Linux Kernel extensions to minimize effects of Software Aging

Ariel Sabiguero, Andrés Aguirre, Fabricio González, Daniel Pedraja, and Agustín Van Rompaey

Instituto de Computación, Facultad de Ingeniería, Universidad de la República
J. Herrera y Reissig 565, Montevideo, Uruguay

{asabigue|aaguirre}@fing.edu.uy {fabgonz|danigpc|fenix.uy}@gmail.com

http://www.fing.edu.uy/inco

Abstract. State of the art software rejuvenation techniques tackle different levels of transient system errors in a coarse-grained manner: full system or full process reinitialization. Our team implemented a prototype that addresses a finer-grained approach. The implemented prototype aims at the detection and correction of corruption in read only memory frames of a GNU/Linux system.

This work describes the class of soft errors addressed, error correction approach and its application to system reliability and availability. To evaluate the solution implemented by the prototype we performed extensive studies on different applications under simulated load. A fault injector was used to generate and simulate errors on the system. Implications to performance and enhanced availability are presented too, showing the applicability of the technique is adequate under general conditions.

Keywords: rejuvenation, soft errors, memory correction techniques

1 Introduction

Memory-leaks, unreleased locks, shared-memory pool latching, non-terminated threads, storage fragmentation, data corruption, cosmic rays and thermal dissipation failures are some of the reasons why running software images corrupt as execution time grows. This phenomenon is known as software aging, and has been observed in enterprise clusters, telecommunication systems, web servers, home PC and other systems. Evidence of these effects is most common on always-on systems, like web servers, e-mail servers and so on.

A naive and common technique to fight software image degradation is periodic reboot of affected systems. This technique is sometimes referred as software rejuvenation. Despite its sexy name, the technique brings to the data-center the grandmother’s recipe of powering-off and then powering-on the stumbling device. Different software rejuvenation techniques put some order into rebooting chaos and gives us the sensation that we are fighting software aging technologically.
This work analyzes and isolates specific sources of software image degradation and its effect on running software. A deep analysis of operating system memory management techniques allows us to propose a software independent, general purpose fine-grained software rejuvenation technique that does not require operating system reboot. A prototype was implemented and benchmarked to gather experimental data.

This work is organized as follows. Section 2 provides an introduction and analysis of software aging. Section 3 provides an introduction to memory management on modern OS, focused on read-only memory, and corruption detection. Afterward, Section 4 describes details of the prototype implemented. Experimental results are presented too. Different lines of work and perspectives to the field are suggested in Section 5 and the work is concluded in Section 6.

2 Software Aging

Not all flaws in our systems are due to faulty software. In fact we could have perfect software running on a system, and the system could fail due to external causes. There is a big chance that, in a big subset of the cases external causes affect the hardware and by consequence affect the software running on it. There is also a chance that transient hardware or software conditions affect the expected execution. The rest of this section introduces relevant concepts related to software aging.

2.1 Taxonomy

Jim Gray [1] classifies software errors in two different categories, Bohrbugs and Heisenbugs. This distinction is based on the ease of reproduction of the fault caused by these errors. The first category, the Bohrbugs, which are named after Bohr’s atomic model, are essentially permanent design errors and are by nature almost deterministic. They can be easily identified and corrected during the system testing phase in the software’s life cycle. On the other hand, the Heisenbugs, named after Heisenberg’s uncertainty principle, include those internal faults that are intermittent. Faults whose activation conditions occur rarely and thus, are hard to reproduce. Incorrect or incomplete exception handling mechanisms are a clear example of Heisenbugs. These errors are much harder to detect during normal system testing than Bohrbugs, due to their non-deterministic nature.

K. Vaidyanathan and Kishor S. Trivedi [2] add a third category to this classification, which includes faults caused by the occurrence of software aging. This category of faults is similar in some way to Heisenbugs, in which their activation is triggered by certain conditions such as the lack of operating system resources, something that is not easily reproduced. However, their recovery methods differ significantly. While Heisenbugs are tackled by purely reactive techniques, faults caused by software aging can be prevented by the application of proactive techniques, such as periodic restarts of the affected software.
We are particularly interested in hardware errors, especially those occurring in the memory chips of a system. These errors are known as soft errors [3], and can be classified as software aging related errors, even though they are not caused by the aging itself but by external causes, such as cosmic rays, temperature, humidity, among others.

