Monitoring Memory-Related Software Aging: An Exploratory Study

Rivalino Matias  Bruno Evangelista  Autran Macedo

School of Computer Science
Federal University of Uberlândia
Uberlândia MG, Brazil
AGENDA

• Introduction
• Memory Leaking and Software Aging
• Monitoring Aging Effects of Memory Leaks
• Conclusions
Introduction

• Many experimental researches have focused on characterizing the aging phenomenon and evaluating the effectiveness of rejuvenation mechanisms.

• For both cases monitoring aging effects is essential.

• In this work, we investigate important aspects related to monitoring memory-related software aging effects, especially related to memory leaks.
Research Goal

• The correct understanding of software aging is a major requirement to make educated decision and providing reliable system diagnostics.

• Our goal is to present a practical body of knowledge to support the solid understanding of important aspects of monitoring aging effects related to memory leaks.

• As an exploratory study, this work discusses findings that will assist experimenters to choose better strategies for measuring aging effects.
Research Scope

- Software aging effects related to main memory are the most cited in the literature.

- Fundamentally, memory-related aging effects are caused by memory leak or memory fragmentation problems.
  - Memory leak is mainly caused by inadequate use of memory management routines.
  - Memory fragmentation is a consequence of the system operation dynamics.
Research Scope

• Memory fragmentation has gained more attention in the recent software aging studies.

• However, the majority of experimental researches in this field still focusing on memory leaking.

• The accurate detection and measurement of memory leaks are not trivial tasks.
  – it requires a deep understanding on the underlying mechanisms behind this problem.
Memory Leak

- Memory leaking occurs when the application process does not release previously allocated memory blocks that will not be used anymore.

- There are several reasons for this happen, and we propose to classify them in two main types:
  - involuntary and voluntary non-releasing of memory.
Memory Leak

- **Involuntary** non-releasing of memory occurs when a process is unable to release a previously allocated memory block.
  - it happens mainly due to losing the reference (address) to the allocated block.
  - Another example is the unbalanced use of `malloc (or new)` and `free (or delete)` routines.
Memory Leak

... ptr1=malloc(60)
ptr2=malloc(250)
...
...
...
...
...
...
ptr1=malloc(1000)

Leaked chunk !!!
Since this is not referenced anymore, it cannot be freed calling free(ptr1)
Memory Leak

• **Voluntary** non-releasing of memory is related to the software design.

• In order to improve performance, some applications preallocate pools of memory blocks.
  – to avoid the computing costs of repetitive operations for allocating and releasing memory objects.
  – the size of resource pools usually increases monotonically.

• We can find this *design pattern* in many nowadays server applications.
Memory Leak

- The practical result in both situations is the application process gradually consuming more memory during its runtime.
  - in long-term executions this may exhaust the computer main memory.
Memory Leak

• Since the memory leaking causes the heap saturation, additional memory has to be requested to the OS.

• Hence, a new memory area (new heap) is added to the process address space
  – making the process size to grow!

![Memory Segments Diagram]
Conclusion #1

• Based on this observation, an important conclusion can be drawn:

  whereas the process heap is not enlarged the process size still unchanged, even under occurrences of memory leaks.
Aging Indicators

- Software aging effects can be detected only during the system’s runtime.
  - through monitoring aging indicators.

- Aging indicators are variables that represent the stable state of a given system.
  - comparing the monitored aging indicators with standard values for a given system could help to reveal the presence of aging effects.

- The quality of aging indicators affects the effectiveness of rejuvenation mechanisms
  - given that their efficiency depends on the activation time that is influenced by the accuracy of the chosen aging indicators.
Aging Indicators

• Aging indicators can be considered at different system levels:
  – application’s components, application’s process, middleware, operating system, virtual machine, hypervisor, ...
  – since software aging can occur inside all of these layers, it is important to adopt the appropriate indicators for each layer.

• In general, aging indicators are classified in two categories according to their granularity:
  – system-wide and application-specific.
System-wide Aging Indicators

• These indicators provide information related to subsystems that are shared and influenced by other system components.

• Examples of shared subsystems
  – Application middleware, operating systems, VMs, and Hypervisors.

• Indicators of this category are used to evaluate the aging effects in the system as a whole.
  – e.g., total free/used memory, free/used swap space, # of open files, # of processes, # of soft interrupts, …
These indicators provide specific information about an individual application process.

