Memory snapshotting of self-modifying systems

Guido Chari
Universidad de Buenos Aires
charig@gmail.com

Javier Pimás
Universidad Nacional de General Sarmiento
jpimas@ungs.edu.ar

Gerardo Richarte
Disarmista
gera@corest.com

Gabriela Arévalo
CONICET, Universidad Nacional de Quilmes
garevalo@unq.edu.ar

Stéphane Ducasse
INRIA
stephane.ducasse@inria.fr

Abstract

Self describing (pure reflective) and meta-circular systems have many advantages, but in some low-level operations, such as memory snapshotting, self-modification brings some problems.

The problem of *atomic-copy* deals with object immutability and virtual machine state consistency that should be enforced when the same system is used to save itself. Specifically in Smalltalk environments, generating a complete snapshot of the Smalltalk object memory, using the same system that has to be persisted is a challenge since the system changes itself during the saving process.

In the context of our work, the SqueakNOS environment (an OS written in Smalltalk) lacks persistency features because of the complexity of the atomic-copy problem.

In this paper we present a page-based memory manager which can handle native page faults at language level and then a copy-on-write strategy on top of it. Except low level hardware features, all the implementation is written at Smalltalk abstraction level.

Additionally to the advantage of dealing with very low-level operations using high-level languages, we introduce a mechanism that reduces memory usage up to 95%, compared to other approaches on SqueakNOS.

Keywords SqueakNos, operating system, snapshotting, paging, atomicity

1. Introduction

Within object-oriented languages, one of the most relevant features in Smalltalk environments is the *image*, which is the region of memory where all environment objects are stored. Thus, the image snapshotting consists of generating an atomic copy of this memory into a file [Gol84, IKM+97, BDN+09]. When needed, this snapshot can be loaded directly from the file system, restoring the system to the exact same state as it was when saving it.

Snapshotting a self-modifying system on top of another system is an easy task. In standard self-modifying platforms, file writing performed by the operating system primitives. This process helps the implementation of image snapshotting on Smalltalk-like environments because it means that the Smalltalk execution engine[1] is paused while the file is being written by the Operating System. Thus, the objects that are being serialized to a device are not being modified at the same time. Initially, this is a very important property that must be enforced to hold the consistency of the snapshot and is valid in standard environments due to the way they delegate the file copying on the Operating System.

Snapshotting a self-modifying system itself is an interesting problem to be solved. SqueakNOS (Squeak No Operating System) is an approach to an Operating System written mostly in Smalltalk [Squ11b], based on Squeak smalltalk implementation[Squ11a], and its recent fork Pharo. It promotes to *"implement the bare minimum as native code (a mix of assembly and C), and then do everything else at language level."* Thus, it provides its own file writing implementation developed almost completely in Smalltalk. This novel design decision implies that file writing requires the execution engine to be running. But then it poses a challenge for snapshotting the system since it implies that some of its

[1] We talk about execution engine to refer to the engine in charge of executing instructions, be them bytecodes or native instructions if just-in-time compiled

objects, the ones used during file writing, might be changed during the process of file writing. This means that the immutability of the objects in memory (since the user asks for the snapshot until they are actually written to the file) could not be assured.

Hibernation in standard Operating Systems. Hibernation is the action of saving the state of an Operating System so that when the computer is turned on later, it returns to its original state. In open environments, like Linux, this technique is known as suspend to disk and corresponds to the Power Management area [Sws11]. This mechanism serves two goals. Firstly, it saves the current working environment for later use and, secondly it opens new paths on improving system startup times [Kam06]. Conceptually, the problem of snapshotting and suspending to disk (or to RAM) are very similar, but in practice they are very different. Power management techniques interacts deeply with devices, and the biggest challenge is that all resources and its drivers work correctly on restarting. ACPI [Acp] standard is the de facto way of implementing these techniques on nowadays Operating Systems [Gar07]. But unlike the problem we described, such a system saves its own memory with very low-level operations like **memcpy()** that assures that the memory is not modified during the copy.

Atomicity problems. The atomic-copy problem happens when snapshotting self-modifying systems, as in the case of SqueakNOS environment. Other kinds of atomicity problems happen among many other areas of computer science, like transactional operations on databases, transactional memory, semaphore operations on operating systems and more [SGG08, Tan07]. In this paper, we focus only on the problem of attaining atomicity for memory snapshotting in a meta-circular, reflective and self contained environment like SqueakNOS, which was lacking an image snapshotting mechanism due to the reasons described previously. Summarizing, image snapshotting in SqueakNOS is a really interesting problem to work on, since it involves tackling a kind of problem that was not solved yet and also because a substantial solution will really improve SqueakNOS.

