

USENIX Association

Proceedings of the
Java™ Virtual Machine Research and
Technology Symposium
(JVM '01)

Monterey, California, USA
April 23–24, 2001



© 2001 by The USENIX Association

All Rights Reserved

For more information about the USENIX Association:

Phone: 1 510 528 8649

FAX: 1 510 548 5738

Email: office@usenix.org

WWW: <http://www.usenix.org>

Rights to individual papers remain with the author or the author's employer.

Permission is granted for noncommercial reproduction of the work for educational or research purposes.

This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Automatic Persistent Memory Management for the Spotless Java™ Virtual Machine on the Palm Connected Organizer*

Daniel Schneider[†], Bernd Mathiske, Matthias Ernst[†], Matthew L. Seidl

Sun Microsystems, Inc.

901 San Antonio Road

MS MTV29-112

Palo Alto, CA 94303-4900

Daniel.Schneider@coremedia.com, Bernd.Mathiske@sun.com,

Matthias.Ernst@coremedia.com, Matthew.Seidl@sun.com

Abstract

Palm organizers are widely used in a multi-tasking fashion. *Users* switch from one application to another without losing the context established in either of them. Despite its obvious usefulness there is no *automatic* support for this convenience in the organizer's operating system, PalmOS. *Programmers* must implement event callbacks that use the PalmOS database API to save and reload specific application state. In this report we describe how this burden can be eliminated.

We enhanced the Spotless Java™ virtual machine for the Palm organizer with transparent multi-tasking support that automates persistence.

As a consequence, running Java programs can be transferred between two Palm organizers using the infra-red link. A transferred program will resume on the receiving organizer in the exact same state as it had on the sender. In addition, a HotSync operation can effectively be used to establish a checkpoint for each Java program involved.

Originally, the address range available for running

*TRADEMARKS Sun, Sun Microsystems, the Sun logo, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

[†]now at: CoreMedia AG, Düsternstrasse 3, D-20355 Hamburg

programs was restricted to a few tens of KB in the *dynamic* RAM area. By directly addressing the much larger *static* RAM area our modified VM supports address ranges of several MB.

We provide an easy-to-use protocol that leverages persistent threads for automatic life cycle control of *external* resources (e.g. windows, forms and databases). When applied at the library level, this protocol maintains complete persistence transparency for the application programmer.

1 Introduction

The Spotless VM is a Java™ Virtual Machine for the Palm connected organizer that has been developed at Sun Microsystems Laboratories with the goals of small size, portability and readability of its source code [Taivalaari et al., 1999]. A version of Spotless was then further developed to become the KVM [Sun, 1999], an extremely lean implementation of the Java Virtual Machine for use in devices that have very restricted physical memories (tens of kilobytes). The KVM is one of the core elements of the Java™ 2 Micro Edition (J2ME), which is targeted at consumer electronics and embedded devices. It has been ported to and is supported on a variety of platforms.

The Spotless VM runs only on PalmOS, the operating system of Palm connected organizers. It is a research vehicle at Sun Microsystems Laboratories for exploring new ideas and technologies for program-

ming on small devices. In this report we describe how we enhanced the Spotless VM with memory management capabilities that automate persistence, that is, saving and restoring of code, objects and threads.

The Palm users' experience is that an application's state is saved when another application is invoked and that said state is restored when the former application is resumed. Given that the device's RAM is battery-backed, this behavior seems only natural to users. However, this advantage over desktop systems is only apparent to users – not to developers.

Application switches can happen at any point during program execution. Hence, a Palm application must be able to handle persistence at any point. The operating system, PalmOS, offers support by invoking specific event callbacks before every application suspension and resumption. Application programmers must implement these callbacks to provide the illusion of multi-tasking.

When writing suspension and resumption code, application programmers have to deal with the traditional dichotomy between primary and secondary memory. Only 64 to 256 K bytes of the Palm's RAM are freely accessible and available to represent dynamic application program state. The OS, using hardware memory protection, guards *all* write accesses to the larger part of the RAM and imposes a simple database API.

Unfortunately, a single write access through the database API can take 100ms or longer – the more items stored in the database, the slower the access. This leaves programmers with no choice but to carefully write significant amounts of code for the purpose of storing and recovering long-lived data – an error-prone task that repeats as new applications are developed and existing ones are modified and extended. This process very much resembles the work done by a multitasking OS or a thread library when performing a context switch, namely to save volatile processor state to memory.