2.2 Soft Errors

As introduced in Section 2.1, soft errors are errors in the memory hardware of a system. It is important to focus on them because they are not normally taken into account when software is designed, and can eventually cause full systems to completely crash. It would be interesting to have some kind of protection against these errors that is not built into the hardware itself, like error correcting code memory chips (ECC). Due to the nature of these errors, it is impossible to predict when they will occur, and so we need a reactive technique that allows our software to gracefully recover from them.

Soft errors can be caused by a variety of events. One of these events is the emission of alpha particles. This was particularly notorious after the introduction of Dynamic RAM (DRAM) in the seventies. These new chips used small amounts of radioactive material that emitted alpha particles, which could disrupt the electron distribution in semiconductors, causing the digital signal to switch from zero to one or vice versa.

After the memory manufacturing industry found ways to control the emission of alpha particles, and noticing that soft errors were still happening, it became clear that errors were being caused by something else. James F. Ziegler, an IBM researcher published a number of papers proving that cosmic rays could be another cause of soft errors [4]. During these experiments, IBM discovered that the frequency of soft errors increased with height, doubling itself at 800 meters above sea level. In Denver, which is located 1600 meters above sea level, the frequency of apparition of soft errors was 10 times that of sea level. In another experiment, they discovered that systems located underground were experimenting a much lower rate of soft errors, and since the 20 meters of rock can block almost all cosmic rays, they concluded that cosmic rays were indeed one possible cause of soft error occurrence.

Another possible cause of soft errors is electromagnetic interference (EMI). Said interference appears when an object generates a magnetic field which interacts with the electric charge of integrated circuits. This interaction can cause alterations in the transistors or buses, which in time can cause their logic value to change. This kind of interference is most common in places where high frequency electric motors operate [5]. In industrial environments, where this kind of motors are used and many embedded systems control machinery, the probability of a soft error happening is much higher, which can lead to machinery damage or even worse, human operator injuries. These factors are crucial when considering safety aspects.
2.3 Existing solutions and techniques

The most common techniques to deal with errors caused by software aging are known as software rejuvenation. There are two different strategies that can be used to determine the appropriate time to apply rejuvenation. The first one is known as the Open-loop approach [2]. This method consists of applying the rejuvenation process without using any kind of information about the system’s performance. In this case, rejuvenation can be applied after a certain amount of time has passed since the last application of this method, or when the number of concurrent jobs running in the system reaches a defined number.

On the other hand, we have the Closed-loop approach [2]. In this method, the system’s health is continually monitored to detect the possibility of an aging-related error happening, such as the lack of a particular system resource that could lead to a performance hit or even a complete system crash. Within the closed-loop approach, we can further classify our technique by looking at the way our collected data is analyzed. Offline analysis techniques use system performance data collected during an extended period of time, typically weeks or months to determine the optimal frequency of the solution’s application. This kind of analysis is adequate for systems whose behavior is deterministic. Online analysis techniques are based on data sets collected while the system is running, which combined with previous sets determine if it is a good time to apply rejuvenation. Since the scope of this strategy is broader, it is useful for systems whose behavior is difficult to predict. In this case, the future behavior of the system is determined by the current data and the history of sets previously collected and analyzed.

Software rejuvenation techniques can be applied at different granularity levels. An example of rejuvenation applied at system level is a complete hardware reboot. On application level, restarting the affected application or processes is a good example of rejuvenation. Applying rejuvenation clearly has a negative impact on a single system or application availability. For this reason, it is useful to have a clustered architecture, where we can apply rejuvenation techniques to individual nodes without severely crippling our system. In addition, rejuvenation can be applied on non-peak hours, further reducing the system’s down time. If we design our systems to take these factors into account, applying rejuvenation has proven a proactive technique which is very effective in preventing errors with little cost [6]. With this approach we are not enhancing system reliability, but service availability.

