finds the bucket that corresponds to it and notifies the JVM that all the REFs in the bucket are invalid. The bucket is then deleted from the Notify Tree.

If there are no outstanding pointers for an erase unit, the Notify Tree does not contain a bucket for it at all. Also, if there are no outstanding pointers at all, then the Notify Tree does not exist at all.

The need to invalidate pointers causes considerable overhead, so TINYSTORE also supports *short-lived pointers* that eliminate this overhead. If the JVM requests a physical mapping but declares (using an API argument) that it will discard it before its next call to TINYSTORE, then TINYSTORE does not store the REF in the Notify Tree. This reduces the overhead associated with the Notify Tree, especially if most of the JVM's requests are for short-lived pointers.

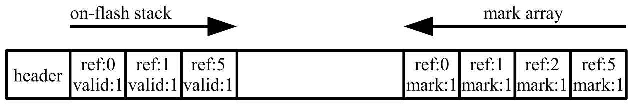

Figure 3. Garbage collection erase-unit

5.5 Garbage Collection

TINYSTORE provides support for garbage collection in RAM-limited systems. More specifically, TINYSTORE supports a stop-the-world mark-sweep collector [9].

Garbage collectors regard the heap as a directed graph whose vertices are objects and arrays and whose edges are object references stored as object fields or array elements. This graph is augmented by another set of vertices called roots, which are not stored on the heap, but which may point to vertices on the heap. The roots represent class fields (`static` fields), automatic variables stored on calls stacks, and possibly variables within the JVM itself. The job of the garbage collector is to deallocate all the objects and arrays that are not reachable from the roots.

The mark-sweep algorithm that TINYSTORE supports traverses the reachable vertices in the graph using a depth-first traversal. It uses two data structures: a stack of references and one mark flag for each object/array on the heap. The stack of references is used to keep track of which objects/arrays have already been visited. The mark flags are used to mark objects/arrays as 'reachable'.

In RAM-based heaps, the stack is stored in RAM, and the mark flags can be stored within object/array headers on the heap. The stack is much smaller than the heap, because it contains only one word (a reference) per object and because its maximal size is bounded by the longest simple path in the graph, not by the size of the graph. Therefore, space for the stack can be reserved without a significant impact on the amount of RAM available for the heap. The mark flags in the object/array headers can be set during the mark phase and cleared during the sweep phase, in preparation for the next garbage-collection cycle. On a RAM-limited system that stores the heap on flash, both solutions are unacceptable. The stack cannot be stored in RAM because there might not be sufficient space in RAM for it. The mark flags cannot be stored in object headers, because a flag in a fixed location on flash cannot be set and cleared over and over again.

Therefore, TINYSTORE provides specialized on-flash implementations for both data structures, as shown in Figure 3. Both data structures are stored on a single erase unit that is dedicated to these structures, called the *GC unit*. The GC unit can be used in several garbage-collection cycles before it is erased. When the remaining space on the unit no longer suffices for a garbage collection cycle, it is erased and another unit is allocated for this purpose. The stacks are stored at the low end of the unit and the mark-flag data structures are stored at the high end.

The abstract stack of REFs is represented by one REF-like variable in RAM, an array of REFs on the GC unit on flash, and two in-RAM indices into this array, called b and n. The REF-like variable in RAM stores the first element pushed onto the stack, which is always a root. Since roots are not stored on the heap, they do not have a REF. Also, not storing pointers to the roots on flash simplifies memory allocation in the GC unit. The RAM index b points to the first element of the on-flash array, which might not be in the lowest address of the unit after the erase-unit header, because of older garbage-collection stacks that might be stored on the unit. The RAM index n points to the first unused location in the on-flash array. To push a REF, we simply store it, along with a *valid* flag, in location n of the on-flash array and increment n. The valid flag remains in the erased state. To pop a REF, we search from $n - 1$

downward for an entry with a valid flag in the erased state. If we find one at *b* or higher, we clear its valid flag and return it. Otherwise, the stack is empty. The linear searches for the top element in the abstract stack may seem inefficient, but they are not actually slow, because reading from NOR flash is much faster than writing to it.

The mark flags are represented by an array containing all the REFs of objects and arrays in the heap in sorted order. We call it the *mark array*. When the garbage collector starts, we traverse the REF tree, which TINYSTORE uses to map REFs to the physical addresses of object/array representations. Since the tree is a search tree indexed by REF, an ordered traversal enumerates the REFs in ascending order. During the traversal, we copy all the REFs that we find to the mark array on the GC unit. In the mark array, we associate a *mark* flag with each REF. During the mark phase of the collector, we clear the mark flag of reachable REFs, which we locate in the mark array using a binary search. During the sweep phase, we traverse the mark array linearly. Each unmarked REF is garbage, so we deallocate the corresponding object/array from the heap.

When a garbage collection cycle begins, we check whether the GC unit contains enough space for two arrays, each of which can store all the REFs plus one flag for each REF. If not, the GC unit is erased and another unit is allocated for the purpose. This ensures that the garbage collector never runs out of space. If there is not enough space even on a just-erased unit, then the garbage collector cannot run; this places a limit on the total number of objects/arrays that TINYSTORE can store.

5.6 Transaction Support and Atomicity

The conventional way to support atomicity and explicit transactions in Java Cards is a commit buffer [3].

A commit buffer can be used in two ways, either as an undo log or as a redo log [16]. In both approaches, values are duplicated in the buffer (the old value in an undo log or the new values in a redo log). A commit buffer is appropriate for EEPROM implementations that overwrite data in place, because during a transaction, both the old and the new data must be retained somewhere.