Atomic copying through memory paging. We implemented a paging-based memory manager at Smalltalk level, with which page protection faults are resolved completely using high-level language behavior. With this mechanism, we are able to track every attempt to change a memory page before it actually happens. Using this new feature, we developed a copy-on-write algorithm for snapshotting where all the object memory pages are set as read-only before the Smalltalk file writing process starts. From that moment, any attempt to modify a memory page will cause a read-only violation where we can hook, save its content to a buffer and then set it again to read-write. Thus, during a snapshot the system continues working as usual and the file writing process starts to execute eventually. It iteratively sweeps through all pages of the object memory, writing them directly to disk. In the cases where it finds a page that was modified, it takes the content from the buffer. With this strategy we finally reach the goal of a true (consistent) copy and a considerable improvement on memory usage of about 95 percent compared to the naive approach of implementing a complete copy of the memory to a free region using low-level functions (like *memcpy()*) inside the object engine. Additionally, we managed to describe very low-level behavior as page fault interruption handling on language level. This feature is not achieved on any other existing environment.

Thus, the main contributions of the paper are:

- Challenges of memory management in self-described system,
- A lazy object memory snapshotting algorithm based on page-protection mechanism.

The structure of the paper is as follows: Section 2 explains the challenges we faced with on the environment we worked on. Section 3 provides details about the copy-on-write like solution we developed for solving the atomic-copy problem and exposes the results we achieved. Section 4 analyzes some issues related to the design of our approach. Section 5 shows some related work and compare them with our approach, and finally Section 6 concludes and details some future work.

2. Challenges for Memory Management in Reflective Systems

Reflective OSes focus on several key challenges when dealing with memory management. As our approach is based on SqueakNOS, we analyze in detail the different challenges we must cope with it.

2.1 The Snapshotting Challenge

We define the *Snapshotting Challenge* as:

> How to persist an exact copy of a memory region of the system (the object memory [Row01] in our particular case) at a given point in time, when at the same time the saving process must modify it to achieve the desired goal.

Metacircular Operating Systems (the ones written in reflective environments) face *The Snapshotting Challenge* because they cannot rely on the underlying OS primitives to pause the execution (and as such the modification of memory) while the system is saving part of itself. Current general purpose Operating Systems, such as Windows or Linux, do support memory saving through hibernation or similar techniques. However, they are not metacircular, rely on very low level code to solve these problems, and even require special hardware[VV10].

2.2 SqueakNOS: an Example of the Problem

SqueakNOS: a metacircular OS. SqueakNOS implements Operating Systems features in a pure object-oriented environment. It deals with hardware resources management, bringing an interface that application level software uses to interact with devices. Unlike standard Operating Systems, this interface is implemented in a highly dynamic way, due to inherent capabilities of Smalltalk. This means for example that all stages of file saving are managed using Smalltalk code: from the bare ATA [ATA11] drives protocol, through FAT32 [FAT11] filesystem implementation and up to file streams wrapping.

Snapshotting challenges in SqueakNOS. The process of writing data into files is different in SqueakNOS compared to standard environments because it involves executing Smalltalk methods until the actual level of transferring the bytes to the device. This is a feature inherent to the environment, and strongly affects the image snapshotting mechanism because it means that in SqueakNOS *pausing the execution engine for file writing is not possible*. The behavior for that task is defined at language level, so *the object engine must be executing Smalltalk methods* to deal with file writing. But the execution of Smalltalk methods implies that at the same moment the object memory is being copied, its objects might be being mutated by the execution of the process in charge of the snapshotting (and others too). Summarizing, neither the atomicity could be assured nor the consistent behavior of snapshotted images. Figure 1 shows the difference between the two approaches.

2.3 Zooming on the challenge.

Let's illustrate with an example what would happen if the object engine was not paused while snapshotting. Suppose there is a linked list object which has an instance variable size that represents the amount of objects it contains, and first, a reference to the first item. There is an invariant that size should always be the same than the amount of items in the list (which could also be calculated by following the chain starting in first and ending on a nil object). This invariant should be valid any time, except when an item is being added or removed, because then the internal state is being modified. Now let's suppose that a snapshot is made while the object engine is adding an object to the list[2]. Clearly, the list may be saved in a state where the invariant is not valid. This process would not represent a problem as long as the process that is modifying the list is also saved, in the correct state. So that when the snapshotted image is loaded and restarts its execution, it finishes adding the corresponding object and so the invariant is restored. But in case that snapshotting is not atomic, there is a possible race condition. The list could be written in an inconsistent state and after that, the adding process could be scheduled and run, restoring the invariant in the running image. But then, when the saving process writes the adding process to disk, the invariant has already been restored in the running image, but not in the written one. This leaves an inconsistent list in the snapshotted image that will not be restored to a valid state.