3 Addressing soft errors in R/O memory

In Section 2 we studied software aging and different classes of errors that can cause it. We paid special attention to a particular class originated in hardware and called soft errors. These kind of errors are popular in memory hardware and existent solutions to detect and fight their occurrence are based on hardware implemented correction techniques. In this Section we explain how software im-
implemented change detection over read-only memory can be a cheaper alternative
to finding and correcting these errors.

Modern operating systems and hardware architectures provide a protection
scheme over available memory. These schemes essentially sign read/write/execute
access rights to each one of the memory frames into which the physical memory
address space is divided into. This frame division and access rights are directly
supported by most architectures thanks to processor extensions associated to
virtual memory handling instructions. In this work we use the term read-only
memory to refer to those parts of a system’s RAM that the OS protection scheme
denies write access. Because of OS implementation of virtual memory techniques
to give the user space processes the idea of having the entire linear address space
to themselves, a given frame may be shared by several tasks and its access rights
may vary during system execution. For now, let’s keep things simple and just
focus on the fact that it is possible, at any given time, to classify memory regions
in two main categories: read-only and read-writable. Each process is composed
of some writable frames, that can be modified during execution and read-only
frames that remain unchanged during execution.

With the previous definition of read-only memory its straightforward to see
how changes observed on those regions can be interpreted as a fault. If we assume
the OS protection scheme implementation to be “bug free”, we can assume that
we are on the presence of a soft error. Based on this hypothesis we can formulate
the use of software change detection techniques over read-only memory as a soft
error detection mechanism. Subsection 3.1 introduces some facts about read-only
memory usage by regular software pieces and Subsection 3.2 addresses Linux
kernel usage.

3.1 R/O memory on commodity software

Any piece of software running on standard, general purpose operating system
is actually a set of object files containing libraries and executables. Compilers
and linkers work together with the OS to turn source code into object code that
can be loaded and executed in any particular platform. There are many different
file formats such as ELF (Executable and Linking Format) commonly used on
Linux systems or PE (Portable Executable) used on Windows NT systems.

Programming language evolution, enforced by modern compilers is turning
running code into a safer one. Executables built with modern compilers enforces
a separation of read-only frames from read-write ones. Distribution of read-only
vs. read-write frames depends on programs themselves. Small pieces of code that
manipulates considerable amounts of data (such as media players) present a low
ratio of R/O frames, while complex pieces of code that manipulate a small set
of data (like office suites) present a high ratio of R/O pages. Due to this fact,
coverage of the solution is not the same in every scenario. Some metrics are
presented in Section 4.3.
3.2 R/O memory usage on Linux

Standard GNU/Linux object, executable and library file format is ELF. As it happens with most object file formats, ELF supports the concept of sections, which is a collection of information of similar type. Each section represents a portion of the file. For example, executable code is always placed in a section known as `.text`, all data variables initialized by the user are placed in a section known as `.data` and uninitialized data is placed in a section known as `.bss`. On modern machine architectures, the memory manager can mark portions of memory as read-only, so that any attempt to modify a read-only memory location results in the program execution being terminated by the operating system. Thus, instead of merely saying that we do not expect a particular memory location to change, we can specify that any attempt to modify a read-only memory location is a fatal error indicating a bug in the application, ceasing its execution. Given that we want all executable portions of a binary executable in read-only memory and all modifiable memory locations (such as variables) in writable memory, it turns out to be most efficient to group all the executable portions into one section (the `.text` section), and all modifiable data areas together into another area of memory (henceforth known as the `.data` section). Experimental data related to a LAMP server is presented in Sections 4.2 and 4.3.