Examples of aging indicators in this category are:

- Process’ resident set size (RSS), JVM heap size, and application’s response time.

If the application of interest is not running directly under the OS, but inside of a virtual machine platform, these aging indicators could be applied indirectly, targeting the process running the virtual platform (e.g., Java and C#).
Aging Indicators Monitoring

• For both classes of aging indicators, it is possible to collect data from the user-level and kernel-level.

• Collecting data from user-level is implemented by special-purpose programs that run at the user level.
  – at this level the system-wide monitoring is limited to the information that the OS exports to the user level.

• It is possible to collect data from the kernel level through an instrumented kernel
  – Kernel changes for this purpose can be done either statically or dynamically.
Using System-wide indicators

- They are adequate when there is no previous knowledge of the system under investigation, offering a first overview.

- This is particularly important when working with a new or unknown system and many candidate variables are preliminarily considered.

- Many previous research works in software aging and rejuvenation have used this approach.
Using System-wide indicators

• Due to the shared nature of system-wide indicators, they usually present significant noises.

• Specific on memory leak, the most frequently used system-wide aging indicators have been the free or used main memory.

• Only monitoring these indicators is very hard to make a precise diagnostic if the system suffers or not from memory leaking.
Experiment #1

- We run a workload emulating typical file system operations executed in webserver systems.

- We use the well-known *filebench* workload generator for this purpose.

- We adopted three types of loads:
  - low (w1), moderate (w2), and high (w3).
Experiment #1

Table 1. Workload Profiles (filebench.f)

<table>
<thead>
<tr>
<th>Profile</th>
<th>Workload Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td>#files= 5000       #threads= 25</td>
</tr>
<tr>
<td>w2</td>
<td>#files=10000       #threads= 50</td>
</tr>
<tr>
<td>w3</td>
<td>#files=20000       #threads=100</td>
</tr>
</tbody>
</table>

- The three workloads are executed twice, in a cyclical way.
- Each run lasts for one hour. Total time per experiment is six hours.
- We repeated each experiment 10 times to minimize the experimental errors.
- Thus, the discussed results are based on the averaged values.
Experiment #1

![Graph showing memory consumption over time with equation $y = 4134.5x + 2E+06$.]

Figure 1. Pattern of increasing memory usage observed in memory leak scenarios.
Experiment #1

- As can be observed, the results indicate an increasing trend in the memory consumption.

- This pattern is frequently observed in experimental tests using system-wide aging indicators (e.g., used physical memory).

- However, analyzing carefully the dataset we note that this trend is not consequence of software aging, but simply the effect of the OS disk cache (so-called buffer-cache).
Experiment #1

- Since the adopted workloads are disk I/O-intensives, the OS kernel memory management requests unused physical memory to the page-level allocator in order to create additional disk cache objects.

- This behavior is expected considering that:
  - i) there is free physical memory available in user level, and
  - ii) the disk I/O subsystem is under pressure.
Experiment #1

- In such scenario, it is not uncommon to observe the amount of free memory decaying considerably.

- This behavior may easily be misinterpreted as memory leaking, where in fact we have the available memory being temporarily moved to the OS buffer-cache subsystem.
  
  – note that the used memory can be made available to the user level again as soon as it is necessary.

- Looking at free/used memory alone or combined with other noisy system-wide aging indicators (e.g., swap space) may lead to erroneous conclusions about memory leaks.
Experiment #1

- For cases like this, we suggest to monitor not only the used/free memory, but also comparing it to the memory used in `buffer-cache` and the total memory allocated for the `user-level`.

- These combined analyses will help to understand the memory flow inside the system, and see if leaks are really occurring.

- This approach is especially important to avoid misdiagnosis when using memory-related system-wide aging indicators.
Experiment #1

Figure 2. Monitoring combined system-wide memory related aging indicators
Experiment #2

- Preferentially, for well-known systems we recommend to compare the values of aging indicators with a baseline known to be aging free.

- To illustrate this we conduct a second experiment.

- In Exp. #2 we execute an application process along with the workload used in Exp. #1 running in background.

- Exp. #2 is composed of four tests (#2.1 – #2.4).
**Experiment #2**

- In #2.1 we monitor the total used memory while running a leak-free application (our baseline).

- In #2.2 we inject random memory leaks only in application level.

- In #2.3 we inject random memory leaks only in kernel level.
  - through a faulty kernel module.