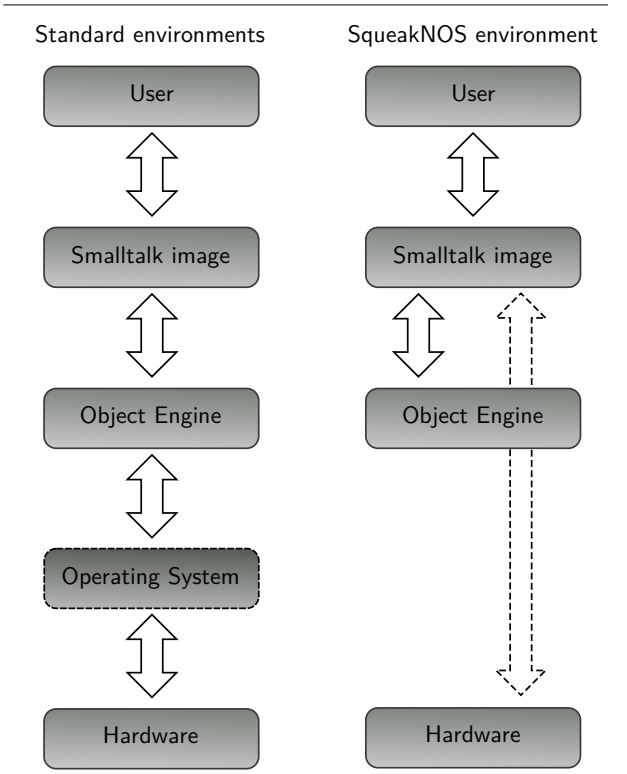

Figure 1. Comparison of standard and metacircular environments.

The problem could be mitigated in the case of multithreading, by allowing only the execution of the file writing process, but that solution will not solve the problem in presence of reentrancy. Due to this, if the execution engine is not paused during snapshotting, the snapshotter will have to deal with objects that change while writing to the file. It will have to determine which objects that changed shall be written after being modified and which should be written before. For the ones that should be written before, a copy of their original state will need to be kept until they are effectively written.

Summarizing. The snapshotting problem we faced on SqueakNOS (it should be the same for any self contained system) is similar to some known atomicity problems, and we defined this particular case as the atomic-copy one.

[2] Take into account that this can only happen with the presence of multithreading or reentrancy

Atomicity problems are very well known in databases and operating systems. In databases, it is important for a transaction to be executed completely as if it was done isolated, or not to be executed at all, in order to avoid inconsistent data to flow or persist. On the other hand, in operating systems, semaphore implementations require atomic operations to guarantee that two processes never enter a critical section at the same time. In our context, atomicity is related to the fact that the snapshot must be an accurate and consistent copy of the object memory at the moment the user asks for it. To provide the required atomicity, standard Smalltalk virtual machines pause bytecode execution throughout the snapshotting process. But that design is not possible nor desired on SqueakNOS. Then, the only possible solution is to simulate atomicity in a controlled way to leave a consistent snapshot of the memory in a file.

Solution in a nutshell. Under this assumption, we considered different approaches. Firstly we considered to modify the snapshot primitive of the SqueakNOS object engine to directly support the tracking of objects that change while being written, but that is a very complex alternative because it requires extensive modifications to other parts of the object engine. So we discarded that solution in favour of a simpler one, that is to wrap around the saving process in a way that it looks as if the execution engine was stopped during the process. To fulfill this goal, we decided to develop a paging-based memory manager almost completely in Smalltalk and set all the memory pages as read-only when file writing starts. When a protection fault interrupts the snapshotting process, we deduce that an object is being modified by it and we copy that page into a buffer before setting the page as writeable again. Then, the page is copied to the file from the buffer instead of from the original object memory assuring the immutability. This mechanism is well known as the copy-on-write strategy. We also developed a naive low-level approach to provide interesting comparisons, that before writing to files, mirrors all the object memory with low-level operations and then writes the mirrored memory space to the file instead of the object memory.

3. Lazy protection fault-based memory snapshot

3.1 Snapshot mechanism

To understand the main approach implemented to solve the atomicity problem exposed, we need to present firstly a complete background of how the process of image snapshotting is performed in a standard Smalltalk object engine. Actually, we will explain the Squeak/Pharo standard virtual machine implementation, because that is the basis object engine used by SqueakNOS.

The image saving mechanism is implemented in a primitive called primitiveSnapshot (the essential component for image persistence). In most object engines, when primitives are running the normal flow of execution gets paused (bytecode interpretation) and this is a very important property. This primitive for image snapshotting has two main steps:

1. Prepare the object memory for serialization to assure that the copy will be consistent for loading it later. This means roughly to finalize some special objects that need special considerations after collecting the garbage.
2. Calling a platform specific C function (sqMemoryFileWrite) that writes all the object memory to a file. The internals of file writing are implemented by the operating system.

3.2 SqueakNOS limitations for standard snapshotting

In SqueakNOS, the only way to write a file is by sending a message to the corresponding filesystem object in the same environment. There is no underlying Operating System primitive that can be called from C for writing to files. The file system object communicates with the hard drive controller object, which in turn sends messages to the hard disk one. This chain ends when the hard disk object configures the actual hard drive, by issuing generic in/out assembly instructions, which are implemented as primitives.

The function sqMemoryFileWrite had been implemented, on its first versions, as an empty function meaning that any time that the primitive snapshot was called, the second step of the process explained above would just fail. This means that the fault would not be noticed by the user until he restarts the system and realizes that the image was not saved.