Another good example of read-only memory usage is found on embedded systems, where GNU/Linux is gaining market. In this context, flash memories are used for storage and there are various implementations of read only filesystems that extend flash memory life minimizing the number of erase cycles. This data is mapped as read only pages in RAM. These systems are generally assembled on platforms where there is no hardware-based error detection and correction technologies, like ECC. The benefits of software-based solutions could be of high interest to the field of embedded systems running Linux, where faults can have a direct impact in systems attributes like dependability.

4 Linux prototype implementation

In Section 2 we introduced the concept of software aging and we presented the damage it can lead to in a production environment. Among the possible causes of this phenomenon we paid special attention to transient errors occurring on system memory. We also reviewed existing tools and techniques to deal with the effects of software aging.

After that we devoted Section 3 to explain the access rights of a process over the memory resources required by the software, and how are commonly specified, making it possible to differentiate read only memory pages from those which can be modified. In that Section, we introduced how modern operating systems enforce this division, giving us tools to detect the occurrence of soft errors, one of the most common classes of memory errors. We also mentioned the role played by compilers and operating systems in assisting and guaranteeing this division over memory utilization. As an example of the latter and to get into subject, we gave an overview on Linux Read-Only memory support and treatment.
We now introduce a prototype which makes use of the hypothesis presented in section 3 to provide a system tool to fight the effects of software aging caused by memory errors. This tool was developed as an extension to the Linux Kernel, introducing additional behavior to the Memory Manager, its memory management subsystem. In section 4.1 we describe the prototype’s functionality and the most relevant details of its implementation. As its main functionality is based on the detection of hardware errors, testing the developed module is required to observe its behavior when that kind of event occurs. As we will see in section 4.2, we made use of Fault Injection techniques to achieve this goal, which allowed us to simulate hardware errors in a repeatable, cheap and secure way. Finally, in section 4.3 we present the results of some tests made in order to measure the effects of the prototype over the Linux Kernel computational performance. These tests are, as will be explained, based on benchmarking tests for Linux.

4.1 Implementation details

A Linux based prototype was built as part of the grade thesis work by González, Pedraja and Van Rompaey [7]. The first goal of this prototype was to detect the occurrence of soft errors on the physical memory of a system, by taking advantage of the existence of read-only memory regions. As we saw in section 3, Linux manages memory resources with frame granularity, so the first step to achieve our goal is to identify the subset of those frames for which the OS guarantees read-only access. The kernel is said to “trusts itself” and consequently it doesn’t implement a protection scheme in its address space. However, it does enforce one for the address space of every user space process. This protection scheme is represented by means of the access rights of the virtual memory regions that form part of a process address space and it is enforced by the process page tables. Linux’s virtual memory mechanism makes it possible for memory frames to be shared with different access rights by the address spaces of several processes. Taking this into account, we can define the subset of read-only frames which are mapped to the address spaces of one or more user space processes with read-only access rights in all cases. The task of identifying RO pages at execution time, required mastering Linux memory management subsystem. All events that changed the allocation and access rights of memory frames were carefully analyzed. By handling these events and keeping some extra state information we are able to determine at any time if a given frame belongs to the desired subset. Changes to the kernel were kept as restricted as possible and most functionality was factorized and coded as a kernel module. Figure 1

Given this, we can formulate as one of the prototype’s functionalities to identify this subset during the execution life cycle of the OS. Accomplishing this in the Linux kernel required a great investigation effort to search for all those events in which a frame was mapped or unmapped to a process address space and those in which an existing mapping access rights changed.

Once we have identified the subset of read-only frames, the second step is to provide a bit level change detection method over its elements. Any bit flip in this context can be interpreted as a soft error detection. The general solution
to provide this functionality is to keep for every frame in the set a redundancy code over the bytes belonging to the memory range it represents. In the Linux prototype this was implemented by adding a redundancy code to the state kept on every memory frame. This code was required to provide multi bit error detection in an efficient way, that is, using a small amount of redundancy bits for the target data volumes (memory frame size varies from 4KB to 4MB). After evaluating different options, the error detection algorithm selected for this task was CRC32, which was implemented by reusing kernel library functions. The redundancy kept must be updated on the events mentioned earlier, so that it is the one corresponding to the contents of the frame while it belongs to the read-only set. The strategy followed to guarantee this, was to recalculate and store the code every time the frame enters a state that places it inside the set (for example, first read-only mapping is added on a previously free frame) and to clean (or ignore) redundancy in transitions to states outside the set (such as, when a write mapping is added on a previously read-only frame). Thanks to the stored code, checking a frame for errors is as simple as recalculating the CRC32 and comparing it with the one stored.