TINYSTORE, however, never overwrites data in place. When updating fields, it writes the new value to an empty slot in the object-record list. When updating a tree, it uses spare pointers (and possibly a new root) to keep track of both the old and the new trees. Because TINYSTORE always stores updated information in new memory locations, both the old and the new values are always stored persistently on flash even without a commit buffer.

TINYSTORE does use a log (a journal) data structure, but unlike a commit buffer, the log does not contain copies of fields/array elements. The log does contain pointers to the new and/or old copies of objects and tree nodes, so that the system can redo or undo a transaction.

Atomic Operations that are not Part of Transactions

There is an empty slot for the new field in an existing object record. We simply write the new value of the field value to the empty slot, which is in the erased state, and update the preceding record in the linked list to indicate that the value of the field in it is invalid. Since the valid flag of the preceding record is cleared only after the new value is written to the empty slot, an invalid flag always corresponds to an already-written new value. However, if the system crashes after the new value is written but before the preceding valid flag is cleared, the just-written slot is left in a state that is neither used nor empty (erased). To address this possibility, we always check whether empty slots are in the erased state, and if they are not, we fix the problem by compacting the object.

If the empty slot is in the first record of the object (the field has never been set), we use a similar method, but clear the initialization flag of the field rather than a valid flag. If such a modification is interrupted, then when we discover the unused but not erased slot, we explicitly write the default value into it first, which is always possible because all the bits in the default values are zero. Then we clear the initialization flag. Now we retry to write the new value. This is more efficient than compacting the object.

Filling an empty slot in an already-allocated record is the only atomic field modification that is not represented in the log at all. All the other cases require a more complex algorithm, and they are all represented in the log by a sequence of log entries that constitute an implicit nameless transaction.

A new object record must be allocated. If there is no empty slot for the field in an existing record and we have not reached the maximum number of records in the list, then we allocate a new record. The logical address of the new record is written to a log entry before the record is actually initialized. This allows the system to reclaim the memory if a failure occurs now. Next, we write a log entry pointing to the last record in the existing list, and then write a logical pointer to the newly-allocated record into the last record's `next` field. If writing the `next` field is interrupted, we clear all the bits of the `next` field, to indicate that it is simply invalid. A list ending with an all-zeros `next` field cannot be extended, so when we need to extend it, we first compact it.

The object-record list must be compacted. To compact a list, we allocate a new object record, copy the old contents of the object into it (but the new value of the modified field). Next, we allocate and initialize a new object descriptor; it points to the newly-allocated object record. Finally, the REF tree is modified: the new object descriptor replaces the old one. All of these operations are logged as part of an implicit transaction. Now we append log entries that state that the old representation of the object (its descriptor and records) should be deallocated. We do not yet deallocated them, because if this implicit transaction aborts, the object will revert to its old representation. After writing the commit marker to the log, we go over the log entries for the transaction again, deallocating memory and marking spare pointers (and possibly the new root) in the REF tree as valid.

Explicit Java-Card Transactions

A transaction can modify multiple fields in multiple objects. To allow the state of the program to roll back, we create a new representation for each object that is modified by the transaction. That is, we do not write new field values to empty slots in the existing object-record list and we do not clear initialization flags in the object descriptor. Instead, upon the first modification of an object in a transaction, we allocate a new object descriptor and new object records. We copy the old values of the fields into the new representation, except, of course, for the field that is being modified. We then modify the REF tree to point to the new object. The modification is to the read-write version of the versioned tree; if the transaction later aborts, then the tree can revert to the read-only version that points to the old representation of the object.

6. Experimental Results

We evaluate TINYSTORE by comparing it to a traditional overwrite-in-place EEPROM implementation of the same API. The comparison uses simulated running times as well as several device-independent metrics, such as the amount of user data that can be stored in TINYSTORE before it runs out of memory, or the average number of erasures per field modification.

6.1 EEPROM Memory Management Implementation

To quantitatively evaluate TINYSTORE relative to the state of the art, we implemented a conventional EEPROM-based Java-Card memory management system. The API of this system is identical to that of TINYSTORE except for the lack of garbage-collection support. Our EEPROM implementation uses overwrite-in-place whenever possible; to the best of our knowledge, this makes it similar to most other Java Card implementations. In particular, we did our best to make it efficient and not to cripple it in any way. We were not able to find an existing implementation that we could use for evaluating TINYSTORE.

Because the EEPROM implementation overwrites data in place, it does not include any wear-leveling mechanisms. To the best of our knowledge, there is no simple way to combine the simplicity of overwriting in place with an effective wear-leveling policy.

We divide the EEPROM into two sections: the log and the heap. The size of the log (a commit buffer) is a compile-time parameter. When an object is created it is allocated at a fixed physical location. Objects can be modified (in-place) and deleted, but they never move. The implementation uses a log (a commit buffer) to implement transactions, atomic operations, and atomic maintenance operations. The log is stored at the beginning of the EEPROM memory. The log is a cyclic array of variable-size records.

We have not included error detection in this implementation and we compare it to TINYSTORE compiled without error detection.

The EEPROM Memory Allocator

The allocator is responsible for atomically allocating and deallocating memory blocks. The allocator has its own fixed-size statically-allocated log to assure atomicity in its data structures. The reason the allocator's log is separate from the commit log is to assure that there is always enough log space to allocate or free a memory block regardless of the state of the commit log. The allocator requires a maximum of three log entries for its operations. There are three operations supplied by the allocator; allocate, free and `REFToPhysical`. The `REF` in this system is simply the offset of the object within the heap. The allocator maintains a free list of memory blocks, and associates a small header with each allocated block. The system is initialized with a free list containing one element, the entire space reserved for the heap. The pointers of the free list are stored inside the free blocks, and allocation is done using a first-fit policy. The allocator also supports coalescing; when a block that is adjacent to a free block or two is freed, the blocks are coalesced into a larger free block.