One of the causes of this problem is that in Smalltalk even the internal threads and the process scheduler are first class objects. Clearly, a reflective and self described environment would have lot of this reentrancy problems to surpass, and this is a very evident one. Thus, the conclusion is that the limitation of SqueakNOS for this task is inherent to its definition and this process could not be accomplished completely atomically in this kind of environments.

3.3 Snapshotting with copy-on-write over memory page write protection

We introduce a solution to this limitation that keeps track of which objects are modified while writing the snapshot to a file. An object tracking process is setup before starting the file writing and works on background. Each time the file writing process tries to modify an object, the hardware emits an interruption that activates the object tracking handling process. This process, before allowing the modification, copies the original state of the complete memory page into a buffer. Therefore, when writing each object to disk, it can be taken directly from the object heap, unless it has been modified. In this latter situation, it would require to grab the saved copy of the original state from the buffer. This is almost the optimal theoretical space usage solution to the atomic-copy problem. There is a trade-off between space and time optimality. We work with the granularity of mem-

ory pages instead of singular objects, so we resign a little on space efficiency in favour of time and simplicity.[3]

Fundamentals of paging mechanism. To track which objects are modified while writing the snapshot to disk, this implementation uses a memory paging mechanism. For better understanding of the general idea we first provide a brief review of this mechanism. Many processor architectures, such as x86, implement hardware paging support. In this scenario, memory is seen as an array of pages, where each one has a fixed size. This serves many purposes, including virtual memory and protection. In each access to memory, the processor translates the pointed address by means of a page table entry. The table entry indicates the physical address of the virtual address, and also contains a set of flags related to that page. One of these flags tells if the page can be written. Figure 2 depicts this mechanism. The hardware catches any attempt to write a page flagged as write-disabled and generates a processor interrupt [NL06]. By handling this interrupt, along with the write flag, the paging mechanism can be used for implementing paging-based copy-on-write of memory pages.

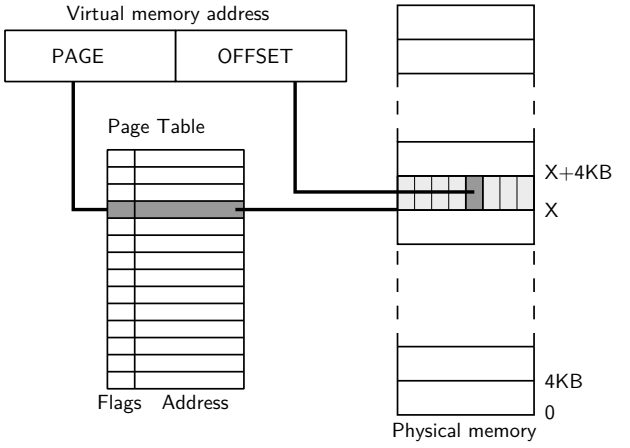

Figure 2. Virtual to physical address mapping.

Our snapshotting implementation. The implementation of *copy-on-write* snapshotting relies on the idea of copying the image memory in a lazy way. Following we explain in detail how this mechanism works

1. The set of pages where the object memory is hosted is marked as read-only by setting the write bit of each page as disabled, just after it is prepared for serialization.
2. The interrupt handling routine for write-protection page faults is set to a strategy that is in charge of copying the to-be-modified page in a buffer and finally clears the write flag to allow its modification.
3. Execution continues normally and eventually the snapshotting process will take its turn and write the object heap to a file. It should check for every page of memory to be copied, if it was already modified or not. In case it was not, the snapshotting process directly takes the page from memory and writes it to the file. But if the page had been modified, then its original content was stored in the pages buffer by the interrupt handling routine, so the snapshotting process can take it from there and write it to the file.

This guarantees that the heap will not be modified during the file write or that, if a page was changed, a copy of its original state would be kept in a buffer. The main advantage of this mechanism is that it will only make a copy of the pages whose objects changed throughout the process, which are less than all the ones stored in the object heap.

It is worth noting that this behaviour is mostly defined at language level. The only place where native code is used is in the initial handling of the interrupt. This is because the hardware requires to setup the address of a native routine to call when the interrupt occurs. But this native code does not resolve the page fault itself. Instead, it only saves the execution context and reenters the Smalltalk object engine activating a handling block, which was configured at startup time by the Smalltalk memory manager.

Figure 3 shows the Smalltalk methods required to handle a page fault, when the page accessed was marked as not present. In this case, the page directory and table are accessed to find the corresponding page entry, and the page is just marked as present. In the case of copy-on-write handling this method is slightly different. This code can be debugged with the standard Smalltalk debugger, and even a *halt* can be set on the method that is handling a very restrictive operation such as a processor interruption caused by a page fault. Thanks to this features, it is posible to dynamically inspect the objects that represent the memory and processor and also to change protection-fault handling code on-the-fly.