- In #2.4 we inject memory leaks in both application and kernel levels simultaneously.
# Experiment #2

<table>
<thead>
<tr>
<th>Algorithm 1. No memory leaking</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$: sleep time in seconds</td>
</tr>
<tr>
<td>$f$: multiple of page size</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>$t = \text{random (1..30)};$</td>
</tr>
<tr>
<td>$f = \text{random (1..5)};$</td>
</tr>
<tr>
<td>$c = \text{malloc (1024 * } f);$</td>
</tr>
<tr>
<td>if (c is equal to NULL) then break;</td>
</tr>
<tr>
<td>for each position in c</td>
</tr>
<tr>
<td>c[position] = 0;</td>
</tr>
<tr>
<td>sleeps for $t$ seconds;</td>
</tr>
<tr>
<td>free (c);</td>
</tr>
<tr>
<td>end loop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm 2. Memory leaking</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$: sleep time in seconds</td>
</tr>
<tr>
<td>$f$: multiple of page size</td>
</tr>
<tr>
<td>$k$: luck number</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>$t = \text{random (1..30)};$</td>
</tr>
<tr>
<td>$f = \text{random (1..5)};$</td>
</tr>
<tr>
<td>$c = \text{malloc (1024 * } f);$</td>
</tr>
<tr>
<td>if (c is equal to NULL) then break;</td>
</tr>
<tr>
<td>for each position in c</td>
</tr>
<tr>
<td>c[position] = 0;</td>
</tr>
<tr>
<td>sleeps for $t$ seconds;</td>
</tr>
<tr>
<td>$k = \text{random (even..odd)};$</td>
</tr>
<tr>
<td>if ($k$ is even) then free (c);</td>
</tr>
<tr>
<td>end loop</td>
</tr>
</tbody>
</table>
## Experiment #2

### Table 2. Test Profiles for Experiment #2

<table>
<thead>
<tr>
<th>Test</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>No memory leaking (baseline)</td>
</tr>
<tr>
<td>2.2</td>
<td>Memory leak inside the App.</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory leak inside the OS kernel</td>
</tr>
<tr>
<td>2.4</td>
<td>Memory leak inside the App. &amp; OS kernel</td>
</tr>
</tbody>
</table>
Monitoring Aging Effects of Memory Leaks

Experiment #2

\[ y = 4429.4x + 2E+06 \]

\[ y = 4506x + 2E+06 \]

\[ y = 4438.4x + 2E+06 \]

\[ y = 4453.6x + 2E+06 \]
Experiment #2

- As can be seen, all tests of Exp. #2 show similar results.

- The background workload forces the OS disk caching effect
  - that dominates the memory usage variability in the system, thus
    hiding most of the aging effects injected.

- In average, the predominant size of memory allocations is
  less than 64 bytes, which is easily obfuscated by the larger
  variability of many usual background workloads
  
  - e.g., in Exp. #2 the average amount of memory leaked was 200
    kilobytes within six hours.
Conclusion #2

- We can conclude that system-wide aging indicators suffer different influences (e.g., background tasks), making their quality poor.

- If you need to use them, we recommend adopting a baseline in order to reduce (not avoid) the risks of false positives.

- Application-specific aging indicators are a better alternative to offer noise-free information.
Using Application-specific indicators

- Specially for detecting memory leaks, we proposed [Matias et al. 2006] the use of the process’ resident set size (RSS) as aging indicator.

- This indicator allows us to monitor the specific amount of main memory that a given application process is using.

- Monitoring this indicator is more effective than using system-wide indicators such as free or used main memory.
Using Application-specific indicators

• However, in recent findings we learned that memory leaks do not increase the process resident size (RSS) immediately to their occurrences.
  – because the process size is only increased when a new memory area is requested to the OS in order to enlarge the saturated heap.
  – it could take some time to observe leaking effects only monitoring the RSS

• As a result, now we know that the RSS may present some detection delay.
  – this delay depends on the memory allocator algorithm used.
Experiment #3

- In this experiment, we execute a program that initially allocates 1024 blocks of 1024 bytes (1 megabyte in total).
- Then we free the first 512 blocks and allocate new 512 blocks, but now leaking all of them.
Experiment #3

```c
char *p[1024];
main(){
    unsigned long j=0,i=0,l=0;

    for(i=0; i<1024;i++){
        p[i]=malloc(1024);
        if ( p[i]==NULL) exit(0);
        for(l=0;l<1024;l++) p[i][l]='*';
    }

    for(i=0;i<512;i++) free(p[i]);

    for(i=0; i<512;i++){
        p[0]=malloc(1024);
        if ( p[0]==NULL) exit(0);
        for(l=0;l<1024;l++) p[0][l]='*';
    }
}
```