When the system boots, the allocator checks its log to see if it was interrupted. If so, it rolls back the interrupted operation.

The EEPROM's Commit Log

The implementation uses a log (a commit buffer) to implement transactions, atomic operations, and atomic maintenance operations. The log is stored at the beginning of the EEPROM memory. The log is a cyclic array of variable-size records. Its size is a compile-time parameter. Each log entry contains a log-entry header with three items: a valid/obsolete flag, an entry-type identifier and a transaction identifier. In some cases, and depending on the type of log entry, additional information is stored after the log-entry header. The implementation uses the following log-entry types:

- New Object, New Header. These record types allow the system to undo an allocation by calling the allocators `free` function. The records are ignored when a transaction is committed. They are used to free the memory if the transaction aborts.
- Delete Object. The system actually deletes an object (calls the allocator's `free` function) only when a transaction is committed. If a transaction is aborted nothing occurs and the object remains allocated in place.
- Commit Marker. Ensures that the transaction is redone at boot time
- Old Value. These record types contain old values of object fields and array elements. If the transaction is committed, nothing occurs, the new value is already written in the correct location. If the transaction is aborted, the log is traversed to find the old values and to restore the objects to their pre-transaction state.
- Old Header Pointer. This record is used for changing the header of an object. When a header is replaced, a new block is allocated and the pointer to the previous header is replaced. We keep a log entry for a possibility of an interrupt during writing of the pointer.

EEPROM Implementation Details

When an object or an array is created, the allocator is used to allocate memory for the object or array. Each object or array starts with a descriptor that contains a pointer to the JVM header, the size of the object and the field size for arrays. Mutating the object or array is done in-place. The bytes that are to be replaced are first erased and then the new value is written. Prior to the erasures, a log entry, containing the old value of the field, is written to the log. This value is restored in case of an abort. Values of objects or arrays are read using direct access. When an object or array is deleted, the allocator's `free` function is called.

The array utility functions are implemented using a sequence of mutations to the array. `ArrayCopy` copies the value from one array to the other after first writing the old value into the log and then erasing the modified bytes in the destination array. `ArrayFillNonAtomic` and `ArrayCopyNonAtomic` copy values from one array to the other without writing the old value into the log.

The `WriteHeader` API function uses the allocator to allocate a block that will contain the JVM header. The pointer to this block is written in the object's or the array's descriptor. The old pointer is first written to the log and then to the object or array descriptor, this is to ensure that an interrupted update of the header will be recoverable.

Code-Size Comparisons

The EEPROM implementation is about 2,400 lines alone (in C), whereas the combined sources of TINYSTORE and TFFS [6] are about 17,000 lines long. TINYSTORE and TFFS share a considerable amount of code (e.g., manipulation of pruned versioned trees, the boot sequence and layout of erase units), so we maintain the two systems using one set of source files. Therefore, it is hard to estimate how large TINYSTORE alone is, but its implementation is at least 4–5 times longer than that of the EEPROM system.

This does not mean that the EEPROM implementation is particularly naive or inefficient. Rather, the difference in code size reflects the complexity of TINYSTORE. This higher complexity is a drawback of TINYSTORE, since the higher complexity incurs higher development and testing costs and a larger binary footprint, which is important for Java Cards.

6.2 Experimental Setup

Our experimental setups consist of three components: the heap implementation under test, an emulated flash/EEPROM device, and a C program that calls the heap. The C program emulates the behavior of a JVM running an applet (or applets). At this point in our research, we have not yet integrated TINYSTORE into a JVM and we have not yet run TINYSTORE on actual flash devices (but we have ported earlier flash-management software to flash devices so we are confident that TINYSTORE can actually run on actual flash hardware).

Device Specifications

The simulations use a NOR flash device that can be used in a Java Card. Smart cards today can have a flash memory of up to 1 MB, but most have less. Our experiments use a flash device with 448 KB of memory, partitioned into erase units of 8 KB. We based our flash and EEPROM timing estimates on industry estimates for smart-card flash/EEPROM devices [19]. For EEPROM, the estimates in [19] are: 100 ns per read, 4 ms for erase+write. For NOR flash, the estimates are: 100 ns per read, erase time is 0.5 seconds, and write time is 30 μs. We used these estimates as is, except for the EEPROM erase and write time, where we used 3.5 ms for the erasure and 30 μs for the subsequent write (the total amounts to 3.53 ms, less than the estimate in [19]).

We assume that both the NOR flash and the EEPROM device can read and write words of 1, 2, or 4 bytes. We also assume that the EEPROM device can erase in 3.5 ms any such word. In the results reported below, we always count the number of reads, writes, and erasures, not the number of bytes that are processed.

TinyStore configuration

The comparisons to the EEPROM implementation use a baseline configuration: 8 KB erase units, at most 2 records in lists, the initialization-flag optimization is enabled, error detection and garbage collection are both disabled.

We disabled garbage collection since it is largely irrelevant for comparing the two implementations. In particular, to the best of our knowledge most existing EEPROM implementations do not support garbage collection.

In some experiments, we varied TINYSTORE's configuration to assess the impact of some configuration parameters. We used error detection in experiments designed to quantify its cost; we varied size of erase units between 2 KB to 64 KB to assess the impact on the user-data capacity of the heaps; we ran TINYSTORE with and without initialization flags to assess their impact.