4. Discussion

4.1 Implementation of a full copy approach

A simple solution. As a first approach we designed a simple solution which consists on making a full and atomic copy of the object memory to a free section of memory. The atomicity is assured by the use of low-level memory copy functions (memcpy()) inside a primitive. The file writing process uses this pristine copy instead of the actual object memory, which will be changing in the mean time. The copy can be accomplished with a few lines of C code, but is highly inefficient in terms of memory usage because it needs a free space of the same size of the object memory being copied. This will not pose a problem on images of a few MBs and systems with many GBs of available memory, but will be an obstacle when running SqueakNOS on embedded systems

[3] It can be argued that if all objects in a single page are modified, there is no resigned space efficiency, but in practice that is highly unprovable.

```
Paging >> pageFaultHandler
    ^[:args :result | self resolvePageFaultOn: args virtualFaultAddress]

Paging >> resolvePageFaultOn: anAddress
    | pageDirectoryEntry pageTableEntry |
    self halt.
    pageDirectoryEntry := self pageDirectoryEntryFor: anAddress.
    pageDirectoryEntry isPresentAndAllowed ifFalse: [^pageDirectoryEntry setPresentAndSupervisor].
    pageTableEntry := self pageTableEntryFor: anAddress.
    pageTableEntry isPresentAndAllowed ifFalse: [^pageTableEntry setPresentAndSupervisor].
    self error: 'FAIL'
```

Figure 3. The Smalltalk behavior for handling page faults.

with little physical memory, or with images sized up to many GBs.

Discovering the contents of system memory. The main development problem was to find a suitable area of memory in which the frozen copy of the object heap can be stored, and after set up the object engine to make the copy in the right moment. Finding a free chunk of memory big enough to hold the complete copy of the image required a deep analysis of the code of the object engine, to determine all the areas of memory that were already in use. The result of this analysis is shown on Figure 4, which depicts the areas occupied by the BIOS, the bootloader, the object engine and the object heap. The BIOS, bootloader and the object engine all use a small area at low addresses, as they consist of compiled assembly and C code. There is also a free zone for the C call stack, used by the object engine. Lastly, the bootloader loads the image file as a kernel module at an arbitrary position, just a bit higher that the one of the kernel[4], and passes its address to the kernel at start time. The amount of space reserved initially for the object heap is the size of the image file plus an extra amount that is used when allocating new objects. This space will not move its base address, but can grow as much as needed to allow allocating more objects if the original space is exhausted.

Establishing the position of the copy. Because of the variable size of the object heap, it is very important that it is located after the object engine and not before. The biggest block of free memory is the one that goes from the end of this heap to the end of the available memory. We chose this block as the one to contain the frozen copy of the memory to be written. In this block, we chose to write the copy in the highest possible memory addresses to reduce the chances that the object heap would run into the buffer if it grows while writing to the file.

[4] It is worth noting that here kernel means the object engine, but for maintaining the Operating System terminology we called it kernel.

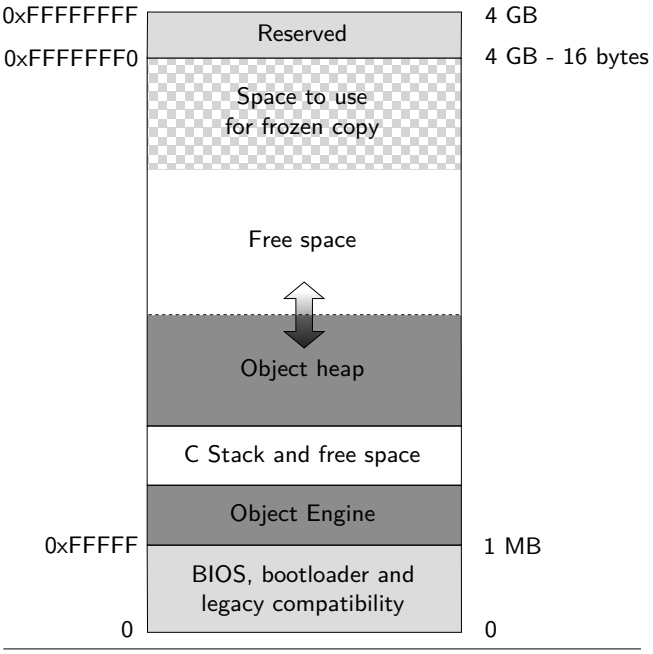

Figure 4. SqueakNOS memory organization.

4.2 Implementation of the paging-based method

Marking the whole object memory as read-only involves major problems. To understand the reasons, it is important to remember SqueakNOS philosophy, which implies doing as much as possible at language level, and then to consider some kinds of special objects that are stored in the heap.

Marking pages atomically. If the algorithm that marks pages as read-only were written in Smalltalk, then it would raise the atomic-copy problem. The sole execution of the Smalltalk code that marks pages would cause changes in the object memory, getting the snapshot corrupt. To avoid this problem, the page-marking phase is done by a primitive, which does not require execution of Smalltalk code. This was chosen as a trade-off, as the required code for marking pages consists of only a few lines. On the other hand, after

writing pages to disk, or copying them to the temporary buffer, pages have to be set as read-write again, and that time marking is done with Smalltalk code, as there is no atomicity limitation at that moment.