In order to gain efficient write access, we bypass the database API by calling a undocumented PalmOS function that disables the memory protection. Thereupon all RAM on the device has the same access performance. By representing application stacks and heaps directly in the newly unprotected RAM area, we manage to eliminate the distinction between primary and secondary memory and

to make all program state persistent.

In the next section, we describe relevant features of the Palm device and operating system. In section 3 we then present our implementation of persistence automation. We cover memory management, program life cycles, and a protocol to integrate external state that is not under direct control of the runtime system. Section 4 discusses the safety issues that are introduced with our solution. Section 5 presents the previous work most closely related to this research, and Section 6 presents our conclusions.

2 The Palm Connected Organizer

Due to their relatively restricted requirements, handheld devices generally have quite different architectures than desktop systems. In this section, we give a brief overview of the Palm platform and the PalmOS operating system for readers who are not familiar with either.

2.1 The Device

The Palm Organizer is equipped with a Motorola 68000 compatible processor (the 16 MHz Dragonball MC68EZ328 in the case of the Palm V) and from 2 to 8 MB of RAM backed by the battery such that its state is preserved even if the device is switched off. The display has 160 by 160 pixels.

2.2 The Operating System

PalmOS is a simple, single-tasking operating system with a GUI widget set that enables the application developer to build event driven, graphical applications.

The 2 to 8 MB of physical RAM is divided into two parts:

- 64 KBytes of “dynamic memory” in older version of PalmOS, and up to 256 KBytes in more recent versions. This is the amount of RAM in which a PalmOS application executes. Memory in this area can be acquired from the operating

system through the `MemPtrNew` system call and then be freely written and read.

- The remaining memory – called “static memory” – is battery-backed. This area is maintained by the OS as secondary memory. It is organized as a list of “databases” with mostly untyped records of variable size to which the user can obtain a handle using the `DmGetRecord` call. By locking such a handle using the `MemHandleLock` call, a pointer to a record’s location can be obtained. As long as a handle is locked the operating system guarantees not to move the associated record in memory. The pointer returned by `MemHandleLock` only allows read access to the record’s memory area. Writing has to be performed using the `DmWrite` call – direct write access will result in a fatal exception and will eventually reset the device.

This partitioning of the device’s physical memory can be avoided by using the undocumented system calls `MemSemaphoreReserve/MemSemaphoreRelease`.

These will switch the memory protection of the “static” memory area off and on. When these calls are used, great care has to be taken when issuing other system calls as some may need the semaphore. This forces the programmer to release the memory semaphore before entering certain system calls.

As mentioned in the introduction, PalmOS has no automatic multi-tasking capabilities. If a user changes from one application to another, the first one will be asked to stop by an `AppStop` event. Once it has stopped, all dynamic memory will be reclaimed by the operating system. The second application will then be started using the same dynamic memory as the first. The operating system does not provide any implicit support for saving the context of an executing application. Instead, the application programmer is forced to save the state explicitly, in a database residing in static memory.

3 Automation of Persistence

Most computing environments distinguish transient primary memory (RAM) and stable secondary memory, typically on hard disks. Because the latter normally has much slower access characteristics,

programs operate in primary memory and persistence is achieved by copying to secondary memory and restoring from there.

PalmOS treats dynamic RAM as primary memory and static RAM as secondary memory. But on the Palm device the *physical* access speed of dynamic and static RAM are exactly the same! Only because PalmOS controls accesses to static RAM by certain exclusive access functions does it appear to behave like “typical” secondary memory.

By circumventing this control, though, and making more direct use of static RAM, we can eliminate the distinction between primary and secondary memory. Hence, no data has to be copied between the two areas when a program is suspended.

The Spotless VM was first modified to have a “persistent heap,” which is directly allocated in stable memory. The bytecode interpreter still uses dynamic memory, but all of the Java data (including bytecodes and threads) is allocated on the Java heap and thus in static memory. In order to be able to freely write to the heap, the VM uses the system call `MemSemaphoreReserve`, holding the semaphore during the normal interpretation of bytecodes and only releasing it when entering certain system calls that require it.

However, allocating the heap in “static” memory does not unburden the VM from having to shut down and resume as a Palm program, as discussed in section 3.2. Also, this “in place execution” inhibits the possibility of making a snapshot of a heap and then continuing execution without overwriting the snapshot data. We still chose this implementation because it greatly speeds up the shut down process as almost no data has to be copied to stable storage. Snapshots can still be achieved by stopping an application and then transferring the persistent store to a desktop system using the Palm device’s serial port.