So far, we have presented a tool capable of checking frames for soft errors, but you may be wondering when and where are those checks performed. The tool can be configured to follow one of three error search strategies. To be able to understand the reasons for this choices it is important to bear in mind that, because we are trying to react to events that have already happened, error detec-
tion is asynchronous to its occurrence and there is always a delay between them. Making that interval of time as small as possible was the goal when selecting search strategies, because it decreases the probability of a process accessing the corrupted memory section before actions can be taken. The first and most simple approach presented to achieve this goal is a dedicated kernel thread that checks repeatedly the whole read-only frame set in a cycle with the highest frequency possible. This strategy might sound a little un-performant but it has shown to be quite effective.

The prototype has some configuration functionalities that allow a user space administrator to tune it according to each scenario and its requirements. We will talk about them later, but let's give an advance on process registration which is essential for the existence of the two remaining search strategies. Process registration basically allows the administrator to configure the tool so that it focuses efforts on a selected group of processes. When the module is configured to work just with registered tasks it can perform error checks in two ways. The first is similar to the strategy already explained but in this case the thread cycles over the list of registered tasks checking only those read-only frames mapped to the address space of the task. This strategy has proven to be considerably more efficient than the previous one. Experiments conducted to compare error detection delays between these two techniques showed an average time interval of 275 milliseconds for the basic one. On the other hand the tests pointed to a 1 millisecond average delay for the registered processes strategy, even when system load was being simulated by registering the LAMP stack tasks.

The last error search strategy implemented also works with registered tasks only and was developed modifying the kernel task scheduler to check the read-only frames mapped to the address space of the next task to be assigned to the CPU, just before it actually gets that resource. The aim of this strategy is to take actions on all existing corrupt memory sections belonging to a process address space before the process gains control of the CPU and is able to access them.

Unlike the two previously presented approaches, in this case the goal is not to reduce corruption detection delay. This strategy has proven to be effective but it can also mean an important overhead on process scheduling, so it should be used with great carefullness. A tool with the functionalities mentioned so far might sound interesting, but it becomes barely useful if it doesn’t take actions to correct the detected errors or at least notify user space of their occurrence. For this reason, a sequence of error handling actions were defined as part of the prototype. The first step is the most simple and consists of employing an error correction code to try to automatically fix the affected memory. There are several types of codes widely used for this task, from which Hamming Codes were selected because of its ease of implementation. Hamming codes are actually the method most commonly used by ECC memory chips, but in this case the single error correction it provides might not be enough for a whole memory frame. With different coverage and performance characteristics Reed-Solomon
techniques could have been applied, but they were discarded due to the complexity of its implementation.

The second action taken to try to fix errors was based on the concept of rejuvenation. According to Kishor Trivedi, software rejuvenation is a proactive fault management technique aimed at cleaning up the system internal state to prevent the occurrence of more severe crash failures in the future. This technique is widely used to fight the effects of software aging. A possible solution to heal corrupted read-only memory frames using rejuvenation is to keep a backup of its contents and reload it in error conditions. Most read-only memory frames are the image of a disk file, and so that file becomes the required backup for these frames. The prototype provides a functionality that automatically reloads a frame’s file image from disk when a soft error is detected inside it. For the case of non file mapped frames (also called anonymous) a backup storage implementation should be provided, but its development exceeded the scope of the prototype.