Workloads

We used five workloads that represent different object-modification usages, but we only show detailed results for two of them. The first workload, which we call *SingleObjectAllFields,* repeatedly updates all the fields of a single object. The second workload, *SingleObjectSingleField,* always updates a single field in a single object. The third workload, *TotallyRandom,* updates a random field within a random object. The fourth workload, *20/80AllFields,* splits the objects into 20% hot objects and 80% cold objects. Cold objects are selected for update with probability 0.1, and hot objects with probability 0.9. When an object is selected, all its fields are modified. The fifth workload, *SequentialObjectsAllFields,* updates all the fields of all the objects sequentially, in the order in which the objects were created.

We present in this paper detailed results for the two workloads that represent reasonable approximations of real-world behaviors, *TotallyRandom* and *20/80AllFields*.

In these workloads, we used two-byte fields (or two-byte array elements) and objects/arrays of varying total size. In experiments designed to measure the performance of array utility functions, elements are single bytes, since the Java Card library only supports global operations on byte arrays. The objects and arrays in the experiments were created without JVM headers.

The same five workloads were run under three different fullness conditions. In the first condition, the devices are only about 20% full. In the second, the devices are about half full, and in the third, the devices are almost full (90%). For each of these levels of fill, we first ran a preliminary experiment in which we filled TINYSTORE with objects until there was no more space left on the device. We recorded the number of objects stored in this completely-full state, and subsequently used 20%, 50%, and 90% of this number.

We ran most of the experiments until one of the erase units (or a byte on EEPROM) was erased 100,000 times. This assumes that the guaranteed endurance of the devices is 100,000 erasures. We have also ran experiments with an endurance limit of 10,000 erasures and obtained very similar results, so we do not think that the results would vary much even with much higher or much lower endurance limits.

We are not aware of any existing benchmarks that we could have used. To the best of our knowledge, there are no high-level benchmarks for Java Cards. One set of micro benchmarks, called SCCB, has been announced [4, 18], but they were not appropriate for the evaluation of TINYSTORE. First, the SCCB benchmarks have not yet been released. Second, these benchmarks are distributed as a set of Java applets, so they can only be used to evaluate a whole JVM, not to evaluate only the heap. Third, SCCB is unlikely to include enough benchmarks for a thorough evaluation of the heap data structures.

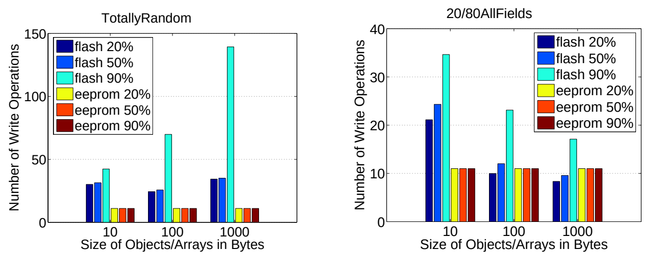

Figure 4. Amortized Number of Write Operations. Default TINYSTORE configuration, two (out of five) workloads, three fullness conditions. Each groups of six bars represent experiments with one specific object/array size (in bytes).

6.3 Results

Performance

The first set of experiments explores the performance of TINYSTORE. We measure the performance by counting the number of write, read and erase operations (only to flash or EEPROM, not to/from RAM). From these numbers and from the flash/EEPROM hardware performance estimates we can estimate the run times of the experiments. Let N_w, N_r and N_e be the total write, read, and erasure counts in an experiment, and let N be the total number of fields that were updated during the experiment. The *amortized operation counts* are $n_w = N_w/N$, $n_r = N_r/N$, and $n_e = N_e/N$. The *estimated amortized times* per field modification, in nanoseconds, are

$t_{flash} = 100n_r + 30 \times 10^3 n_w + 500 \times 10^6 n_e$,

$t_{eeprom} = 100n_r + 30 \times 10^3 n_w + 3.5 \times 10^6 n_e$.

The performance data is presented in Figures 4, 5, and in table 1.

Two characterizations of the EEPROM implementation that emerges from all the graphs are that its performance is independent of the fullness of the device, and that its performance is independent of the size of objects and arrays. Both of these are due to the overwrite-in-place updating strategy.

Figure 4 shows the amortized number of writes per field-modification operation. On an EEPROM, each field modification must also be recorded in the log. The total number of writes, 11, includes the copying of the old value to the commit buffer (along with some meta-data), copying the new value to the object, and the invalidation of the commit-buffer entry.

In contrast to the constant performance of EEPROM, the performance of TINYSTORE is highly influenced by the fullness of the device and by the workload. The graphs lead to several important observations about the write performance of TINYSTORE.

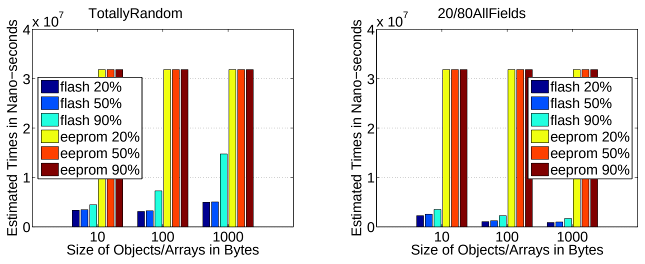

Figure 5. Estimated Amortized Time. The same experiments as in Figure 4.