Recursive protection-faults problem. The implementation of all this mechanism using Smalltalk objects would mean, among other things, that protection faults should be resolved with Smalltalk code. But in this approach, all the process would need to be carried out with the entire object heap set as read-only. Doing this raises a similar problem to the atomicity that was described previously: the execution of a method while memory is read-only -even during the brief process of handling an interrupt- might cause a change in some object of the heap, changing the contents of memory even before the interrupt is handled. Unlike with atomicity, under these circumstances the execution of methods does not corrupt the image, but it generates a problem of *infinite recursion*. The handling of a protection fault will cause a new protection fault before it is handled, which raises another one and continues that way recursively.

The first experiments with this technique for validating the theory showed that this problem appears and occur frequently. We analyzed it by reverse engineering techniques because the code to test was very deep inside the execution engine. We had to analyze heavily the context in which protection faults occurred, and try to discover which objects were being written and causing infinite recursion. The goal was to find out which processes cause the modifications on the memory. This analysis was really hard, but it led us to a thorough understanding of the very primitive object engine inner workings.

Surpassing infinite recursion. We discovered that the first read-only violations were caused by the process scheduler while attempting to reschedule the page handling process as the active one, and afterwards when activating other processes to handle hardware interrupts for devices like mouse and keyboards. The solution to this problem was to copy the pages where these objects are stored, setting them as read-write before the first activation of the page handling method. Then we found out that the execution of primitives caused other recursive read-only violations, which was related to an optimization of the look up of the native code. Therefore, that optimization had to be disabled while a read-only violation is being handled. Summarizing, the object engine has knowledge about some special objects as the scheduler, some semaphores and some optimizations strategies. These special objects are being modified by the object engine for doing any task, so the best solution we found was to pre-save manually these memory pages and leave them as read-write to avoid the infinite recursion issue.

Native read-only violation handler. After all, we realized that not all the causes of recursive protection faults could be managed. The object engine is a very complete, big and complex software to prove an execution property as this one. So realizing that there was no warranty (formal proof) that a recursive read-only violation would not be eventually raised, we chose to implement a second chance protection fault handler: if a read-only violation is raised while another one is already being handled, then the second one is handled by native C code avoiding the infinite recursion again. This was considered as a trade-off between the main objective -implement everything in Smalltalk- and the problems we would have needed to solve in order to implement it in a pure way. The native handler only has a reduced functionality, which is the minimal required to use it as a fallback, and is not dynamic as changing it requires recompiling the whole object engine and rebooting the system. For that reason it is better to keep the Smalltalk one handle most cases, and resort to the native one only when using Smalltalk is not an option.

Garbage collection issues. SqueakNOS relies on a generational garbage collector[Ung84] to free space. During the execution of the snapshotting code, the execution engine can fall out of space and interrupt the running process to collect garbage. This does not cause corruption in the resulting snapshot, because of the hardware-based copy-on-write mechanism, but can cause other problems. Garbage collector moves surviving objects, which could cause a massive amount read-only violations. In the worst case all pages would be mutated by the collector, making our method very inefficient memory-wise. But before the snapshot is started a full collection is done, compacting the heap and cleaning the new space. The only possible collection that is likely to happen is a generational one, but that only affects newly created objects, which are not written in the snapshot. On the other hand, execution of Smalltalk interrupt handlers is not possible during GC because of the limitation of the object engine, and for that reason read-only violations raised during garbage collection are handled with the native handler mentioned above.

4.3 Validation

We conducted a series of experiments to measure the amount of read-only violations occurring throughout the entire process of writing an image file. Using a standard Pharo image file with SqueakNOS modules loaded, with a size varying between 30 and 40 MB, the amount of protection faults handled during the snapshot is close to 100. With pages of size 4KB, the amount of space used to write the snapshot is around 400KB. This means that our technique uses less than 2 percent of the total image size, or that it improves the complete memory copy approach by 98 percent.

On the other hand, experience showed that movement of objects caused by garbage collection is not an issue. Full garbage collection was never raised during file writing in the tests, and incremental one does not cause big mutations of the old space, which is the area of memory to be written.

The implementation proved to be robust and was added to SqueakNOS as the default image saving mechanism.

4.4 Other possible approaches to the snapshotting challenge

There are other possible alternative solutions to the problem presented in this paper. Nevertheless, these solutions were discarded mainly because of the complexity of their implementation. One of them consists of tracking objects modification in a different way, and other can be designed to avoid modifying the heap at all while writing to disk.

Per-object modification tracking. Tracking of modified objects can be implemented in multiple ways. One alternative would be to modify the object engine so that it would make a copy of objects when writing to its instance variables. This mechanism would be a per object tracking and would require major modifications to the object engine.