3.1 Store Structure

Although we eliminate the distinction between primary and secondary memory for objects, we borrow the notion of a “store” from persistent object caching systems to describe the entire durable memory image of an application. A Spotless store captures the execution state of a running program, its

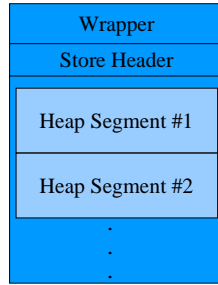


Figure 1: Store Structure

heap, and additional boot information necessary to resume the virtual machine at the point where it was interrupted.

Such a persistent store is represented by a PalmOS resource database of the type 'appl'. This type tag causes the application manager to display the store like any other application installed on the Palm. Figure 1 illustrates the basic structural components of a store:

1. The wrapper, which contains a minimal boot program (less than 1 KB in size) that simply locates the Spotless VM program on the device and then starts it, passing the user-indicated store as a parameter.
2. The store header, which contains all global C variables that cannot be re-established from the context. These are mostly the various roots of the system, that is, references to objects located in the segmented heap. It also contains pointers to the first and the last of the resource database records that represent the store contents. The store header itself is stored in a resource record of type 'STOR'.

Figure 2 shows the definition of the associated C structure. The field `externalManager` will be explained in more detail in 3.4.

3. The segmented persistent heap. This is the area where objects are placed by the allocator and moved or deleted by the garbage collector. Each segment is stored in a resource database record of type 'VMem' and is 64KB in size (the largest contiguous area that can be allocated in any Palm database).

Each record has its own header block containing status information for the segment. Figure 3 shows the definition of the C structure that contains the header data.

```

struct storeStruct {
    int     lastRecordId;           // ID of the last record
    RECORD  first;                 // First record of this store
    RECORD  last;                  // Last record of this store
    RECORD  nextAllocation;        // Record in which the next
    // object allocation should take place
    cell *  bitmap;                // Image of the Palm screen
    INSTANCE externalManager;      // when the store was suspended
    // Singleton manager of external data

    // These are VM roots and globals:
    THREAD  UP;                    // List of all threads in the VM
    int     nActiveThreads;        // Number of active threads
    BYTE *  tagStack;              // Marks pointers on the current stack
    ...
    int     elapsedTime;           // The time the store was up
};

```

Figure 2: C Data Structure Representing the Store Header

```

struct recordStruct {
    int     id;                    // The database ID of this record
    RECORD  next;                  // The next record in the list or NULL
    int     size;                  // The number of cells in this record
    cell *  bottom;                // Bottom of this record
    cell *  top;                   // Top of this record
    CHUNK  firstFreeChunk;         // Allocate the next object at this location
    int     bumped;                // TRUE when the underlying MemBlock
    // starts 2 bytes before this record
    cell *  breakTable;            // Break Table used
    cell     breakTableSize;       // by the garbage collector
    int     address;               // The address where this record was located
    // when the store was suspended
};

```

Figure 3: C Data Structure Representing a Heap Record

The segmented heap is represented by a linked list of records using the `next` pointer to locate the following one.

The VM requires all heap objects to be located at a 32-bit aligned address, whereas the operating system may return addresses that are only 16-bit aligned. This problem is solved by actually requesting 16 bits more than needed to hold the record, which leads to one of the following situations.

If the memory area returned by the OS is not 32-bit aligned, the record is “bumped” up to the next 32-bit aligned address. When an application is suspended, the fact that the record was bumped is indicated by setting its header’s `bumped` field to `TRUE` and the two bytes before the record to `0xFFFF` (Figure 4). This will be recognized by the VM when it resumes the application, and then it can properly align the contents of the record as described further in section 3.2.

If the memory area returned by the OS is 32-bit aligned, the record is simply placed at the beginning of that memory area leaving 16 bits at the end of it unused.

3.2 Lifecycle of a Persistent Program

Program execution comprises the following stages:

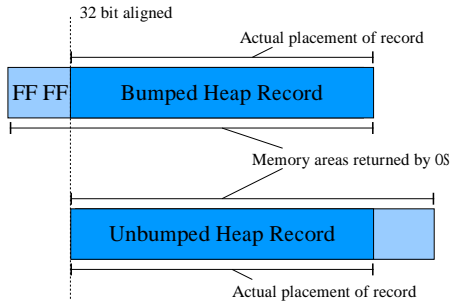


Figure 4: Bumping of Unaligned Addresses

Creation: A program is first started and a new store is created.

Execution: The program runs.

Suspension: The user switches to another application. The current application is interrupted and made persistent.

Resumption: The user resumes a previously suspended application. Execution will continue from the exact same state where it was left off.