- In most cases, TinyStore performs more write operations than the EEPROM implementation.
- However, when objects are large and all the fields of an object are modified before moving on to other objects, TinyStore performs fewer write operations than the EEPROM implementation. The relative inefficiency of the EEPROM implementation in this case stems from the copying of every field to the commit buffer; TinyStore, on the other hand, simply writes most of the new field values to empty slots in an already-allocated object record.
- TinyStore handles larger objects more efficiently than small ones in workloads that modify all the fields of an object. This is due to smaller overheads caused by the `REF` tree and by the allocation of object records.
- TinyStore handles larger objects *less* efficiently than small ones in workloads that modify only a single field in an object before moving on to other objects. The inefficiency is caused by the compaction of the object-record list every time the list reaches its maximal length. The cost of the compaction (in writes) is proportional to the size of the object, not to the number of fields that have been modified since the previous compaction.
- Under random-access workloads (*TotallyRandom* and *20/80AllFields*), performance drops as the device gets fuller. The drop is caused by inefficient reclamations. In an almost-full device that is updated randomly, the erase unit that is reclaimed usually contains a significant amount of valid data that must be copied to the new unit. This causes copying of data over and over again.

The number of read operations that needs to be done in EEPROM is low; these read operations are all associated with the commit buffer. In TinyStore however, `REF`s are indices into a pruned versioned tree, so TinyStore performs almost 50 times more read operations than the EEPROM implementation. However, reads are fast.

We also counted the number of erase operations. In the EEPROM implementation, each erasure is associated with a write operation, so their number is exactly the same. The amortized number of erases for TinyStore is much smaller, because each erasure erases 8 KB. Also, flash erasures are more than 100 times slower than EEPROM erasures. Therefore, there is not much point in comparing erase counts. We gain more insight from comparing estimated run times, which are dominated by erasure times for both flash and EEPROM. These estimates are shown in Figure 5.

In most cases, TinyStore is much faster than the EEPROM implementation, because it spends much less time in erasures. In most cases TinyStore is faster by a factor of 5–10. However, single-field modifications of large objects are slow, sometimes even slower than on EEPROM. This slowness is caused by many large but almost-empty object-records, which force TinyStore to erase frequently. In the *TotallyRandom* workload, a fuller device leads to higher run times, due to less efficient erasures: valid and obsolete data both populate every erase unit, so reclamations do not free much space.

Max length of lists	*Millions of field updates*	*Amortized number of reads*	*Amortized number of writes*	*Amortized number of erasures*
1	146	973.66	280.13	0.0384
2	3,301	291.92	12.02	0.0017
2*	3,359	288.71	10.94	0.0017
3	4,225	282.39	9.30	0.0013
5	4,868	245.65	8.03	0.0011
10	5,317	245.91	7.32	0.0010
20	5,533	222.75	7.02	0.0010

Table 1. The effects of the maximal length of object records, of the initialization flags* (third row is without them), . The workload is *20/80AllFields*, with objects consisting of 50 fields and a 50%-full flash. The amortization is per field modification.

Table 1 displays the results of an experiment that is designed to measure the effects of the maximal length of object-record lists and of the initialization flags. The second column measures the effects of these parameters on the endurance of the on-flash heap; we defer the discussion of this aspect until later in the chapter, and focus instead on columns 3–5 that measure performance. Clearly, allowing the lists to grow longer reduces the number of reads, writes, and erasures. This effect is dramatic at the low end of the allowed lengths and tapers off at longer lengths. The benefit of lists of length 2 is spectacular over lists of length 1. The effect is magnified by the fact that the workload that we used modifies all the fields of the object before moving on to another object. This implies that every slot in every object record is filled before the record is discarded. Increasing the maximal lengths from 2 to 3 still improves performance because the number of object-record-list compactions decreases, but from then on the effect of longer lists is weak.

The performance benefits of longer object-record lists come at a cost: the lists take up space, thereby allowing the system to store fewer objects on the same physical storage device.

The table also shows that in this experiment, initialization flags degrade performance. The degradation occurs in this experiment because the objects are long-lived; they are never deleted until the device wears out. Initialization flags only speed up the allocation of new objects. When this activity is rare, as it is in this experiment, the flags contribute little to performance, but incur a fixed cost at every object-record-list compaction. In a more dynamic workload, where objects sustain only few field modifications before they are deleted or garbage-collected, the initialization flags can improve performance.

The complex data structures of TinyStore cause field accesses (JVM field-read operations) to be high. To measure this cost, we filled the heap with 5,500 objects with two single-bytes fields in each. We then wrote to all the fields (to fill one object record), and then wrote to one of the two fields again, to create another object record. This filled the heap almost completely. The situation makes the `REF` tree as tall as possible and the object-record lists as long as possible; this is a worst-case situation for field-accesses in TinyStore. We now read all the fields, to measure the cost of field accesses. The average number of flash-reads per field is 731. This is obviously a high overhead for simply reading a field, but unless the workload is heavily read-dominated, writes and erasures are likely to dominate the running times, because they are more expensive. In

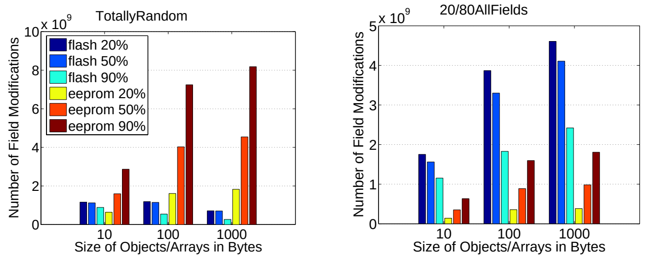

Figure 6. The number of field modifications that the system can perform before the device wears out.

read-dominated workloads the JVM should cache physical pointers to objects; after caching such a pointer, the cost of accessing a field is a single read.