We have managed to use rejuvenation in kernel mode, but actually most theory about this technique is aimed to user space. As many rejuvenation strategies are based on resource statistics to decide when to apply it, we understood that in this case the role of the OS was to provide error detection information and notification in order to assist such decisions. First, we decided to provide user space software rejuvenation agents with synchronous notifications of error detections. This feature was implemented by means of Linux signals. When a process is registered on the tool, another process is also specified and takes the role of the registered task’s rejuvenation agent. In case of error detection on the tasks address space, a specific soft error signal is sent to its agent so that actions can be taken.

Sometimes rejuvenation simply consists of processes or even system restart, however, if the required information is available, higher granularity actions (such as remapping a single file) can be taken. To help agents recognize the opportunity to take such actions, the prototype publishes to user space detailed information of the error detected. Those details include for each affected process (registered or not) the exact virtual and physical address of the bytes changed, the virtual memory area in the process address space to which the address belongs, and which type of mapping (file, anonymous) this corresponds to. In case the error occurred on a file mapping the file name and mapped range are also provided. Finally, the resulting state of the error (fixed or not) is also shown. The user space interface selected to present this information was the /proc virtual filesystem, using task level folders and files.

We mentioned the prototype also provided some configuration functionalities, such as the already explained registered tasks. The module also supports the concept of a global mode which is composed by a set of flags. Each of these flags allows the administrator to activate/deactivate particular features. Flags have been created to choose from error search possible strategies and to turn on and off error handling actions or even the whole module. The user space interface to configure mode and registered task was also implemented using /proc virtual filesystem by writing special files.
We have been using the term *module* across this section to refer to prototype implementation, but haven’t explained how it actually integrates with the Linux kernel. As it is implied by the same term, all the functionalities presented are grouped in a single module in the kernel. The same provides a group of interfaces with a set of callbacks which enable it to handle kernel internal events and user space interface. These callbacks were designed to reduce as much as possible kernel coupling with the prototype. When we say module, we don’t refer to the dynamic module concept the kernel provides. Such modules are able to add functionality to an active kernel instance and are mainly used to develop device drivers. In this case, because of the sections of the kernel where the module interfaces are called, we need to add functionality to the kernel at build time. Because of the kind of functionality it provides, we decided to include the module sources as part of the Linux kernel Memory Management subsystem.

4.2 Prototype testing

Verification of prototype correctness was relevant for several reasons. On one hand, there is always the need of adding quality to the implementation being tested, but, on the other hand, we needed to separate prototype bugs from soft errors in memory. The tests consist of altering memory values located in read-only memory frames and verifying that they can be detected and corrected. Designing test cases was not an easy task. We did not have the means to inject hardware errors into memory subsystems, so we addressed error injection through software. Detailed technological intricacies are beyond the scope of this paper, but we just mention that high level kernel API cannot be used to alter read-only memory frames (the kernel is too smart). Altering read only memory values from userspace generates segmentation faults. Altering read only memory values from portable, high level API (like *ptrace*) generates unwanted effects: the kernel turns the read only frame into a read-write one before affecting the change. As soon as the page is turned into a read-writeable one, it is removed from the set of frames monitored by our implementation.

Software injection of errors has to be done in supervisor mode, where unrestricted access to physical memory is obtained and raw memory modifications can bypass kernel memory protection scheme. We achieved this goal with a little kernel programming, by addressing memory directly from kernel virtual address space, avoiding user processes page tables access limitations. We exposed this functionality into userspace through a new syscall (named *sys_kpgsa_inject()* and wrote different high level tools. We can alter single byte memory positions or even simulate periodical and random memory failures (cosmic rays rain).

4.3 Benchmarking

Performance considerations are central to the success of the prototype. Despite of reliability, availability and safety consequences of automatic and general-purpose software rejuvenation, it has to remain performant. In order to monitor changes, system memory has to be completely scanned periodically, turning the prototype
into a memory/CPU-bound application, with no IO impact. This introduces competition for memory access and loss of locality.