Using a temporary object heap or a second object engine. Instead of tracking which objects change, the object engine could be modified to use a temporary object heap during the writing process. This would involve making the object engine be able to switch its context to the temporary heap, and also require a complex analysis of which objects need to be bootstrapped into it. Another similar path would be to have a second virtual machine entirely loaded in background, to be used when the first one has to be saved.

5. Related work

The most related technique to the main problem solved in our work is what is known as *suspend-to-disk* on standard Operating Systems. The problem faced with this technique, is the one of snapshotting a complete Operating System environment, so it is possible to start the computer on that particular state later. There isn't too much formal documentation on this technique, but Linux documentation exposes it situated inside what is more generally called *Power Management*. According to the documentation, what is more challenging in this area is the interaction and compatibility of power transition states with different kind of heterogeneous devices[VSSI04]. At first, APM was the mainstream model for managing the power of devices. Nowadays, ACPI provides better abstractions so it allows more behavior to be modelled on software in contrast to BIOS-based interactions as it is with APM. But although the Operating Systems have to copy their own memory to disk, it is not a big challenge since it works in a completely different level of reflectiveness than SqueakNOS. Essentially, the model of execution is different, and so executing some special tasks (as a memcpy() operation) does not mean necessarily a modification of its environment. That is why although the problem is by definition the same, the obstacles and challenges are completely different and we didn't benefit too much on analyzing that problem.

The design of Operating Systems using *high-level* programming languages is a limited field. Here we mention some approaches that are close to the main idea behind SqueakNOS. JNode (Java OS) [JNo11] (Java New Operating System Design Effort) is a free software project to create a Java platform operating system. The goal of the project was to create all the software in Java itself, with the exception of a *microkernel* developed on assembly language to boot and load the system. The *JVM* compiler (which normally uses just-in-time compilation) is used to build native binaries out of the core Java code. Thus, nearly the entire system is written in *Java*. This project focuses on the security, stability and robustness that the *Java* environment should provide to the system [lsa].

JX [JX11] is a *Java* Operating System that focuses on the flexibility and robustness of the *JVM* too. The JX system architecture consists of a set of Java components executing on the core that is responsible for system initialization, CPU context switching and low-level domain management. The *Java* code is organized in components which are loaded into domains, then verified and translated to native code. The domains are protection units that mainly isolate objects. Each domain has its own *heap* and *garbage collector*. The domains are presented in a hierarchical way where the lowest one contains the *microkernel* [GFWK02]. Same as JNode project, it does not support *virtual memory* and the memory is managed the same way as the *JVM* does. This means concretely that both projects carry on a sustantial difference with SqueakNOS and then that they don't face with reflective snapshotting. In addition that programming environments have some important differences, Smalltalk is the only that introduces the concept of an image. In *Java* based *operating systems*, a precompiled program should be taken by the loader. So, working dynamically in the image, interacting with the hardware and persisting the changes, all in the same environment is not possible.

House [Hou11] is a small operating system coded almost entirely in *Haskell*. It builds on the previous *hOp* project. The system includes device drivers, a simple window system, a network protocol stack, and a command shell window in which a.out files can be loaded (via TFTP) and executed in user-mode. This is the first functional-language operating system that supports execution of arbitrary user binaries (not just programs written in the functional language itself). This project focuses on achieving a monadic interface to interact with devices [HJLT05]. It then exploits functional programming attributes to demonstrate interesting properties of the resulting software and its security. Here, the functional paradigm is a different way of modelling problems so the difference with SqueakNOS is conceptual and can not be directly compared. However, *House* is the only project that presents a *memory manager* implementation that supports *virtual memory*. However, it is clear that the difference in

paradigm here makes that a comparison on a reflective snapshotting mechanism is a completely nonsense task.

In addition to Operating System projects, there is a related *atomicity* problem, solved with *virtual memory* technique, already explored. A real-time, concurrent copying collector that use the *virtual memory* hardware to implement synchronization between the collector and the different mutators (*threads* that modify objects) [App04]. This project simulates *atomicity* when collecting while mutators are modifying objects. SqueakNOS should be an ideal environment to implement this kind of algorithms.

6. Conclusion

In this paper, we achieve the goal of implementing the object heap snapshotting with the tools that were already present and changing as little as possible the SqueakNOS foundations. There is a straight and naive approach that we implemented mainly to contrast with the main solution. This approach achieves a complete atomic copy of the object heap in memory for then writing it into a file executing Smalltalk code, thus, the object engine. It is a solution that is transparent to the user, meaning that we only made a semantic change on a *C* function that instead of writing into a file, it now does a memory copy. This means no significant changes to the object engine were done, thus also not meaningful low-level add-ons.

But this naive solution has an important shortcoming of being very memory consuming. It does not seem to be efficient that for writing n bytes from memory to disk, another n free bytes are needed on memory to complete the process. So we proposed another approach. We presented an implementation that, instead of making a copy of the object heap in memory, sets the *pages* of the *page table* that contains the object heap as read-only. Then we setup a mechanism, completely in Smalltalk, to handle every protection *fault* generated on that range of *pages*. This handler makes a copy on memory of the faulted *page* before allowing to write on it. So, the object heap is copied from memory directly with the only exception that if a copy of the original page is found in a buffer, that copy is written instead of the original one. This new mechanism presents improvements of more than 98 percent of memory usage with the tests we have done compared with the first case. In the worst case, every page is copied and the result is the same as the other approach, however, this weird case should be very infrequent and on very special and bounded cases.