Endurance

We evaluate endurance by counting the number of field modifications that the system can perform before the flash or EEPROM wears out (that is, before the first flash-erase-unit or EEPROM-byte is erased 100,000 times). The endurance data for our main experiments are presented in Figure 6.

In workloads that only modify a single object, the EEPROM wears out quickly. Once a given field in that object is modified 100,000 times, the device wears out. Under the same workloads, TINYSTORE can sustain at least 74×10^6 modifications (a factor of 730 better than the EEPROM implementation), and in some scenarios, it can sustain more than 4×10^9 modifications.

In the *TotallyRandom* workload, the EEPROM endures longer than TINYSTORE. The workload itself levels the wear on the different areas of the EEPROM, allowing many fields in many objects to be modified 100,000 times or close to that number. On the other hand, TINYSTORE incurs erasures that are due to the REF tree, to multiple records in objects lists, and to its explicit wear-leveling algorithm. But even in these nearly-best cases for the EEPROM implementation, the endurance of TINYSTORE is within an order of magnitude of that of the EEPROM implementation.

In the *20/80AllFields* workload, which is probably the most realistic of all the workloads, TINYSTORE always endures longer than the EEPROM implementation. When the device is relatively empty (20% full), TINYSTORE endures about 10 times longer, because its wear-leveling algorithm moves the 20% objects that are accessed 90% of the time around the device. Moving them around prevents their frequent modification from wearing out specific erase units. When the device is nearly full, TINYSTORE endures only slightly longer than the EEPROM implementation, because its erase units wear out quickly due to frequent ineffective reclamations.

In general, the endurance of TINYSTORE is inversely proportional to the number of erasures per field-modification that it performs. A high average number of erasures both slows TINYSTORE down and wears out the device quickly.

Table 1 shows that longer object-record lists deliver not only performance benefits, but also endurance benefits.

Capacity

Figure 7 presents the results of experiments intended to measure the storage overhead of TINYSTORE. In these simulations, we initialize TINYSTORE and then create fixed-size arrays until we run out of memory. The graphs show the percentage of the physical flash memory that is allocated to user data (i.e., to the fixed-size arrays).

We note that in these experiments, TINYSTORE creates only one object record for each array (or array section). When it runs out of space, it is essentially incapable of modifying these arrays. If we allowed for enough space for a length-2 list for each array section, the useful amount of storage would be roughly halved.

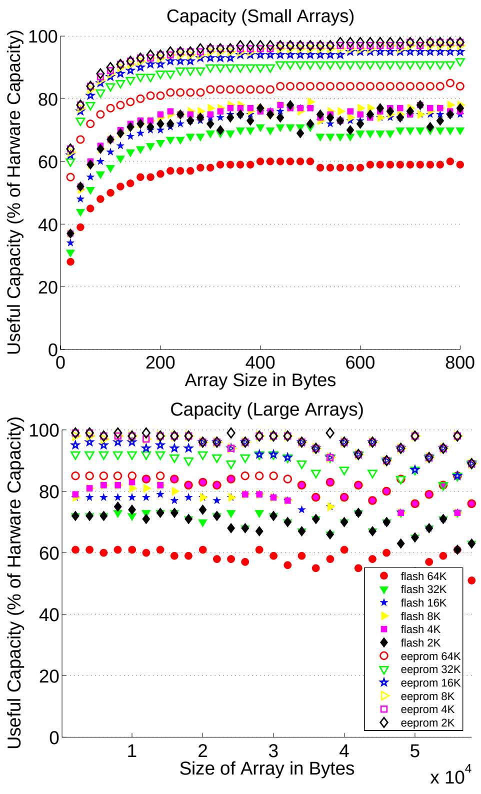

Figure 7. Capacity as a function of array size under different device and system configurations.

The different flash device configurations correspond to several erase-unit sizes.

The log in TINYSTORE is stored on a full erase unit, so the size of erase unit also affects the size of log, and hence the non-log storage available for user data. To account for that, we also configured the EEPROM implementation with the same log (commit buffer) sizes. The only other system-configuration parameter that we modified in this experiment is the use of initialization flags: we tested the default 8 KB erase unit size with and without these flags.

When objects or arrays are small (the smallest ones in these experiments are 10 bytes long), both systems incur significant storage overheads: more than 35% overhead for the EEPROM implementation, and more than 60% for TINYSTORE. In the EEPROM implementation, the overhead is caused almost exclusively by the header that the implementation associates with each object (not the JVM header, whose size is zero in these experiments). In TINYSTORE, the overhead is caused by the REF tree, by the object descriptors, and by the flags and `next` field of object records.

As the arrays grow, both systems store more user data in the same devices. The EEPROM implementation can fill 83–98% of the device with large user arrays, and TINYSTORE can fill about 60–80%. The modify-in-place EEPROM implementation is more space efficient than TINYSTORE.

The somewhat erratic behavior of the graphs on very large arrays (tens of thousands of bytes) is caused simply by the small number of arrays of this size that the 448 KB devices can store; it does not reveal an interesting behavior of the systems.