Allocate 1 megabyte

Release 512 kilobytes

Allocate 512 kilobytes, leaking them
Experiment #3

```c
char *p[1024];
main(){
  unsigned long j=0,i=0,l=0;
  RSS= 224 kB
  for(i=0; i<1024;i++){
    p[i]=malloc(1024);
    if ( p[i]==NULL) exit(0);
    for(l=0;l<1024;l++) p[i][l]='*';
  }
  for(i=0;i<512;i++) free(p[i]);
  for(i=0; i<512;i++){
    p[0]=malloc(1024);
    if ( p[0]==NULL) exit(0);
    for(l=0;l<1024;l++) p[0][l]='*';
  }
}
```
Experiment #3

```c
char *p[1024];
main(){
    unsigned long j=0,i=0,l=0;

    RSS= 224 kB
    for(i=0; i<1024; i++){
        p[i]=malloc(1024);
        if ( p[i]==NULL) exit(0);
        for(l=0;l<1024;l++) p[i][l]='*';
    }
    RSS= 1276 kB

    for(i=0; i<512; i++) free(p[i]);

    for(i=0; i<512; i++){
        p[0]=malloc(1024);
        if ( p[0]==NULL) exit(0);
        for(l=0;l<1024;l++) p[0][l]='*';
    }
}
```
Experiment #3

```
char *p[1024];
main()
{
    unsigned long j=0,i=0,l=0;

    RSS= 224 kB
    for(i=0; i<1024; i++){
        p[i]=malloc(1024);
        if ( p[i]==NULL) exit(0);
        for(l=0;l<1024;l++) p[i][l] = '*';
    }
    RSS= 1276 kB
    for(i=0; i<512; i++) free(p[i]);
    RSS= 1276 kB
    for(i=0; i<512; i++){
        p[0]=malloc(1024);
        if ( p[0]==NULL) exit(0);
        for(l=0;l<1024;l++) p[0][l] = '*';
    }
}
```
Experiment #3

```c
char *p[1024];
main(){
    unsigned long j=0,i=0,l=0;
    RSS= 224 kB
    for(i=0; i<1024; i++){
        p[i]=malloc(1024);
        if ( p[i]==NULL) exit(0);
        for(l=0;l<1024;l++) p[i][l]='*';
    }
    RSS= 1276 kB
    for(i=0;i<512;i++) free(p[i]);
    RSS= 1276 kB
    for(i=0; i<512;i++){
        p[0]=malloc(1024);
        if ( p[0]==NULL) exit(0);
        for(l=0;l<1024;l++) p[0][l]='*';
    }
    RSS= 1276 kB
}
```
Experiment #3

- The common sense would expect that right after releasing the memory, it would be returned to the OS (reflecting on the RSS).

- The observed behavior is specific for the memory allocator used, `ptmallocv2`, which moves the released blocks to its heap’s free lists in order to keep them for a possible future use.

- Note that replacing the allocator the results will be different.

- In addition to the RSS detection delay, now we know that not all leaking scenarios are captured monitoring only the RSS.
Conclusions

- Our experiments indicate that a precise memory leak monitoring should be done inside the process’ heap rather than outside, as it has been implemented so far.

- Monitoring the RSS is better than other system-wide aging indicators, but also presents drawbacks.
  - it could not reveal precisely the correct memory leak rate, due to detection delays and the influences of the allocator design.

- Hence, only analyzing the RSS is also not sufficient to conclude about the existence of memory leaks.
Suggestions

- To verify if the increasing of RSS should be considered or not a consequence of memory leaks, we propose:
  - testing the application in a controlled environment, where it is possible to reduce the amount of physical memory.
  - If the process size still growing behind the limits of the physical memory, i.e., increasing its virtual memory size (VSZ), then it must be considered a memory leak problem.
  - If not, then the VSZ should not be significantly larger than the RSS, which is limited to the available physical memory.
  - Therefore, we consider that monitoring the RSS in conjunction with the process virtual size (VSZ) is a better strategy than using only the RSS.
Thank You!

Rivalino Matias Jr.
rivalino@fc.ufu.br