Summarizing, we presented a solution to a very important milestone of SqueakNOS, which now could be persisted on its usage. Also we proved the advantage of having low-level hardware modelled at high-level languages, by introducing a solution that does heavy use of memory management techniques for a general software problem. Furthermore, we presented a solution to a new problem that arises with reflective Operating Systems, we define lot of terminology and clearly exposed the problems and challenges for this kind of artifacts. Regarding the results achieved, it is clear that the algorithm presented would use only a little kilobytes of extra memory for copying a considerable big object heap in contrast to a naive and straight solution that needs the same amount of extra memory as the object heap.

Finally, and although not thought as a central contribution, this work summarized some of the *Squeak / Pharo* object engine architecture and how SqueakNOS links with it. We have also shown that *high-level* programming languages are useful to work on Operating Systems development. However, there are some *low-level areas* that could not be tackled yet, such as for example processor architecture assemblers to run anything that needs to interact with devices. These border cases are isolated by implementing them in primitives. Beyond these primitives, SqueakNOS shows that Smalltalk is powerful enough to work on this area. There is few documentation available about Squeak / Pharo object engine, and for that reason we had to do an almost complete reverse engineering of it. Specially we had to understand the special cases where the read-only violations handling could fall into infinite recursion. We had to research from the most simple one, as it is the code for snapshotting, to more complex tasks as garbage collection properties and plugins architecture.

6.1 Future work

There are still areas where SqueakNOS could have extensions and / or improvements.

Regarding image snapshotting, the main goal is to get rid of files (the implemented *filesystem* models should serve to communicate with other systems files) and to have a transparent persistency, similar to the one *Gemstone*[BOS91] offers. The *FAT32* implementation needs some performance improvements too. Other newer and better filesystems as *Ext2*, *Ext3* or *NTFS* should be implemented too. To have transparent persistence as *Gemstone*, virtual memory support is mandatory. The actual version of SqueakNOS has some support on this, but it is basic, lacking offloading of pages to hard disk.

Another idea that would solve the snapshotting persistence problem and we had not explored, is to have two actors, or the same, two object engines. That means that one of them could be the responsible for executing the user bytecodes and the other should be the chosen to execute all control or administrative tasks, as snapshotting. This should have the result of an atomic copy with not even one byte of extra memory. But, on the other hand, it would require a fixed amount of memory for the actor binary code and several changes to the original object engine for synchronization mechanisms.

References

[Acp] Acpi. The ACPI specification.

[App04] Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Not.*, 39:205–216, April 2004.

[ATA11] Ata, 2011. http://www.t13.org.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The gemstone object database management system. *Commun. ACM*, 34:64–77, October 1991.

[FAT11] "microsoft efi fat32 file system specification", 2011. http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx.

[Gar07] Matthew Garrett. How linux suspend and resume works in the acpi age. *Blog http://www.advogato.org/article/913.html*, 2007.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The jx operating system. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.

[Gol84] Adele Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. *SIGPLAN Not.*, 40:116–128, September 2005.

[Hou11] House: Haskell user's operating system and environment, 2011. http://programatica.cs.pdx.edu/House/.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[JNo11] Jnode: a new java operating system, 2011. http://jnode.sourceforge.net/index.html.

[JX11] Jx: The fast and flexible java os, 2011. http://www.jxos.org/.

[Kam06] Hiroki Kaminaga. Improving linux startup time using software resume (and other techniques). In *Proceedings of the Linux Symposium Volume II*, 2006.

[lsa] Jnode creator blog. http://lsantha.blogspot.com/.

[NL06] Linda Null and Julia Lobur. *The Essentials of Computer Organization And Architecture*. Jones and Bartlett Publishers, Inc., USA, 2006.

[Row01] Tim Rowledge. *A Tour of the Squeak Object Engine. In M. Guzdial and K. Rose, editors, Squeak: Open Personal Computing and Multimedia*. Prentice Hall, 2001.

[SGG08] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[Squ11a] Squeak, 2011. http://www.squeak.org.

[Squ11b] Squeaknos, 2011. http://www.squeak.org/squeak/1762.

[Sws11] Swsusp, 2011. http://www.mjmwired.net/kernel/Documentation/powe

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9:157–167, April 1984.

[VSSI04] Srivatsa Vaddagiri, Anand K. Santhanam, Vijay Sukthankar, and Murali Iyer. Power management in linux-based systems. *Linux Journal. http://www.linuxjournal.com/article/6699?page=0,0*, March, 2004.

[VV10] Mudit Vats and Ishu Verma. Linux power management, iegd considerations. Technical report, Intel Corporation, March, 2010.