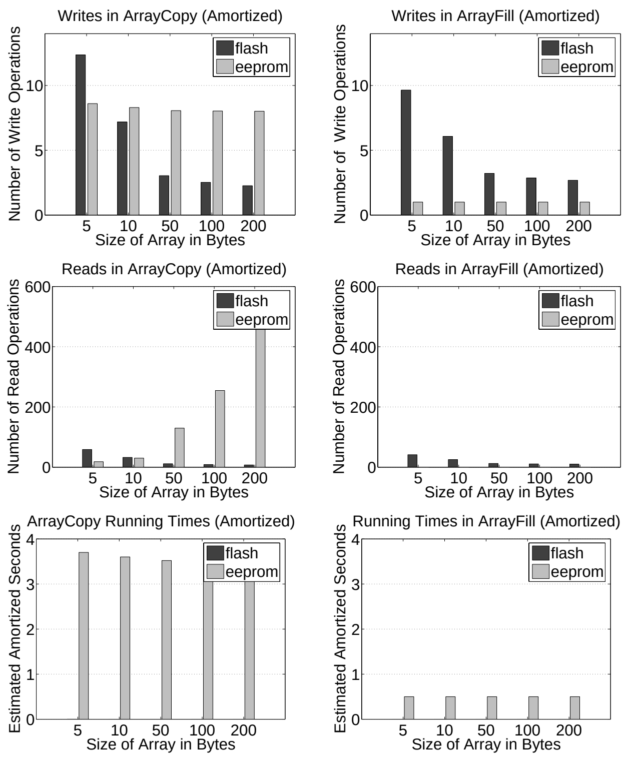

Figure 8. The performance of `ArrayCopy` relative to that of `ArrayFillNonAtomic`.

We ran another experiment to evaluate the effect of the length of object-record lists on TINYSTORE's capacity. The results clearly showed that the shorter the lists the higher the useful capacity of the system.

The Cost of Transactions

To assess the cost of transactions, we compared the performance of `ArrayCopy`, which is atomic, the that of `ArrayFillNonAtomic`, which is obviously not atomic. Copying an array atomically performs the same action as the following code:

```
BeginTransaction();
for (i = 0; i < length; i++)
dest[destOff + i] = src[srcOff + i]
CommitTransaction();
```

Therefore, the experiments also characterize the overhead of explicit transactions in both TINYSTORE and the EEPROM implementation.

Figure 8 shows the results of two experiments. In one experiment we called `ArrayCopy` on a single pair of arrays until the device wore out. In the other experiment we called `ArrayFill-NonAtomic` on a single array until the device wore out. The graphs show the number of read and write operations and the estimated times, all amortized per field modification.

TINYSTORE performs a little better in non-atomic operations than in atomic ones, but the difference is small. The small penalty for atomicity is a direct consequence of never overwriting data in place. The penalty that we do observe is a consequence of the fact that `ArrayFillNonAtomic` can use slots in an object record in an existing list, whereas `ArrayCopy` must create a new representation of the array and must insert it into the `REF` tree.

In the EEPROM implementation, atomicity causes significant overheads. The need to ensure atomicity causes a factor of 7–8 increase in both the number of writes and in the estimated run times. These are caused by the need to write old values to the log.

The amortized number of operations are calculated per field in the array. `ArrayCopy` is an atomic operation, so all the previous values need to be saved. The amortized number of writes for flash decreases as the array size grows, this is because `ArrayCopy` operations are similar to a sequential field update of the object, so the results are similar to the graph of the sequential field update. The main difference being that the object needs to be found only once, and not for every field that is copied. We can see that the amortized number of writes for the EEPROM is almost constant. (The small decrease is related to a constant overhead of 3 writes for the commit marker at the end of every `ArrayCopy` operation). The amortized number of reads for EEPROM increases significantly when the array size increases. This is because, when logging an 'old-value' in the log, we first need to traverse the log to make sure no other 'old-value' is logged for that field, therefore, each write operation requires scanning the log up until that point. The amortized number of reads for flash decreases, since the arrays need to be found only once in the search tree for the entire copy operation, unless the array is larger than the fixed array-segment size in which case a large-array tree is traversed; still, once the object is found, there are not many reads to perform. The number of erasures in EEPROM almost matches the number of writes, because each write requires an erasure. The amortized number of erasures for flash is very low (almost zero) and this is because an erasure is only required when an erase-unit fills. The cost is highly affected by the number of erasures. Since flash performs fewer erasures, the amortized cost is much lower as well.

When atomicity is not required, EEPROM performs much better in terms of amortized number of reads and writes. This is because EEPROM implementation does not write every old value to the log. TINYSTORE performs slightly better in comparison to its performance for the atomic copy, but not by a large factor. As in the `ArrayCopy` case, erasures dominate the running times, so TINY-STORE performs better.

7. Summary

We have presented a novel approach for storing a Java heap on NOR flash memory. To the best of our knowledge, this is the first attempt to implement a memory manager for Java Cards that is intended for NOR flash devices with fairly large erase units (a device with erase units of only a few bytes can be treated as EEPROM).

Managing NOR flash is complex. To address the difficulty, we developed novel data structures and utilized sophisticated existing flash-specific data structures [6]. Prior work, especially on flash file systems, does not address all the requirements of Java-Card heaps; our innovations allow TINYSTORE to efficiently modify even single-byte fields.

We have also developed a mechanism to support mark-sweep garbage collectors. Our design allows garbage collection in systems that have very little RAM. Our garbage-collection-support API allows the collector to run with only a small constant amount of RAM. The NOR flash is used for marking objects and for storing the mark-stack. Garbage collection for Java Cards may seem like a luxury, but we believe that garbage collected Java Cards will dominate in the future, as new sophisticated Java-Card applications are developed.

Extensive comparisons of TINYSTORE to a traditional overwrite-in-place EEPROM implementation lead to several important conclusions. TINYSTORE usually outperforms the EEPROM implementation in terms of estimated times. This is mostly due to fewer erasures, even though individual flash erasures are slower than individual EEPROM erasures. This shows that TINYSTORE exploits the main advantage of flash memories over conventional EEPROM: fast bulk erasures. Much of the performance benefits of TINYSTORE stem from its efficient atomicity and transactions mechanisms compared to the relative inefficiency of a traditional commit buffer. TINYSTORE usually endures longer in typical workloads that contain some frequently-modified objects or arrays. It endures much longer in highly-unbalanced workloads with very frequent modifications of very few objects. Moreover, the endurance of TINYSTORE is less dependent on the workload than the endurance of overwrite-in-place EEPROM implementations.