Destruction: The computation terminates. This can be caused either by calling `System.exit`, terminating the last non-daemon thread or by not catching an exception. The store is then deleted.

The next sections will describe these four stages and associated actions taken by the VM in more detail.

3.2.1 Creation

When a new computation is started the user must supply a name for the store that will be created. The VM will then create a new PalmOS resource database of type 'app1' and will first copy the contents of the wrapper database (`Wrapper.prc`) to the beginning of the newly created store.

Then the VM creates a store header record and a first heap record and initializes all fields. These records remain locked allowing the VM to access them directly.

3.2.2 Execution

The limited size of Palm database records dictates a two-level allocation scheme. When in the course of program execution the amount of available memory in the current heap segments becomes insufficient despite garbage collection efforts, another record is allocated and locked in memory. New records are appended to the end of the linked list of heap records.

Failure of record allocation is signaled by an exception.

3.2.3 Suspension

When the user changes to another Palm application, the VM automatically saves the current state of the computation by closing the persistent store in a controlled manner:

1. All objects whose state is partly transient (i.e., not located on the persistent heap) are requested to internalize their current external state. This process is further explained in 3.4.
2. Individual modules of the VM, such as the event handling system, are requested to memorize their state. This is implemented by copying C variables either to the heap or to appropriate fields in the store header.
3. All heap records and the store header are unlocked and can then be freely moved by the operating system.
4. The store database is closed. It will now appear as a normal PalmOS application in the application manager.

3.2.4 Resumption

In order to continue a computation from a persistent store, the user simply selects it in the application manager. The operating system will then invoke the wrapper code at the beginning of the store which will in turn locate the `SpotlessVM` and invoke it, passing the store as a parameter. The VM will then perform the following steps in order to resume the suspended computation.

First, it will open the store database and map the store header to the appropriate C `struct` (see Section 1). The store header is the only record of type `'STOR'` in the database and can thus be uniquely located by the VM.

Next, the VM will start opening the individual heap records – these can be identified by their record ID (ranging from 1 to `lastRecordId`) and type (`'VMem'`). After opening and locking a record, the header information has to be updated in case the record has been moved since it was suspended. This operation involves three major changes to the record header.

1. **Adjust the “bump” state.** If the record has been moved from an aligned to an unaligned address or visa versa, its bump state has changed. The new bump state can be obtained from the record’s new memory address. The old bump state can be detected by reading the first two bytes of the record’s new memory area. If these contain `0xFFFF` the record was bumped when the store was suspended. If the old and the new state differ, the record has to be moved by 16 bits to a 32-bit aligned memory address. Note that the required space is always available as the initial allocation requested 16 bits more memory than necessary to hold the heap record (Figure 4).
2. **Adjust header fields.** If the record has been moved, all pointers from the header into the heap segment need to be adjusted. The old location of the record can be read from the header field `address` and be compared to the record’s new location.
3. **Append an entry to the break table.** We build on an improved version of the SpotlessVM with a compacting garbage collector that uses a break table [Haddon and Waite, 1967] to update pointers to relocated objects. We reuse the garbage collector code to adjust pointers to objects located in heap records that have moved (see 3.3 for more detailed information on the garbage collection algorithm). At this point the VM simply has to append an entry to the global breaktable to reflect the movement of the current record.

After all records have been opened and the appropriate break table entries have been appended, the

VM runs a (slightly modified) pointer update phase of the garbage collector in order to adjust the pointers within the heap and from global C variables (still being stored in the store header) into the heap.

After all pointers have been updated, individual modules of the VM are asked to restore their state from the contents of their associated store header field.

Then callbacks that have been registered to recreate external state are executed (see 3.4).

Finally, the virtual machine resumes the execution of the application from the point where it was suspended earlier.

3.2.5 Destruction

When a program terminates – either by calling `System.exit`, terminating the last thread or by not catching an exception, the VM simply deletes its entire store. In case of abnormal termination this may seem a bit harsh, but considering that the store would be useless as it cannot be restarted and – without a tool like a store browser – cannot be analyzed for errors, it seems the right choice. As mentioned earlier, checkpoints of a store can and should be achieved by hotsyncing a store to a desktop system. This behavior could be changed when tools exist to do a post-mortem on programs that exit abnormally.

3.3 Memory Management

Store records can be moved in memory by the operating system, when the store is inactive. To compensate for this, as mentioned in the previous section, pointers need to be updated. This requires the VM to be able to tell the difference between a pointer and scalar data at any point where a pointer adjustment may take place. Otherwise, say, an `int` could be misinterpreted as a pointer and would thus be modified after an object (or a record) was moved in memory – leading at best to an erroneous computation.