Fig. 2. Performance impact on different applications

We addressed two different application scenarios: CPU bound applications and IO bound ones. The CPU bound application selected is POV-Ray [8], and the IO bound one is a LAMP\(^1\) server. To measure the impact on system performance we used their own benchmarking tools, comparing their throughput on the same system with and without memory corruption correction routines. When memory corruption and correction routines are enabled, they can be applied to all read only frames of the whole system (mode 142) or to a subset of registered processes (mode 146). Tests were run in both modes.

Benchmarked results are consistent with intuition, as shown in Figure 2. Performance impact on a LAMP server is marginal, as it can be seen on Figure 2(a), while the impact on raytracing is significant, as shown in Figure 2(b). In other words, memory corruption correction routines almost do not compete with IO bound task while they do with CPU bound ones. Despite of the degradation measured, results are promising. Implementation is already applicable in different scenarios.

<table>
<thead>
<tr>
<th></th>
<th>MySQL</th>
<th>Apache httpd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Father</td>
<td>Child</td>
</tr>
<tr>
<td>RO frames</td>
<td>287</td>
<td>47%</td>
</tr>
<tr>
<td>RW frames</td>
<td>330</td>
<td>53%</td>
</tr>
</tbody>
</table>

Fig. 3. RO / RW memory frames on used software

\(^{1}\) Linux, Apache, MySQL, PHP
As a last remark regarding the impact on running software, not all systems have the same read only/read write ratio. Analyzing Apache and MySQL from previous experiments, we found that both share the same fork based client attention mechanism. Father processes have a higher ratio than children processes as we can see from Figure 3. Observed results are consistent with the idea that father processes mostly do not contain actual execution data used for answering a single client request. The processes that handle requests are children ones. Father processes are longer lived than children in this scenario, and those are the processes that get the most important benefit from the memory corruption correction algorithms.

The impact of a soft error on a child process, may affect only a single session while a similar error on a father process may affect the whole service from there on. Corruption detection has more relevance on father processes, where RO/RW ratio is higher.

5 Final remarks and future research

There are still different tuning parameters identified that should be added to the implementation in order to become more customizable to different application scenarios. \texttt{freqd} and other CPU equalization daemons have to be considered in order to reach mobile platforms too. Current implementation would keep notebook CPU running at 100%, consuming the battery too fast. Interaction with energy management subsystem is a must.

Ports to different platforms are required too. So far x86 and x86,64 architectures were considered during implementation and only x86 used for testing. Ports to smaller platforms are required too. Linux usage on embedded systems, where ECC memory is not commonly used, is an area where impact can be significant.

Current implementation proved its capacity of enhancing the availability of a LAMP server, under heavy load, under simulated heavy cosmic rays rain. A system without read only memory corruption correction implementation fails after a few seconds of random modifications on its memory. The memory corruption correction showed that it is possible to make a system reload fresh software images into RAM when corruption occurs, without impact on services provided by it.

As a final remark, we still need to ascertain that the implementation is able to detect an correct real soft errors. Our estimations [7] indicate that on state of the art individual systems (4GB DDR2 memory), there will be one correctable error, addressable by our prototype, every 22 years on average per system. We have not found yet any error on patched kernels in production environments. It is desirable to gain access to environments where error rate is higher.
6 Conclusion

Industry already knows about software aging: when a system starts having an erratic behavior, first reboot it. Software rejuvenation techniques have been gaining a more significant position in system administration in recent years and different products introduce them as a part of their enhanced availability solutions with different levels of granularity. As far as we are aware, only full system or full process rejuvenation were addressed before this work. Our prototype proved that finer grained rejuvenation is addressable, as an operating system tool, in general purpose application environments. Even though implemented solution only addresses read-only memory frames, it does not require any changes at deployment levels, as it is the case of techniques based on replicas and load balancing. Due to the fact that it is applicable to any piece of software without any changes, it offers a simple and non-intrusive way of reducing production environment management costs.

We cannot replace ECC memory with a software based solution yet, but our memory corruption correction implementation is able to enhance system availability where ECC is not available. Even though more research is required, we showed that it is possible to enhance arbitrary application availability and reliability with general fine grained rejuvenation techniques implemented at operating system level, with reasonable impact on performance.

References