When the flash device is nearly full, both the performance and the endurance of TINYSTORE degrade significantly. When the flash is nearly full, erase-unit reclamations, which are supposed to release deallocated space, reclaim only small amounts of space. Since each reclamation erases a unit, ineffective reclamations increase the erasure frequency, thereby slowing down the system and wearing out the device.

TINYSTORE can store less user data in a given amount of storage than an overwrite-in-place EEPROM implementation. Furthermore, the storage overheads of TINYSTORE are less predictable than those of the EEPROM implementation.

The main benefits of TINYSTORE over the EEPROM implementation are its improved performance and endurance. On the other hand, its higher storage overheads and its relatively poor performance/endurance on nearly-full devices are likely to offset any chip-density benefit that flash might have over traditional EEPROM.

The complexity of TINYSTORE and its performance characteristics are both strongly influenced by three of the constraints under which it was designed: large flash erase units, small RAM size, and the need to manage small data objects. Two other characteristics of NOR flash mitigate some of the difficulty: the ability to clear additional bits in an already written word, and the fast read times. However, even with these mitigating factors the three constraints still require a complex design with somewhat peculiar performance characteristics (especially on a nearly-full device).

Therefore, perhaps the most important conclusion is that system designers can improve the situation by removing at least one of the three constraints. Larger RAM sizes can simplify the design, although it is unclear whether large RAM size can improve performance and endurance on nearly-full devices. Small erase units, or better yet, the ability to erase both small and large blocks, can probably improve many aspects of TINYSTORE and of Java Card systems in general.

Integrating some of the ideas of TINYSTORE into a more conventional overwrite-in-place heap design may help exploit small or variable-size erase units. For example, frequently-modified small objects may be represented by a few object records on a single (small) erase unit, to reduce erasures. Similarly, the use of logical pointers and the occasional random swapping of large blocks can improve wear leveling and endurance in an otherwise overwrite-in-place system.

It appears that with hardware endurance of 100,000 erase cycles or more, which is typical on today's devices, endurance is not a significant issue for current applications of Java Cards. We believe, however, that in the future this factor may become much more important. The Java Card platform, with its beautiful and unconventional memory model, is used today mostly in smart cards. Application activity in a smart card today is usually linked to a physical activity: inserting a card into a reader, making or receiving a phone call on a mobile phone, making a purchase, the shipment of RFID-bearing products, and so on. But emerging applications of smart cards (e.g., micropayments) and possibly the introduction of Java Card or similar platforms into a wider range of embedded systems (sensors, automotive systems, etc.) can significantly increase the erasure rate and shorten the lives of non-wear-leveled systems. In such future systems, simple overwrite-in-place heaps may be unacceptable.

References

[1] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[2] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *OOPSLA*, pages 403–417, 2003.

[3] Zhiqun Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 2000.

[4] J-M. Douin, P. Paradinas, and C. Pradel. Open benchmark for java card technology. Presentation given at *e-Smart 2004*, Sofia Antipolis, available from `http://deptinfo.cnam.fr/~paradinas/presentation/E-smart-09-2004.pdf`, September 2004.

[5] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, 2005.

[6] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Meeting*, 2005.

[7] Sun Microsystems Inc. JSR-12: Java data objects specification. Technical report, 2001.

[8] Sun Microsystems Inc. The Pjama project. Technical report, 2001. available online at `http://research.sun.com/forest/opj.main.html`.

[9] Richard Jones and Rafael Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[10] Im Y. Jung, Sung I. Jun, and Kyo I. Chung. A persistent memory management in Java Card. *WSEAS Transactions on Systems*, 2(1):160–166, 2003.

[11] Brian T. Lewis and Bernd Mathiske. Efficient barriers for persistent object caching in a high-performance javatm virtual machine. Technical Report TR-99-81, Sun Microsystems Inc., 1999.

[12] Brian T. Lewis, Bernd Mathiske, and Neal Gafter. Architecture of the pevm: A high-performance orthogonally persistent java[tm] virtual machine. Technical report, Sun Microsystems Inc., 2000.

[13] Tom Lunney and Aidan McCaughey. Information systems: Object persistence in java. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java PPPJ '03*, June 2003.

[14] Sun Microsystems. Java Card technology at a glance. Press kit for the JavaOne conference, available at `http://www.sun.com/aboutsun/media/presskits/javaone2005/`, 2005.

[15] J. Eliot B. Moss and Antony L. Hosking. Approaches to adding persistence to java. In *First International Workshop on Persistence and Java Drymen, Scotland*, September 1996.

[16] Marcus Oestreicher. Transactions in java card. *15th Annual Computer Security Applications Conference*, pages 291–298, December 1999.

[17] Marcus Oestreicher and Ksheerabdhi Krishna. Object lifetimes in Java Card. In *USENIX workshop on Smartcard Technology*, Chicago, Illinois, May 1999.

[18] Pierre Paradinas. SCCB: Smart card CNAM benchmark. A web page at `http://deptinfo.cnam.fr/~paradinas/sem/sccb/`, April 2005.

[19] Sharp. Addressing security concerns of flash memory in smart cards. Application Note SMA04036, 2005.

[20] Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.2.1 Platform API Specification*, October 2003.

[21] SUN Microsystems, Inc. *Java Card 2.2.1 Runtime Environment (JCRE) Specification*, October 2003.