Being able to tell *exactly* which values are pointers and which are scalars is referred to as *exactness* or *type accuracy*. It can be achieved by:

- keeping separate reference maps indicating pointer locations,
- directly tagging each individual value with at least one bit of type information.

The Spotless VM’s memory management identifies a heap cell’s type by analyzing the class of the object that the cell belongs to. Thus an object’s class stores which fields contain pointer data and which scalar data. To identify a stack cell’s type, the VM keeps a parallel stack containing the required type information. The secondary stack is called a tag stack, as it keeps a tag corresponding to each stack value and whether it’s a pointer or not. Each time the execution stack is modified, the tag stack is updated accordingly.

The garbage collector consists of a mark-sweep algorithm with sliding compaction that uses a break table to update pointers to moved objects [Jones and Lins, 1996]. Although this algorithm is relatively slow, it has two crucial advantages:

1. It does not need any extra space during the compaction/update phase. The break table is built in space that moved objects leave. This is important, as on a very memory limited device, there may not be enough additional memory available for many other garbage collection algorithms.
2. It can adjust pointers “into” any part of objects, not only pointers to the beginning of objects. We inherited this requirement from the original design of the Spotless VM, which features some pointers “into” objects for performance reasons (e.g., a method is directly referenced from a stack frame when executed, but it is not represented by its own heap object; instead, it is part of a method table heap object).

The current garbage collection algorithm does not move objects between heap segments. Although it is a compacting collector, this means there is still some fragmentation. Furthermore, when a computation is suspended, the VM will almost never find an empty segment that could be returned to the operating system. We have considered alterations to the algorithm to make it “multi-segmented”, but would suggest that later implementations should instead take advantage of the segmentation and implement a generational collector.

3.4 External Data

When a program communicates with its environment, this often involves the creation of some external state that is not under the control of the program’s runtime system. In case of the Spotless VM this typically concerns PalmOS structures such as windows, forms or databases. In this section, we introduce a protocol that provides automatic control over the life cycle of external state.

In order for external state to use our protocol, it has to be encapsulated in an instance of a developer-defined class that implements the interface **External** as shown in figure 5. We then regard an **External** as consisting of two sub-states:

External state: data external to the persistent store and references thereof. This also includes attributes representing operating system structures, such as a file descriptor or a form handle that will usually be invalid after the store has been suspended.

Internal state: a representation of the external state that is not dependent on any data structures outside the control of the persistent store. The **External** must be able to reconstruct its exact external state from the internal state including, for example, the read/write position of a file handle.

An **External** has to synchronize its internal state with its external state when the store is suspended and vice versa when it is resumed. In order for this to be performed in a thread-safe manner Spotless was adapted to use a protocol adopted from the Tycoon-2 system [Weikard, 1998, Gawecki and Wienberg, 1998] and implemented in the **ExternalManager** singleton class.¹

All state transitions of **Externals** are controlled by a global **ExternalManager**. By adhering to this control center’s protocol, the external state of a computation can be internalized on suspension and re-established on resumption in a manner that prevents inconsistencies. Figures 5 and 6 show the pertinent interfaces.

¹Tycoon-2’s protocol for external resources builds on the basic protocol of Tycoon-1, which was first sketched in [Mathiske et al., 1995, Mathiske et al., 1997] and described in detail in [Kornacker, 1995]. The latter also influenced the API and protocol presented in [Jordan and Atkinson, 1999].


```

public interface External {
    public void createExternal() throws ExternalException;
    public void internalizeExternal() throws ExternalException;
    public void recreateExternal() throws ExternalException;
    public void destroyExternal() throws ExternalException;
}

```

Figure 5: The Interface for External Resources

```

public class ExternalManager {
    // "Freezes" the execution state of the VM:
    private native void stabilizeStore();
    public synchronized void stabilize();
    public synchronized void createAndRegisterExternal(External x)
        throws ExternalException;
    public synchronized void unregisterAndDestroyExternal(External x)
        throws ExternalException;
}

```

Figure 6: Major Methods of the ExternalManager Class

All four methods shown in Figure 5 must be implemented by an **External** to participate in the **External** protocol. By calling the method `createExternal`, the **ExternalManager** asks the **External** to create its external state. An **External** representing a database would, for instance, issue the appropriate OS call to open the database. The call `internalizeExternal` tells the **External** to internalize all external state, e.g., read the position in a database into an slot so the current state can be reestablished later. By calling `recreateExternal`, the **External** is asked to recreate its external state from its current internal; and by calling `destroyState`, the **External** is asked to destroy its external state, e.g., close the associated database.

The above four methods are invoked exclusively by the **ExternalManager** during different state changes of the persistent store. This means that in order to create external state, the **External**

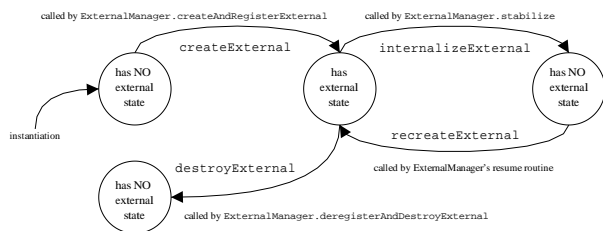


Figure 7: Life cycle of an External

has to register with the **ExternalManager** using the call `createAndRegisterExternal`. The **ExternalManager** will in turn call back to the **External** method `createExternal`. Unregistration is regulated in a similar manner.

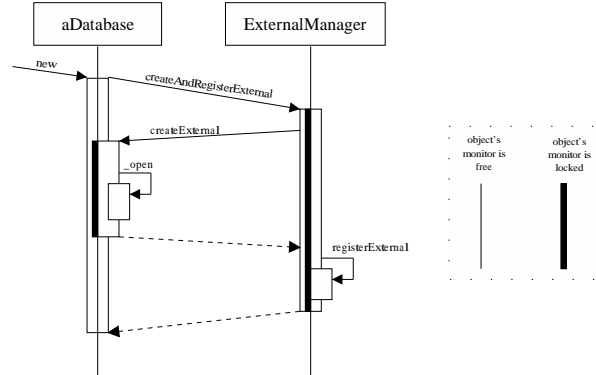


Figure 8: Registering with the ExternalManager

Figure 7 shows the major state transitions from the perspective of an **External**. Figures 8 and 9 show examples of a database class registering and unregistering with the **ExternalManager**.

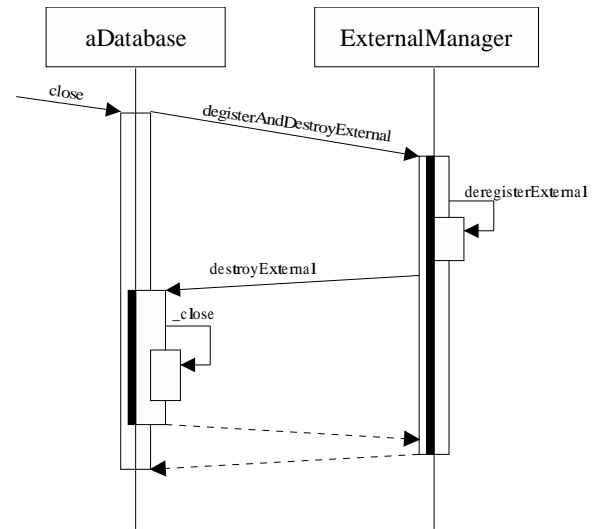


Figure 9: Unregistering from the ExternalManager

To avoid deadlocks at stabilization, the order in which the *monitors* (object locks engaged by the `synchronized` keyword) of the **External** and the **ExternalManager** are acquired is crucial. In the examples, a thick lifeline denotes a locked monitor, a thin one a free monitor.

The rationale for the stabilization protocol is as follows:

1. While a mutator thread is performing work on an **External**, its could potentially be in an inconsistent state. This prevents us from simply stopping all mutator threads before stabilization, i.e. stabilization must be performed in the running system.
2. After external state has been internalized, no mutator thread must be allowed to manipulate the **External**. Consequently, the stabilizing thread must hold on the **External**'s monitor after internalization.
3. No **External** may be registered or unregistered during stabilization.²

This leads to the following protocol: when the Spotless store is stabilized (which is triggered by the user changing to another application), the **ExternalManager** is first locked. After that, it gives all registered **Externals** the chance to internalize their external state in the reverse order to that in which they registered with the **ExternalManager**. This is performed by (i) acquiring the **External**'s monitor and (ii) sending the **External** the `internalizeExternal` message. At stabilization time, all **Externals**' monitors are held.

On application resumption the process proceeds in the reverse order. The **ExternalManager** will (iii) send the **External** the `recreateExternal` message and will then (iv) release the object's monitor. The recreation messages are sent in the same order in which the `createExternal` methods were once invoked.

To avoid deadlocks involving a stabilizing thread it is necessary that the **ExternalManager** object is always locked *before* the **External**. Note that the `new`, `_open` and `_close` methods in the examples are not synchronized. In general, no method that calls `createAndRegisterExternal` or `unregisterAndDestroyExternal` may synchronize on the **External** or else a deadlock is possible.

Figure 10 illustrates the stabilization process with two registered **Externals**. Note that the thick line representing the lock on the **External** is actually starting before the subsequent invocation of the `stabilizeStore` method and is held until the **ExternalManager** sends the `recreateExternal`

²This is for simplification. A more complicated scheme interleaving internalization and (un-)registration could possibly be thought of.

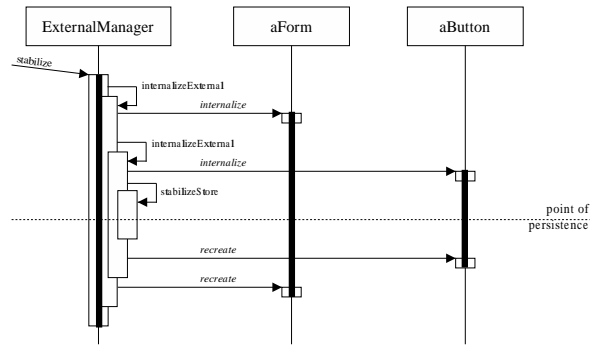


Figure 10: Performing the **ExternalManager.stabilize** Operation

call. Also note the use of persistent threads: the stabilizing thread is simply frozen by calling `stabilizeStore`; it holds on to its locks until it is resumed and then continues execution.

Recapping this section we observe that the **External** protocol offers a thread-safe way to manage a persistent application's external data. It provides an atomic registration/unregistration mechanism and is instrumental in preventing deadlocks and the persistence of inconsistent state.

4 Remaining Safety Issues

By disabling PalmOS' write protection as described in section 3, we create a new safety problem: it is possible for a VM to overwrite the memory of dormant applications and cause damage to them. This would need to be addressed in case our contributions were made into a product.

One could argue that a correctly implemented VM would never attempt any access outside its own store. Hence, debugging the VM and all added native code would "asymptotically" eliminate the problem. However, if one wants to be on the safe side, some form of hardware write protection is required.

If we accepted the limited amount of primary memory in the original Spotless VM, there would be a simple solution. We might then have implemented the persistence of execution state in dynamic RAM by copying all of it into a database record on shutdown and read it back into dynamic RAM on resumption. However, we do not have any intention

to claim: “64 K should be enough for everybody.”

It should be possible to protect only part of the static RAM, *not all* of it. One could then arrange during each switch between applications that all passive programs are protected and that only records of the designated active application are exposed to unprotected write access. For example, if the hardware protection is governed by an address limitation register, then it would be straightforward to relocate each application so that only the running one resides in dynamic RAM.

The remaining problem is that memory *outside* records could be altered inadvertently, which would compromise the integrity of meta data (e.g., handle collections, database indices) managed by the OS. Ideally, the OS would place meta data in the protected memory range. In case it does not, a promising approach would be to save all of the meta data occurring in the unprotected memory area to a database before they can be altered and to restore them before yielding control to another program or the OS.

5 Related Work

Pocket Smalltalk [Arsenau and Brault, 1999] is a Smalltalk-80 implementation for the Palm Connected Organizer. Like our system, Pocket Smalltalk circumvents PalmOS’s database API and implements an in-store execution model by directly modifying its segmented heap in the static RAM area. The system does not offer an elaborate protocol to manage the persistent state of external data.

MERPATI [Suezawa, 2000] extended the JDK 1.1 VM through a checkpointing mechanism for purposes of error recovery and migration of computations. After suspension of all computation, the whole program state is transferred to disk. Contrary to our work, MERPATI does not offer a checkpointing protocol to internalize partially external state. Also, the system does not persist class file contents but requires them to be installed at every migration target. MERPATI employs a conservative garbage collector, so type accurate descriptions of stack contents are only needed for checkpoints. Consequently, the system computes reference maps on demand, instead of updating them with every byte-code instruction, as our system does.

The PJama research prototype [Atkinson and Jordan, 2000] features a modified high-performance Java 2 VM and runs on top of a recoverable, scalable persistent object store. Through incremental faulting and eviction on a per-object basis, the system can operate on stores much larger than physical memory. PJama does not persist threads — this allows for a simpler checkpointing procedure, as synchronization issues between the checkpointing thread and other mutators do not need to be addressed. However, in order to achieve a consistent checkpoint, all executing threads have to agree that they have currently reached a consistent state by using synchronization on the application level.

The GemStone/JTM Persistent Cache ArchitectureTM [GemStone, 2000] uses modified Java 2 VMs to provide access to a multi-user, transactional store. Its multi-user architecture provides a logical separation between the store and the execution engines operating on it. Hence, persistence of execution state (threads) is not within the scope of this system.

6 Conclusion

We extended the functionality of the Spotless VM with automatic memory management that provides orthogonal persistence, including thread state. Java programs running on the Spotless VM are continuous processes: they can be suspended and then later resumed at exactly the same point of execution as where they left off. They may even be resumed on a different device, since suspended programs can simply be beamed between Palm organizers.

Given the described enhancements to the Spotless VM, it is no longer necessary for the programmer to use a Palm database for making data persistent. Palm databases are only needed to share data among applications.

There is one exceptional situation in which programmers do need to write persistence-related code: to maintain external resources that are not under the control of the Spotless runtime system. For such cases, we provide a callback API to include these resources in the suspension/resumption phases. Leveraging the automatic persistence of threads and synchronization primitive, such call-

backs can be scheduled and executed without compromising data consistency.

Generally, external resources should be dealt with at the library level, such that the *application* programmer is completely unburdened from the task of achieving persistence.

Our main approach is to represent program images directly in stable memory (battery-backed RAM). Because the OS can move memory blocks representing dormant programs around, we build on top of a compacting GC to relocate pointers on program resumption.

Compared with the previous Spotless VM, there is no loss in execution speed, no increase in application footprint and only a small increase in VM footprint (about 4KB or 5%).

7 Acknowledgments

The original Spotless VM was designed and built by Antero Taivalsaari, Bill Bush and Doug Simon.

The exact garbage collection was implemented by Matthew Seidl during his summer internship supervised by Mario Wolczko.

We want to thank Malcolm Atkinson, Mario Wolczko, Mick Jordan and Grzegorz Czajkowski for their many helpful suggestions to improve this document.

References

- [Arsenau and Brault, 1999] Arsenau, E. and Brault, A. (1999). Pocket Smalltalk & Pocket Java. Presentation at 1999 Smalltalk Solutions Conference, <http://www.pocketsmalltalk.com/Solutions99/Final.PPT>.
- [Atkinson and Jordan, 2000] Atkinson, M. and Jordan, M. (2000). A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Sun Labs Technical Report TR-2000-90, Sun Microsystems Laboratories.
- [Gawecki and Wienberg, 1998] Gawecki, A. and Wienberg, A. (1998). Report on the Tycoon-2 Programming Language. Technical Report Version 1.0, Higher-Order GmbH.
- [GemStone, 2000] GemStone (2000). GemStone/J. GemStone Systems, <http://www.gemstone.com>.
- [Haddon and Waite, 1967] Haddon, B. K. and Waite, W. M. (1967). A Compaction Procedure for Variable Length Storage Elements. *Computer Journal*, 10:162–165.
- [Jones and Lins, 1996] Jones, R. and Lins, R. (1996). *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.
- [Jordan and Atkinson, 1999] Jordan, M. and Atkinson, M. (1999). Orthogonal Persistence for the Java™ Platform - Rationale. in preparation.
- [Kornacker, 1995] Kornacker, M. (1995). Persistent Savepoints for Long-Lived Activities in Open Environments (in German). Master Thesis, Fachbereich Informatik, Universität Hamburg.
- [Mathiske et al., 1995] Mathiske, B., Matthes, F., and Schmidt, J. W. (1995). On Migrating Threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*. Also appeared as TR FIDE/95/136, FIDE Technical Report Series, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ.
- [Mathiske et al., 1997] Mathiske, B., Matthes, F., and Schmidt, J. W. (1997). On Migrating Threads. *Journal of Intelligent Information Systems*, 8(2):167–191.
- [Suezawa, 2000] Suezawa, T. (2000). Persistent Execution State of a Java Virtual Machine. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 160–167, San Francisco, CA.
- [Sun, 1999] Sun (1999). The K Virtual Machine (KVM) - A White Paper. Technical report, Sun Microsystems, Inc.
- [Taivalsaari et al., 1999] Taivalsaari, A., Bush, B., and Simon, D. (1999). The Spotless System: Implementing a Java™ System for the Palm Connected Organizer. Sun Labs Technical Report TR-99-73, Sun Microsystems Laboratories.
- [Weikard, 1998] Weikard, M. (1998). Design and Implementation of a Portable Multiprocessor Capable Virtual Machine for a Persistent Object Oriented Programming Language. Master’s thesis, Department of Computing Science, University of Hamburg, Germany. in German.